UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Enhancing Software Quality and
Quality of Experience through User Interfaces

"Mejora de la Calidad del Software y de la
Experiencia del Usuario a través
de las Interfaces Humano-Máquina"

D. Pedro Luis Mateo Navarro

2014

A mis padres y a mis dos hermanos.
Por ser un ejemplo continuo,
por ser un apoyo constante,
por tanto que agradecer,
por nada que reprochar.

Resumen de la Tesis

1 Introducción

Conseguir la «Calidad en el Software» es un **objetivo muy amplio** que abarca varias disciplinas, desde aquellas centradas en la evaluación de los aspectos mas técnicos del software, hasta aquellas encargadas de analizar ciertos aspectos psicológicos de los seres humanos. La calidad en una aplicación o un sistema puede lograrse de varias maneras, como por ejemplo comprobando que se cumplen los requisitos funcionales previamente descritos, validando la salida de un software, analizando la calidad percibida por los usuarios, etc. El objetivo de esta tesis es mejorar la calidad del software centrándose principalmente en el **análisis de las interfaces de usuario** y en la **evaluación de la interacción entre los usuarios y el sistema**.

Esta **interacción** entre los usuarios y el sistema ha **cambiado** rotundamente en los últimos tiempos. No hace muchos años, la interacción de las personas con los ordenadores estaba restringida sobre todo a entornos de escritorio (principalmente para informarse o divertirse) o para realizar tareas en el trabajo, en entornos educacionales o en el ámbito doméstico. En estos sistemas, un ratón y un teclado eran suficientes para comunicarse con el software, el cual se comunicaba con nosotros utilizando la pantalla y unos altavoces. Este tipo de interacción se realiza a través de la interfaz gráfica de usuario (*Graphical User Interface*, GUI), la cual es la única modalidad mediante la cual el usuario puede comunicarse con el sistema.

Hoy en día, sin embargo, estamos interactuando continuamente con los llamados "dispositivos inteligentes". Estos dispositivos nos proporcionan acceso a toda la información, nos proporcionan una conectividad completa con otras personas, y además cuentan con una

alta capacidad de procesamiento. Están en nuestros bolsillos, en nuestras manos, e incluso colocados en nuestras cabezas como si de unas simples gafas se tratara. Estos dispositivos suelen proporcionar diferentes modos de interacción mediante el uso de diferentes modalidades sensoriales, y nos permiten comunicarnos con ellos, e.g., tocando la pantalla con nuestros dedos, usando nuestra voz, o simplemente agitándolos o girándolos con nuestras manos. Este tipo de interacción se denomina multimodal y realiza a través de diferentes interfaces (e.g., la pantalla, el micrófono de entrada, el sensor de acelerómetro, etc.)

2 Objetivos, Metodología y Resultados de esta Tesis Doctoral

Como comentábamos más arriba, esta tesis doctoral se centra en el análisis de las interfaces de usuario y la evaluación de la interacción entre los usuarios y el sistema. Para llevar a cabo este análisis, esta tesis plantea **dos enfoques diferentes**.

Por una parte evaluamos la calidad de los diferentes componentes del proceso de interacción **por separado** (i.e., la entrada proporcionada por los usuarios y la salida ofrecida por el sistema). La principal ventaja de este enfoque es que las herramientas y los métodos utilizados pueden centrar todo su potencial en conseguir la calidad de uno de los componentes de la interacción, y como consecuencia se consigue mejorar la calidad del software.

Por otra parte evaluamos la calidad de la interacción **como un todo**, como un flujo único de acciones desde el usuario hacia el sistema y viceversa. La principal ventaja de los métodos que siguen este enfoque es que se centran en conseguir la calidad de todo el proceso, y no de las partes que lo componen. Así podemos mantener la "totalidad" del proceso de interacción, lo que nos permite analizar las relaciones causa-efecto que se producen entre las acciones del usuario y las acciones del sistema.

Al hilo de estos dos enfoques, esta tesis doctoral plantea **cuatro grandes objetivos** de investigación. Estos objetivos se dividen en dos bloques dependiendo de si los componentes del proceso de interacción se analizan de forma separada (Bloque 1) o si por el contrario se analizan como un todo (Bloque 2). De acuerdo con estos cuatro objetivos de investigación, la espina dorsal de esta tesis se estructura en cuatro grandes capítulos, i.e., Capítulos 3, 4, 5 y 6 en este documento. Cada uno de estos capítulos describe todas aquellas actividades de investigación llevadas a cabo para intentar conseguir estos cuatro objetivos, así

como el diseño y la implementación de los métodos y herramientas resultantes de dicha investigación.

Empezando con el Bloque 1, el **Objetivo G1.1** trata de encontrar un framework de pruebas que permita el desarrollo de herramientas destinadas a validar la respuesta de un software. Este framework debe permitir la automatización de aquellas pruebas GUI que normalmente son realizadas a mano por los expertos (e.g., llevar a cabo una serie repetitiva de acciones sobre un conjunto de botones), así como permitir la simulación de las acciones de un operador humano con el objetivo de proporcionar un proceso de pruebas real, fiable y robusto. También es un objetivo facilitar la integración de estas herramientas en desarrollos de diferentes tipos.

Para conseguir este objetivo, esta tesis doctoral propone un framework de pruebas llamado **OHT** (Open HMI Tester, *Probador de Interfaces Adaptable*). Este framework presenta una arquitectura abierta y adaptable con el objetivo de soportar el desarrollo de diferentes herramientas de pruebas GUI, así como para ser adaptado a diferentes entornos de ejecución. OHT proporciona varios módulos que pueden ser fácilmente extendidos y/o adaptados para trabajar en diferentes entornos de pruebas (e.g., para probar aplicaciones basadas en una plataforma GUI diferente). Esta herramienta se basa en el enfoque captura/reproducción, a partir del cual se pueden automatizar diferentes procesos de pruebas GUI basados en la introspección, en la captura de las acciones de un operador humano, y/o en la ejecución de acciones dentro de la aplicación testada.

Tras diseñar una solución destinada a la validación de la salida del software, el **Objetivo G1.2** se centra en buscar una alternativa ligera y de fácil integración para validar las entradas proporcionadas por los usuarios dentro de una GUI. Las entradas de los usuarios tienen que ser válidas y deben ajustarse a unos requisitos concretos. Estos requisitos deberían poder ser escritos en diferentes lenguajes de especificación dependiendo de las necesidades de cada proyecto. Además, el proceso de verificación debería de ser un proceso interactivo que facilite el trabajo de los desarrolladores y los expertos durante el desarrollo del software, así como la interacción de los usuarios durante su uso.

De acuerdo a estos requisitos, esta tesis propone **S-DAVER** (Script-based DAta VERification, *Verificación de Datos Basada en Scripts*), un framework de verificación en tiempo real que permite comprobar la validez de los datos de entrada al mismo tiempo que el usuario utiliza la aplicación. Esta herramienta proporciona un proceso interactivo en el que los usuarios son notificados sobre los errores cometidos de forma instantánea. Todos los

procesos de verificación están encapsulados en una capa independiente, la cual una vez integrada en la aplicación, establece una relación de confianza entre la GUI y la lógica de negocio. S-DAVER proporciona un entorno de verificación que se integra en los procesos de desarrollo, pruebas y uso del software.

En el Bloque 2 se analiza el proceso de interacción como un todo. Para ello, el **Objetivo G2.1** intenta encontrar una forma de describir la interacción multimodal entre el usuario y el sistema, con la finalidad principal de permitir su instrumentación y evaluación. Esta descripción tiene que ser genérica, y debe permitir la comparación entre diferentes registros de interacción independientemente del contexto en el que fueron obtenidos. También se debería conservar la naturaleza dinámica del proceso de interacción.

Con este objetivo, esta tesis doctoral presenta **PALADIN** (Practice-oriented Analysis and Description of Multimodal Interaction, *Enfoque Práctico para el Análisis y Descripción de la Interacción Multimodal*). PALADIN es un modelo runtime que engloba una serie de parámetros destinados a describir qué ocurre en el proceso de interacción en entornos multimodales. Estos parámetros son anotados paso a paso con el fin de proporcionar una descripción dinámica de la interacción, manteniendo así el carácter de diálogo entre usuario y sistema. Como resultado, las instancias de PALADIN proporcionan un criterio común con el que describir, analizar, evaluar y comparar la interacción en diferentes sistemas, ya sean multimodales o unimodales. Estas instancias también son válidas para evaluar la usabilidad de dichos sistemas.

Basado en la idea de PALADIN, el **Objetivo G2.2** plantea ir más allá y proporcionar un framework que permita la evaluación de las experiencias de los usuarios en entornos móviles. Por una parte debe proporcionarse una forma genérica de describir el contexto que rodea a los usuarios y al sistema, así como su evolución en el tiempo. Por otra parte es necesario capturar las impresiones y otros juicios subjetivos de los usuarios sobre el proceso de interacción. Este framework debe proporcionar un método común con el que evaluar la usabilidad de sistemas y aplicaciones móviles, así como la calidad de la experiencia (QoE) de los usuarios.

Conforme a estos requerimientos, esta tesis presenta **CARIM** (Context-Aware and Ratings Interaction Model, *Modelo de Interacción, Contexto y Valoraciones de los Usuarios*). CARIM es un modelo runtime que describe la interacción usuario-sistema, el contexto en el que ésta se produce, y las valoraciones de los usuarios sobre la calidad de dicha interacción. Al igual que PALADIN, CARIM se basa en un conjunto de parámetros. Éstos se organizan en

una estructura común con el objetivo de proporcionar un método uniforme para describir, analizar, evaluar y comparar las experiencias de los usuarios independientemente del sistema que estén utilizando, el contexto que les rodea, o las diferentes modalidades que estén utilizando. Además, la naturaleza dinámica de CARIM permite a las aplicaciones tomar decisiones basadas en el contexto o la QoE actual con el objetivo de adaptarse, y así proporcionar una mejor experiencia a los usuarios.

3 Conclusiones

Con el fin de intentar lograr los objetivos de investigación que se proponen en esta tesis doctoral, el trabajo presentado en este documento engloba diferentes formas de obtener la calidad de un software, centrándose principalmente en el análisis de la interacción entre los usuarios y el sistema. Este trabajo representa también un paso adelante en la definición de métodos más genéricos, abiertos y adaptables para el análisis y la evaluación de la interacción usuario-sistema, de los elementos que la componen, y del contexto en el que ésta se produce.

Durante esta tesis se han analizado diferentes enfoques y propuestas en los campos de Desarrollo y Pruebas Software, así como de Análisis y Evaluación de la Interacción Usuario-Sistema, con el fin de identificar los principales problemas a la hora de conseguir la calidad de un software.

Esta tarea es un proceso complicado y costoso en tiempo y en recursos técnicos y humanos. El uso de herramientas para automatizar los procesos de pruebas software es muy común y suele presentar resultados satisfactorios. Sin embargo, el desarrollo e integración de este tipo de herramientas en entornos reales es una carrera complicada y llena de obstáculos. En esta tesis se ha tenido especial cuidado con el diseño de las diferentes herramientas, proponiendo siempre arquitecturas adaptables con el fin de ampliar el rango de escenarios sobre los que éstas puedan ser aplicadas.

El proceso de pruebas sobre interfaces de usuario puede decirse que es aún más complejo que sobre otros componentes del software. Esto se debe principalmente a las características particulares de dichas interfaces, las cuales requieren de herramientas especiales para llevar a cabo estos procesos.

Como se comenta más arriba, el usuario y el sistema pueden comunicarse mediante el uso de diferentes modalidades sensoriales. El entorno de pruebas se vuelve por tanto

más complejo debido a la aparición de elementos adicionales tales como los sistemas de interfaces gráficas, los reconocedores y sintetizadores de voz, el uso de dispositivos externos para reconocer las acciones físicas del usuario, etc. En esta tesis se han considerado los componentes de la interacción y sus elementos (i.e., contenido de la entrada y salida) independientemente de las modalidades o plataformas de interfaz utilizadas para su generación, transmisión y recepción. En el caso de interfaces multimodales, se ha trabajado en buscar equivalencias entre las distintas modalidades y así tratarlas al mismo nivel de abstracción.

La evaluación de las interfaces de usuario normalmente implica llevar a cabo el análisis de un proceso en tiempo real durante el cual los usuarios intercambian información con el sistema en un contexto determinado. Los procesos de análisis y pruebas deben ser diseñados e implementados para conseguir la mayor eficiencia posible. Este aspecto es crítico, ya que la ejecución de estos procesos no debe afectar la interacción de los usuarios ni el rendimiento del sistema durante la fase de pruebas. La eficiencia de estos procesos ha sido un aspecto clave y una de las principales preocupaciones a la hora de diseñar e implementar las herramientas propuestas en esta tesis.

Cuando evaluamos interfaces de usuario nos encontramos con otros problemas como los que describimos a continuación. Uno de ellos es decidir que partes de las interfaces serán evaluadas, así como decidir qué datos serán incluidos en el proceso de análisis. Esta etapa es crítica si se quiere proporcionar un proceso de pruebas y análisis eficientes. Otro problema es el bajo nivel de estandarización que existe entre las metodologías utilizadas para análisis y pruebas sobre interfaces gráficas. Finalmente comentar que las interfaces de usuario son elementos software que tienden a cambiar frecuentemente durante el proceso de desarrollo debido, principalmente, a su adaptación a las necesidades de los usuarios. Esta naturaleza cambiante dificulta la creación y mantenimiento del conjunto de pruebas.

4 Resultados

Como resultado de esta tesis doctoral, y como prueba de concepto de este trabajo de investigación, en este documento se presenta el diseño e implementación de cuatro herramientas software: OHT, S-DAVER, PALADIN y CARIM. Estas herramientas han sido publicadas abiertamente como una contribución a la Comunidad Open-Source. Éstas pueden ser descargadas para ser usadas con aplicaciones reales, ser adaptadas a nuevos entornos de ejecución o mejoradas en futuras actividades de investigación y desarrollo.

Con el fin de mostrar la validez de los métodos propuestos en esta tesis, estas herramientas han sido integradas o bien en los procesos internos de empresas del sector de Desarrollo Software, o bien en experimentos con usuarios realizados en entornos reales o de laboratorio. De forma adicional, estas herramientas han sido publicadas en plataformas digitales de ámbito internacional tales como Sourceforge, Google Code y GitHub. El objetivo es mostrar al resto de la Comunidad Open-Source los resultados de este trabajo de investigación, así como permitir su uso de forma abierta y altruista.

Contents

List of Figures

List of Tables

1 Introduction

This chapter introduces the background needed to better understand the aim of this PhD thesis.

After giving a short introduction to the research areas motivating the realization of this thesis, the problems tackled in this research work are posed and then analyzed to be overcome. Resulting from this analysis, a set of goals to achieve in this research work are described.

This chapter also provides a brief introduction to the scientific publications and the software contributions resulting from this PhD thesis. The scientific, academic and enterprise context in which this PhD thesis was carried out is described next. Finally, the structure of this document is presented.

1.1 Motivation

1.1.1 Human-Computer Interaction

The focus of this PhD thesis is on the field of Human–computer Interaction (HCI). HCI is often regarded as the intersection of computer science, behavioral sciences, design and several other fields of study. The term HCI was originally introduced by Card, Moran, and Newell in the early eighties. They defined HCI as:

> *a process with the character of a conversational dialog in which the user — someone who wants to accomplish some task with the aid of the computer— provides encoded information (i.e., input) to the computer, which responds back with information and data.*
>
> *Card, Moran, and Newell [36]*

Almost a decade later, in 1992, the Association for Computing Machinery (ACM) provided a more formal definition of HCI in the "Curricula for Human-Computer Interaction". It was defined as:

> *a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.*
>
> *Association for Computing Machinery [3]*

There have been many definitions of HCI made during the last 30 years. However, more recent ones are closer to the approach provided by ACM, as they see HCI more as a discipline, i.e., the process of analyzing user-system interaction, than as the interaction process in itself. For example, the reader can find several definitions of HCI like the following:

> *the study of how people interact with computers and to what extent computers are or are not developed for successful interaction with human beings*
>
> *Tony Manninen [130]*

> *the study of a process in which the human input is the data output by the computer and vice versa*
>
> *Dix et al. [56]*

> *the study of interaction between people (users) and computers*
>
> *Lampathaki et al. [116]*

Based in the definitions above, we can establish HCI in the context of this PhD as the **study** of a conversational process (i.e., the interaction) in which two main subjects participate:

- **the user** (i.e., the human), who motivated by a task to accomplish, provides input data to the computer and expects an appropriate response;
- **the system** (i.e., the computer), which from the input provided by the user, elaborates output data that is provided as response to the user petition.

Additionally, the environment in which these two subjects are interacting to each other can be considered as well as an element of the interaction process:

- **the context**, which refers to those phenomena surrounding the two aforementioned subjects that may affect their behavior and response during the interaction process.

It can be concluded that HCI is a discipline based on the **analysis** of the three aforementioned elements taking part in the interaction process. First, the analysis of the **user** is essential in the design of any system in which one or more user interfaces are involved. However, there exists a need for agreement in the way user-centered design is applied into systems in order to assure the quality and usability of developments [77].

User interaction involves exchanging not only explicit content, but also implicit information about his/her affective state. Affective Computing makes use of such information to enhance interaction, and covers the areas of emotion recognition, interpretation, management, and generation of emotions [33]. Nevertheless, the automatic recognition and management of user emotions is out of the scope of this PhD thesis.

Second, the analysis of the **system** response is crucial as well, as it is directly related not only with the built-in quality of the system, but also with the quality perceived by the final user. Software testing is used to help assuring both the correctness and robustness of a software or service. However, it often represents a tedious and complex task that should be automatically aided to improve its efficiency and effectiveness [131]. Usability and Quality of Experience measures are used to assess, from the users point of view, the quality of a software and the interaction with it.

Third, the analysis of the **context** surrounding the user and the system. It is gaining importance in HCI, specially due to the proliferation of the use of mobile devices. The importance of contextual information has been recognized by research works from many different disciplines, including personalization systems, ubiquitous and mobile computing, etc. [4] Context information is currently being used to enrich applications of a very different nature, like desktop, mobile, and online applications. [20]

1.1.2 Software Testing

HCI can be considered as a particular discipline within Software Testing. Software Testing, in its broader sense, aims at **validating and verifying** that an application or a system works as expected in the given scenario it is intended to work, and that it **meets the requirements** that guided its design and development. The IEEE (Institute of Electrical and Electronics Engineers) and ANSI (American National Standards Institute) standards defined software testing as

> *the process of analyzing a software item to detect the difference between existing and required conditions (i.e., bugs) and to evaluate the features of the software items*
>
> *ANSI/IEEE [92]*

> *the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*
>
> *IEEE [93]*

Testing is **essential in software development**, representing a very important concern before the software can be put in production. However, implementing and integrating software testing processes into a software is **difficult to tackle**. One reason is the vast array of programming languages, operating systems and hardware platforms, as well as their constant evolution [170]. This hinders the integration of testing tools into developments of a different nature (e.g., using different GUI platforms).

Furthermore, testing processes involve tedious tasks **usually expensive in time and human resources**, which sometimes depend on subjective judgments of experts. Automation has tried to mitigate this problem for decades. However, the participation of the human, even using automatic tools, is in most cases still required when creating test cases and validating the results.

Approaches for testing software can be **classified** in different ways depending on the manner the functionality of applications is evaluated. For example, we can talk about **static testing** for processes involving verification tasks like reviews, walkthroughs, or inspections, while **dynamic testing** involves the execution of a given set of code-based test cases within the software.

We can mention also the *Box approach* that divides software testing methods into white-box and black-box testing:

- **White-box testing** is aimed at testing the internal functionality of a software. The tester needs programming skills as well as to know how the system works internally in order to design an appropriate set of test cases. For each test case the tester chooses the inputs and determines the expected outputs. White-box testing is usually done at the software unit level.
- **Black-box testing**, instead, is aimed at testing the functionality of a software without any knowledge of its internal implementation. In most cases low programming knowledge is required for the tester, who has mainly to be aware of what the software is supposed to do. The tester verifies that, for a given input, the software behaves as expected according to the application requirements. Specification-based testing may be necessary to assure correct functionality, but it is insufficient to guard against complex or high-risk situations.
- **Gray-box testing** combines the two approaches mentioned above. While it is required to know the internal working of a software for designing the test cases, the

tester executes the tests at a black-box level.

User interfaces is a field in which testing is specially critical before a software can be accepted by the final user. There are many different types of user interfaces (e.g., haptic, based on speech or gestures, etc.) However, most common ones are the Graphical User Interfaces (GUI). A GUI can constitute as much as 60 percent of the code of an application [153]. Given their importance, testing a GUI for correctness is essential to enhance the safety of the entire system, robustness, and usability [156]. This should lead us to conclude that advanced GUI testing tools are present in most developments today. However, this is not true in most cases.

Testing the behavior of GUIs is less frequent than testing the application core. Testers regularly apply good testing practices such as unit testing and regression testing, but their scope is often limited to the parts without GUI [40]. Moreover, due to the special characteristics of a GUI, techniques developed to test the core functionality of a software cannot be directly applied to GUI testing [158]. Beyond the aforementioned problems, automating GUI testing involves several additional pitfalls such as treating with the differences between the big amount and variety of windowing platforms, selecting the coverage criteria (i.e., select what to test), generating the test cases, providing a method tolerant to changes in the GUI design, or implementing an efficient verification process.

Current methods, tools, and technologies for **GUI testing** can be classified into four different approaches depending on how the test cases are generated.

Full-model-based approaches build a complete model of the GUI. This model, which includes all the interface widgets[1] and their properties, is analyzed in order to explore all the possible paths on an automated test case generation process (e.g., [104, 159, 231]).

Partial-model-based approaches build a smaller model corresponding only to the part of the GUI to be tested, reducing the number of generated test cases. To build and annotate the model, these techniques usually employ modeling languages such as UML (e.g., [164, 215]).

Capture/replay tools generate test cases directly using the information in the target GUI instead of building any model. These tools normally record user interaction events into test scripts (i.e., the test cases) which are replayed later for validation [122]. This

[1] A GUI widget (also called "GUI control" or "GUI component") represents each of the elements compounding a graphical user interface. A widget may provide an interaction point for the direct manipulation of a given kind of data (e.g., a button, a text box), may only display information (e.g., a label), or may be hidden (e.g., a window panel arranging other widgets).

allow developers to implement a lightweight testing process involving only those elements, actions and properties of the GUI that are interesting for the tests [204].

Finally, in **direct test generation** the testers use programming or script languages to directly apply values into the GUI widgets (e.g., by using internal methods of the GUI) and then check the GUI output (e.g., by observation) to verify whether the requirements are met or not (e.g., [40, 230]).

One of the goals of this PhD thesis is to ease the process of defining a set of test cases to check the functionality of a GUI, as well as to ease the process of validating the GUI output. A capture/replay approach is used to simulate the behavior of the tester (i.e., the human) during the testing process. It is proposed as an open framework to facilitate its integration into developments that could be of a different nature.

1.1.3 Data Verification

Another particular discipline within Software Testing are the data verification and validation processes (V&V). Verification of input and output data in user interfaces helps developers build quality into the software throughout its life cycle [95]. It is considered essential to prevent software malfunction, provide robustness, and improve interaction quality in user interfaces.

Runtime Verification (RV) is a special case of V&V in which the execution of a program is monitored to determine if it **satisfies correctness** properties [191]. The data is verified within the runtime context as soon as it is produced instead of using a static representation of it [26]. It means that the requirements are checked at runtime against the actual execution of the program. RV provides a rigorous mean to state complex requirements, typically referring to temporal behaviors [160]. It also provides a rigorous means to implement error detection during development, e.g., by combining it with test case generation [13] or with steering of programs [108].

Data verification is specially **essential in GUIs**. These user interfaces are composed of widgets that hold data. Such data are error prone, and errors may cause unexpected results or even application crashes. However, building advanced verification processes into a GUI in real-time is not straightforward. Applying traditional RV approaches into a project may be **troublesome**, as described now.

RV often involves the integration of **complex procedures** that formally specify the acceptable runtime behavior [46]. This formalization may impose an overhead in some particular scenarios in which, for example, only a set of data constraints have to be verified. Using RV commonly implies the usage of **formal languages** or specific logic. These languages may discourage its usage by developers, and may also present limitations in their expressiveness when writing complete specifications [119]. Building a formal specification may also be troublesome in those stages of the development in which the design or the behavior of the application changes frequently (e.g., early on in the development cycle).

Also, **aspect-oriented programming** (AOP) [106] is commonly used by RV approaches to separate the verification code from the business logic (e.g., [17, 28, 97]). These languages provide an effective mechanism to keep the application code cleaner and enhance encapsulation. However, they **lack a dynamic nature**. Most of AOP languages need to be recompiled each time the rules are slightly changed. Therefore, tasks like fine-tuning the rules dynamically during testing are inconceivable. This is a feature particularly important in industrial developments, in which compilation and deployment stages are specially time consuming. AOP languages also present other disadvantages, like code bloating and maintainability problems further described in [23].

It is also among the goals of this PhD thesis to find a lightweight and dynamic verification approach for GUI data. An approach in which the developers should be able to choose an appropriate verification language according to the project needs and dimensions, but in which key features of RV like high efficiency and encapsulation of verification rules should remain. With this aim, it is proposed an aspect-oriented approach in which the verification process, which uses rules written using scripting languages and that may be changed at runtime, is completely removed from the application main code.

1.1.4 Software Usability

Usability can be seen as a measure of how easy to use an object is, and how easy it is to learn how to use that object. By object we mean anything a human interacts with, like a software application, a website, a handheld device, a machine, a process, etc. In a more formal attempt to describe usability, the ISO norm 9241-11 defined it as

> *“the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use”*
>
> *Nigel Bevan - ISO 9241-11 [24]*

Among the main goals of HCI we can find achieving high usability for computer-based systems. In the Software Engineering field, usability testing is aimed at assuring **software effectiveness and efficiency**, as well as quality of human-computer interaction and **user satisfaction**. For this reason, usability appears as a highly relevant attribute not only in customer-perceived software quality, but also for project managers, for whom it represents a **major decision factor**.

Usability assessment is a **complex process**. Like software testing, it is often expensive in terms of time and human resources. It however presents a higher dependency on experts, mainly because usability evaluation is often based on their **subjective judgments**. Highly specialized personnel is needed to assure usability guidelines and to design an adequate set of usability tests. Sometimes it is also necessary to instruct end-users to obtain significant feedback.

Evaluating usability also entails additional **problems** like the ones described below. A common one is that different representations are used to **describe user-system interaction** in different scenarios. This hinders the analysis and comparison of interaction extracted from different systems (e.g., when evaluating the usability of an application that can run in different handheld platforms). This requires an additional effort to find the correspondence between data extracted from different execution scenarios.

It is also usual to read about methods based on static parameters or average metrics to describe user interaction. The interaction process is dynamic by nature, therefore static approaches restrict the opportunities for the dynamic **analysis of interaction**, e.g., analyzing interaction data to detect why the user presented a uncommon behavior at a specific part of the test. A dynamic approach would better fit approaches based on “live” instrumentation of users, and would support the runtime analysis of human-computer interaction.

Another problem related to usability evaluation appears when systems use two or more modalities to provide or receive data (e.g., current smartphones using GUI, speech and motion modalities). **Multimodal interfaces** try to combine several types of sensory modalities to augment input and output capabilities of current systems such as smartphones,

smart-homes, in-vehicle infotainment, etc. Different modalities are frequently analyzed separately or at different levels of abstraction, e.g., when different tools are used to analyze speech and GUI modalities. As result, data collected using a specific modality is not considered seamlessly with the rest of modalities, thus user-system interaction is not treated as a single and homogeneous flow of actions, as it is happening in reality.

According to the aforementioned problems, it is also a goal of this PhD thesis to propose a generic and dynamic way to represent human-computer interaction with multimodal systems, and thus provide experts with a uniform basis to assess usability of such systems. For this purpose, this work proposes structuring evaluation metrics into a runtime model in order to dynamically describe user-system interaction in multimodal environments.

1.1.5 Quality of Experience

Usability is directly related to **Quality of Experience (QoE)**. The ISO 9241-210 norm hints that "usability criteria can be used to assess aspects of user experience" [55]. Unfortunately, the standard does not clarify the relation between usability and QoE, although it is clear that they are overlapping concepts. While usability includes more pragmatic aspects, QoE focuses more on users feelings stemming both from pragmatic and hedonic aspects of the system. Actually, QoE is a **subjective measure of users experiences** with a service. It focuses on those aspects that users directly perceive as quality parameters and that finally decide the acceptability of a service.

Unlike quality of service (QoS), that refers to technical and objective system performance metrics, QoE rivets on the true feelings of end users from their perspective when they carry out an activity [41, 165, 169, 227]. It encompasses users **behavioral, cognitive, and psychological** states along with **the context** in which the services are provided to them. This is particularly relevant in mobile environments, where applications can be used in different scenarios and social contexts [165].

The surrounding context plays a critical role when analyzing QoE with mobile products [112]. By context information we mean any data used to characterize the situation of an entity (i.e., person, place, or object). Taking such data under consideration for the application process is commonly known as **context-awareness**, and is widely recognized as a core function for the development of modern ubiquitous and mobile systems [21]. Context-aware systems extract, interpret, and use context information to adapt their functionality

to the current context of use [31].

However, evaluating QoE in mobile environments is not straightforward. Several of the problems the reader can find in this area are similar to those present in usability analysis. However, as QoE analysis has to deal with context and subjective data, new problems arise. Working with context data is not straightforward mainly due to the **low standardization** of technologies used in context-aware systems [88]. The design of a QoE evaluation method has to deal with the variety, diversity, and big amount of context data, and decide which parts of this data are useful to describe the surrounding context efficiently. A similar problem appears when selecting useful parameters to capture subjective data from users in order to rate their experiences [165].

Ickin et al. described in [91] another **problem** that represents a challenge for designers and researchers studying user experiences with new mobile applications. There are not robust methodologies that **combine** quantitative methods to evaluate QoS and performance with qualitative methods for usability evaluation. Therefore, how to integrate a qualitative method to evaluate QoE into an interaction analysis method is still a problem to solve.

It is also a goal of this PhD thesis to provide unified criteria to analyze and compare QoE in mobile environments. For this purpose, it specially focuses on providing a common description of the surrounding context. A set of data to describe subjective ratings for user experiences is provided as well. These descriptions are then incorporated into an interaction analysis method to allow the analysis of QoE in dynamic environments.

1.2 Enhancing Software Quality

Current state of the art in Software Testing and HCI fields shows that the quality of a software can be achieved in different ways. This PhD thesis focuses mainly on **user interfaces**, which represent the software component where interaction between humans and machines occurs. This research work explores the field of improving software quality through the **analysis of the interaction** between the user and the system in real-time.

This thesis is based on the fact that user-system interaction can be analyzed from two different perspectives:

(a) Interaction can be **decomposed into its main elements** (i.e., input, output, and context). Thus, the quality of the different components of interaction can be analyzed

and improved separately, with the aim of enhancing the final quality of the entire software.

(b) Interaction can also be seen **as a whole**, as a continuous flow of actions from the user to the system and vice versa. Therefore, the interaction between the user and the system, as well as its context, can be analyzed and evaluated as a whole in order to enhance the quality of a software.

Section 1.1 described different pitfalls and problems to which developers and field experts have to face when trying to enhance the quality of a software. This section identifies some relevant issues and describes how this PhD thesis tackles them. Such issues are divided into two blocks according to the **two approaches** described above.

1.2.1 Block 1: Achieving Quality in Interaction Components Separately

As described above, *Perspective (a)* stands for analyzing the different components of user-system interaction separately to assure their quality and, as a consequence, improve the quality of the whole software. This approach is aimed at **focusing analysis and testing processes on a particular aspect of the interaction** (i.e., user input, or software output), and thus increasing the effectiveness of the applied methods.

Section 2.1 analyzes state-of-the-art approaches related to this perspective. From this analysis we can identify several **deficiencies**, **pitfalls**, **and problems** that should be considered in order to provide effective testing and validation mechanisms in the future. These are identified in Table 1.1.

Research in the Block 1 of this PhD thesis attempts to ease the development of **software testing and data verification tools** trying to overcome the issues indicated in Table 1.1. The scope of this block is limited to those tools dedicated specifically to graphical user interfaces (GUI).

This part of this research work focuses on providing **adaptable and open frameworks** for developing tools aimed at **automating** testing and validation processes efficiently. By adaptable we mean to be suitable to different execution scenarios, i.e., to be agnostic to elements like the operating system or the GUI platform in use. By open we mean to be easily extensible with new functionality.

Problem description
Diversity in the development environment: operating systems, programming languages, GUI platforms, etc.
Integration of testing tools into new and ongoing developments with different and varied features.
Testing tasks are often expensive in terms of time and human resources. They present also a high dependency on the manual intervention of experts.
Implementation of efficient testing processes that, even if performed at runtime, do not interfere with the natural usage of the application.
Graphical user interfaces have special features that need dedicated testing tools.
Tolerance to changes in the implementation. These changes are specially frequent in the development of user interfaces.

Table 1.1 Main issues found in current state-of-the-art approaches analyzed in Block 1.

At the same time, the **incorporation** of such tools into ongoing and future developments should be straightforward. Code intrusiveness should be minimal as well to reduce the impact of integrating these tools, as well as to widen the range of potential applications to be tested.

In an attempt to overcome these issues, the research activities conducted in this part of the PhD thesis resulted in the design of two different frameworks, OHT and S-DAVER, to analyze the validity of system response, and user input data, respectively:

- **OHT (Open HMI Tester)**. Presented in Chapter 3, it describes a framework to validate system response. User interaction is recorded to generate test-cases, which are executed later in an automatic process (by simulating the user action) to ease the evaluation of the results provided by the system.
- **S-DAVER (Script-based Data Verification framework)**. Presented in Chapter 4, it describes a verification framework aimed at guaranteeing the validity of the user input. It uses script languages to define verification rules in separate files. Rules are executed transparently at runtime.

The separate analysis of interaction components allow developers and testers to focus their efforts when validating user input or system output. They provide a rigorous means to check the validity of data provided by users as well as to assure that the system response conforms to the expected behavior. However, by separating interaction components we

omit some features of user-system interaction that might be helpful to widen the scope of our analysis, e.g., to assess the quality of software from a **user-centered perspective** (see Table 1.2).

If the analysis process is focused only on one component of interaction, the method itself has only a **partial view** of the interaction process, thus loosing its totality. In this way, the connections between the participants, as well as the connections with the means for interaction and the environment, remain out of analysis.

Analyzing interaction components separately also lacks the **conversational nature** of user-system interaction described by Card et al. in [36]. It implies that those cause-effect relationships triggered by an action of the system or the user, which are very helpful for the analysis of the interaction process, are ignored.

Finally, using the approaches included in Block 1 to analyze user-system interaction in environments in which **several modalities** are used (e.g., screen touch, speech, gestures, motion, etc.) might not be the best choice. Analyzing data provided with different modalities, which may be even used concurrently, would not be straightforward.

Problem description
Have a partial view of the interaction process instead of basing the analysis on information about all the stakeholders.
Ignore the conversational nature (i.e., the cause-effect relationships between system and user actions) of user-system interaction.
Implement analysis of user interfaces using several modalities.

Table 1.2 Main issues found in the approaches proposed in Block 1, which motivate further research in Block 2.

1.2.2 Block 2: Achieving Quality of User-System Interaction as a Whole

As mentioned above, *Perspective (b)* stands for **analyzing the whole interaction** between the user and the system instead of separating its components. This approach is aimed at **keeping the totality** of the interaction process during the analysis and evaluation stages. Working in this new perspective was also motivated by the limitations of the first results of this PhD thesis (see Table 1.2) which led us to research new methods to analyze interaction as a whole, from a conversational approach and supporting different sensory modalities.

Section 2.2 analyzes current research approaches under this perspective, which are aimed at describing the interaction process for the development or the assessment of user interfaces. As in Block 1, this analysis helped us to identify several **issues to be addressed** in order to provide a proper mechanism to support the analysis and evaluation of the interaction process. These issues are identified in Table 1.3.

Problem description
Variety, diversity, and big amount of interaction and context data. Decide what data is useful to describe the interaction process efficiently.
Low standardization of methodologies used to describe the interaction process and its context. Comparison of interaction extracted from different systems.
The use of static approaches hinders the dynamic analysis of interaction.
Different modalities are often analyzed separately or at different levels of abstraction. Multimodal interaction is not represented as a single and homogeneous flow of actions.
Need of robust methodologies combining quantitative methods to evaluate performance with qualitative methods for usability evaluation.
Finding parameters to capture subjective data from users with the aim of rating their interaction experiences.

Table 1.3 Main issues found in current state-of-the-art approaches analyzed in Block 2.

The approaches proposed in Block 2 of this PhD thesis aim at providing a representation of interaction trying to overcome the problems described in Table 1.3. While approaches in Block 1 focus on a particular interaction component, these approaches focus on **describing the whole interaction process** between the user and the system in a specific context. Interaction is described **stepwise**, as a dialog between two parties to enable further action-reaction analysis. The analysis of the users and their characteristics gains in importance in this second approach.

First, our efforts are mainly focused on **modeling the user and the system actions** as a conversation (i.e., as a dialog between these two parties). The main goal is providing a **uniform representation** of human-computer interaction that enables the analysis and comparison of interaction in different execution scenarios. The first result in this area is the design of PALADIN:

- **PALADIN (Practice-oriented Analysis and Description of Multimodal Interaction).** Presented in Chapter 5, it describes a metamodel aimed at modeling the dialog

between the system and the user in multimodal environments. It uses a stepwise description of the interaction (i.e., by keeping its order in time) to preserve the dynamic nature of a conversation. It also supports different modalities for providing input and output data.

Nowadays **mobile interaction** is gaining in importance due to the proliferation of the use of mobile devices. Users and their handheld devices are continuously moving in several simultaneous fuzzy contexts [88]. Therefore, the **surrounding context** represents an element that has to be incorporated into those methods aimed at supporting the evaluation of the **mobile usability** of a system.

Moreover, while quantitative data is used to determine whether and when usability specifications are met, **qualitative data** is used to collect users impressions to identify usability problems, their causes, and potential redesign solutions [80]. Interaction analysis methods should also include qualitative data extracted from users in order to further evaluate their experiences.

PALADIN mainly focuses on quantitative data to describe the interaction process. Therefore, at this point our research is aimed at incorporating the **context factors** into the analysis of user-system interaction, as posed in the ACM definition of HCI [3]. Furthermore, qualitative and subjective data have to be incorporated into such a method to analyze the quality perceived by users.

The main goal of this part of the PhD thesis is to analyze the **user experiences** when using a software, as well as those phenomena surrounding the interaction process that may influence not only the quality and usability of a software, but also the quality of the experiences of the users with a software. This research resulted in the design of CARIM as an extension to PALADIN:

- **CARIM (Context-Aware and Ratings Interaction Metamodel)**. Based on the design of PALADIN, CARIM is a metamodel aimed at describing the interaction between the user and the system in mobile environments as well as its context (e.g., physical environment, social context, etc.). It also includes users ratings and other factors used to describe their experiences. CARIM is presented in Chapter 6.

1.3 Goals of this PhD Thesis

Once introduced the research areas that motivate the realization of this PhD thesis, and once identified some of the major problems with which developers, testers, and field experts have to deal when trying to enhance the quality of software and interaction, this section defines a set of goals to achieve in this research work.

We define the main goal of this PhD thesis as the improvement of current techniques for software testing and user-system interaction analysis. Then, as a result of this work, we intend to provide the HCI and Software Testing Community with more effective, robust, adaptable, lightweight, and easy-to-integrate tools for improving the quality of software products, as well as the experiences of users with the software.

Nevertheless, this goal is too abstract and ambitious, and thus a set of more specific goals organized according to the two research blocks mentioned above in Section 1.2 are described in the following.

The **goals to be achieved regarding research in Block 1** in order to provide a mean to **assure quality of system response and user input** are:

- $\mathbb{G}$**1.1** Provide a framework to support the development of tools aimed at **validating the response of a GUI**.
  - Allow the **automation** of those complex and time-consuming tasks that comprise **GUI testing processes** and which are often performed manually.
  - Allow the simulation of the actions of a human tester with the aim of **testing software** functionality in a **real**, **reliable**, **and robust** scenario.
  - Allow the testing tools developed within this framework to be **integrated into different execution scenarios** (i.e., different operating systems or windowing platforms).
  - Allow the testing tools developed within this framework to be **easily integrated into existing**, **ongoing**, **and new developments**.
- $\mathbb{G}$**1.2** Provide a framework to integrate **data verification processes** into GUI developments to guarantee that the **user input** conforms to the data requirements.
  - Allow developers to **choose an appropriate verification language** according to the project needs, its resources, and its dimensions.

  - Provide a **lightweight and easy-to-integrate** verification process to be incorporated into developments of different sizes.
  - Provide a **dynamic verification process** to support and help developers, testers, and users during the whole life-cycle of the software.

The **goals to be achieved regarding research in Block 2** with the aim of providing a mean to **assure quality of user-system interaction** are:

- $\mathbb{G}$**2.1** Provide a framework to support the **instrumentation and assessment of multimodal user-system interaction**.
  - Provide a **generic description of interaction** regardless of the execution scenario (i.e., device, platform, operating system, application, modalities used to provide input/output data, etc.)
  - Capture the **dynamic nature** of the interaction process to support its analysis in a stepwise manner.
  - Allow the **comparison between different interaction records** regardless of the execution scenario in which they were previously obtained.
  - Use data that can be **collected by using current devices capabilities** to ease its integration into real scenarios.
- $\mathbb{G}$**2.2** Provide a framework to support the **analysis and evaluation of the whole user experience in mobile scenarios**.
  - Provide a generic and reusable **definition of the context of interaction** suitable for different mobile and non-mobile scenarios. The evolution over time of the contextual factors should be also considered.
  - Provide a set of **metrics to capture the users experiences** when using a system, regardless of the interaction scenario.
  - Allow the assessment of systems usability and QoE based on the **combination of quantitative interaction/context data and qualitative/subjective data** extracted from users.

1.4 Publications Related to this PhD Thesis

The research conducted during this PhD thesis has been published—or is under revision—on several peer-reviewed, national and international journals and conferences. This section summarizes all the scientific publications directly related to this work. Most of these publications have been the result of the work under the Cátedra SAES-UMU[2] to attend SAES[3] needs, and also result of the work in other parallel lines of research. Moreover, other publications resulted from our fruitful collaboration with the Telekom Innovation Laboratories[4] (Berlin, Germany).

JCR Journals

$\mathbb{P}$**1** [147] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz. **OpenHMI-Tester: An Open and Cross-Platform Architecture for GUI Testing and Certification**, *International Journal of Computer Systems Science and Engineering (IJCSSE), Special Issue on Open Source Certification*, June 2010.
https://www.researchgate.net/publication/256296748

$\mathbb{P}$**2** [142] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz. **A Context-aware Interaction Model for the Analysis of Users QoE in Mobile Environments**, *International Journal of Human-Computer Interaction (IJHC), Taylor & Francis, in press*, May 2014.

JCR Journals Under Review

$\mathbb{P}$**3** Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **A Lightweight Framework for Dynamic GUI Data Verification Based on Scripts**, *Submitted for publication*, May 2014.

[2]The Cátedra SAES-UMU initiative was signed by SAES and the University of Murcia (http://www.um.es) to work on research leaning toward the improvement of industrial and military developments. http://www.catedrasaes.org

[3]SAES (Sociedad Anónima de Electrónica Submarina) is the only Spanish company specialized in submarine electronics and acoustics. http://www.electronica-submarina.com

[4]Telekom Innovation Laboratories (T-Labs) is the the central research and development institute of Deutsche Telekom. http://www.laboratories.telekom.com/

ℙ**4** Pedro Luis Mateo Navarro, Stefan Hillmann, Sebastian Möller, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **Run-time Model Based Framework for Automatic Evaluation of Multimodal Interfaces**, *Submitted for publication*, April 2014.

Other Journals

ℙ**5** [143] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz. **Aplicación de Open HMI Tester como framework open-source para herramientas de pruebas de software** (Using Open HMI Tester as an Open-source Framework for the Development of Testing Tools), *Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS), volume 5, number 4*, December 2009.
http://www.ati.es/IMG/pdf/MateoVol5Num4-2.pdf

ℙ**6** [146] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **A Proposal for Automatic Testing of GUIs based on Annotated Use Cases**, *Advances in Software Engineering journal, Special Issue on Software Test Automation, vol. 2010, Article ID 671284, 8 pages*, 2010.
http://downloads.hindawi.com/journals/ase/2010/671284.pdf

International Conferences and Workshops

ℙ**7** [144] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **Automated GUI Testing Validation guided by Annotated Use Cases**, *MOTES 09 - Model-based Testing Workshop, Lübeck, Germany*, September 2009.
http://subs.emis.de/LNI/Proceedings/Proceedings154/gi-proc-154-252.pdf

ℙ**8** [141] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **Towards Software Quality and User Satisfaction through User Interfaces**, *Fourth IEEE International Conference on Software Testing, 2011. Berlin, Germany*, March 2011
http://origin-www.computer.org/csdl/proceedings/icst/2011/4342/00/4342a415.pdf

ℙ**9** [137] Pedro Mateo, Stefan Schmidt. **Model-based Measurement of Human-Computer Interaction in Mobile Multimodal Environments**, *aMMI (Assessing Multimodal Interaction), 7th Nordic Conference on Human-Computer Interaction (NordiCHI). Copenhagen,*

Denmark, October 2012.
http://www.prometei.de/fileadmin/ammi-nordichi2012/04ammi12_mateo_hillmann.pdf

ℙ**10** [149] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **A Context-aware Model for the Analysis of User Interaction and QoE in Mobile Environments**, *CENTRIC 2012, The Fifth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services. Lisbon, Portugal*, November 2012.
http://www.thinkmind.org/download.php?articleid=centric_2012_5_20_30061

ℙ**11** [150] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, Gregorio Martínez Pérez. **A Context-aware Model for QoE Analysis in Mobile Environments**, *XIV International Congress on Human-Computer Interaction. Madrid, Spain*, September 2013.
https://www.researchgate.net/publication/256296818

National Conferences and Workshops

ℙ**12** [145] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz. **Open HMI Tester: un Framework Open-source para Herramientas de Pruebas de Software** (Open HMI Tester: an Open-source Framework for Software Testing), *PRIS 2009 - IV Taller sobre Pruebas en Ingenieria del Software, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2009), San Sebastian, Spain*, September 2009.
https://www.researchgate.net/publication/262357212

ℙ**13** [148] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, Diego Sevilla Ruiz. **Verificación de Datos en la GUI como un Aspecto Separado de las Aplicaciones** (Verifying GUI Data as a Separate Aspect), *PRIS 2010 - V Taller sobre Pruebas en Ingenieria del Software, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2010), Valencia, Spain*, September 2010.
https://www.researchgate.net/publication/256296667

1.5 Software Contributions of this PhD Thesis

The research activities conducted in this PhD work, aimed at achieving the goals described above in Section 1.3, resulted in a set of **software resources** as proof of concept for the proposed designs. These results are summarized in Figure 1.1 and further described below. This section briefly describes those contributions that represent the **backbone of this PhD thesis** (i.e., *Open HMI Tester (OHT)*, *S-DAVER*, *PALADIN* and *CARIM*) which are further described along this document. There are also other contributions developed as proof of concept of parallel research works. These ones are described in the corresponding chapter along this PhD thesis.

As depicted in Figure 1.1, the main software contributions of this work are classified according to the interaction component they address (either system output, user input, or the whole interaction process), to the testing discipline used (i.e., software testing and validation, or usability and QoE evaluation), and whether the method is mainly oriented to analyze/evaluate the system, or the user.

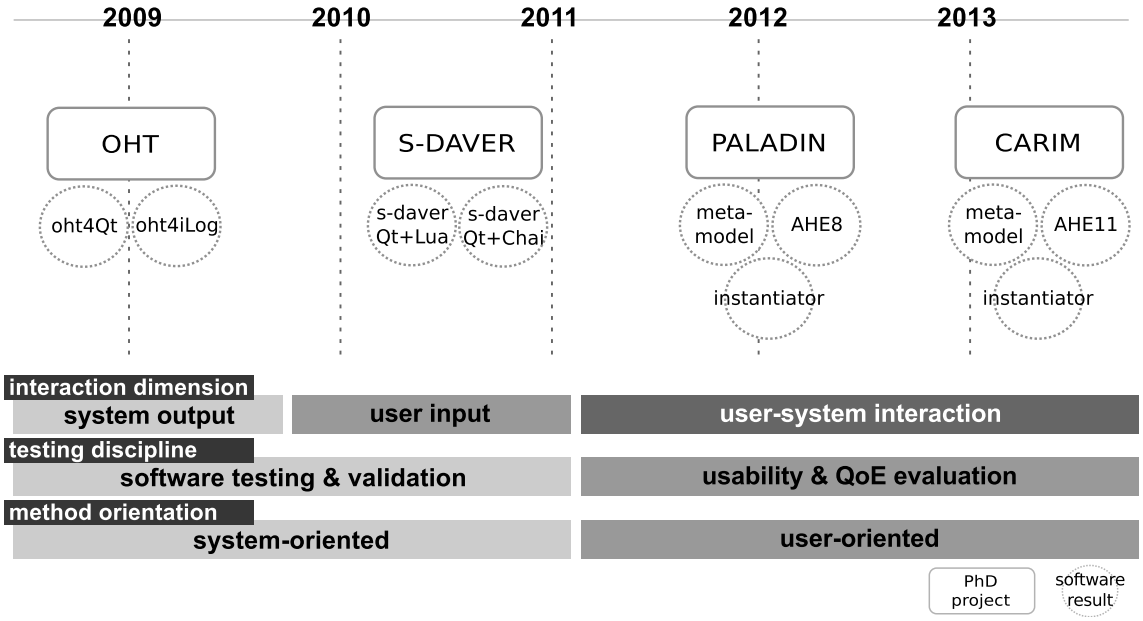


Figure 1.1 Main software contributions roadmap of this PhD thesis.

As said above, these software contributions were used to **show the validity** of the designs and solutions proposed along this PhD thesis. Different software prototypes or final

applications were designed, developed, and integrated into real scenarios as proof of concept. Those contributions oriented to assure software quality (i.e., *OHT*, and *S-DAVER*) were integrated into the internal processes of a software development company (see Section 1.7) and extensively used to test and validate their software components. On the other hand, those contributions oriented to analyze user interaction (i.e., *PALADIN* and *CARIM*) were tested with users in experiments conducted both in laboratory and in real environments.

Furthermore, in order to enrich the validation process, all methods and tools developed during this PhD thesis were provided as a **contribution to the open-source community** by using popular means of distribution like Sourceforge[5], Google Code[6] and GitHub[7]. This way, community members can use our results and then provide us with valuable feedback to enhance our current and future work.

1.5.1 OHT: Open HMI Tester

(𝕊**1**) *Open HMI Tester* [133, 151] is a human-machine interface (HMI) testing architecture. Its design is aimed at supporting major event-based windowing systems, thus providing generality besides some other features such as scalability and tolerance to modifications in the HMI design process. This GUI testing tool follows a capture/replay approach, which represents one of the most used techniques in current windowing systems. *OHT* performs a non-intrusive hooking into the target application in order to capture generated events from the application and post new events to it.

The *OHT* implementation was released as an open-source contribution. The source code is distributed into a packaged file that can be built for different operating systems. Implemented in standard C++, the *OHT* architecture allows adapting the implementation of a few modules to fit both the windowing system and the operating system in use. A final version of the *OHT* to test Qt4-based [174] applications in Linux systems was also implemented.

Info and download: http://www.catedrasaes.org/wiki/OHT

[5] http://sourceforge.net/
[6] http://code.google.com/
[7] https://github.com/

1.5.2 S-DAVER: Script-based Data Verification

(𝕊2) *S-DAVER* [140, 152] is a framework for implementing separate input/output data verification within user interfaces. Following an aspect-oriented approach, it fully decouples the verification logic (i.e., rule specification and verification code) from the business logic. Requirements are translated into rules written using interpreted languages, and are included in separate files. Requirements can change dynamically at runtime without recompilation, enabling live fine-tuning of the rules. Visual feedback is used to assist developers during testing as well as to improve users experience during execution. *S-DAVER* can be adapted to fit different verification languages and windowing platforms.

The implementation of *S-DAVER* has been released as an open-source contribution to the community. The source code is distributed into a packaged file that can be built for different operating systems. Implemented in standard C++, the framework can be modified by developers that need to adapt it to their developments. *S-DAVER* is also distributed as RPM and DEB packages to ease its installation within different Linux distributions. Two final versions of *S-DAVER* for Qt4-based Linux applications were developed using two different verification languages: one using Lua [94] and other using Chaiscript [100].

Info and download: http://www.catedrasaes.org/wiki/GuiVerification

1.5.3 PALADIN: Practice-oriented Analysis and Description of Multimodal Interaction

(𝕊3) *PALADIN* [138, 139] is a metamodel aimed at describing multimodal interaction in a uniform and dynamic way. Its design arranges a set of parameters to quantify multimodal interaction generalizing the existing differences between modalities. Moreover, interaction is described stepwise, preserving the dynamic nature of the dialog process. As result, *PALADIN* provides a common notation to describe user-system interaction in different multimodal scenarios, which can be used to assess and compare the quality of interaction in different systems.

The implementation of *PALADIN* is based on the Eclipse Modeling Framework (EMF) [207], which provides a metamodeling methodology, automatic code generation, syntactical validation, and model transformation functionalities. The EMF description of the metamodel was used to automatically generate its source-code in Java. PALADIN was released along with the *Android HCI Extractor*, a tool to track user-system interaction in Android [105],

and an instantiation framework. *PALADIN* as well as all the tools around it are open-source, so they can be downloaded and modified without restrictions.

Info and download: http://www.catedrasaes.org/wiki/MIM

1.5.4 CARIM: Context-Aware and Ratings Interaction Metamodel

($\mathbb{S}$**4**) *CARIM* [135, 136] is a metamodel aimed at modeling the users quality of experience (QoE) in mobile environments. It establishes a set of parameters to dynamically describe the interaction between the user and the system as well as the context in which it is performed. Parameters describing the perceived quality of users, their attitude toward technology use, and their mood are included as well. *CARIM* provides a uniform representation of interaction in mobile scenarios, allowing the analysis of interaction to determine QoE of users as well as the comparison between different interaction records. Its runtime nature brings new possibilities for the dynamic analysis of interaction, enabling also to make decisions in real-time to adapt the system to provide a better experience to users.

The *CARIM* metamodel was also implemented with EMF. This design was later used to automatically generate the Java source code to be integrated into those applications in which new *CARIM* instances are created. To collect interaction and context parameters we developed the *AHE11*, an improved version of the *Android HCI Extractor*. We also developed the *Generic Quest Library (GQL8)* for Android to automatically collect data from users by using questionnaires. *CARIM*, as well as all the tools around it, are open-source, so they can be downloaded and modified without restrictions.

Info and download: http://www.catedrasaes.org/wiki/Carim

1.6 Summary of Research Goals, Publications, and Software Contributions

Table 1.4 summarizes the research goals, publications, and open-source software contributions described in the sections above. The table is structured based on the two research blocks and the four main research fields in which this PhD thesis is divided.

Research Block	Main research field	Goals achieved	Publications	Software contributions
Block 1	Software testing	$\mathbb{G}$**1.1**	$\mathbb{P}$**1** [147] $\mathbb{P}$**5** [143] $\mathbb{P}$**6** [146] $\mathbb{P}$**7** [144] $\mathbb{P}$**12** [145]	$\mathbb{S}$**1** [133, 151]
	Data verification	$\mathbb{G}$**1.2**	$\mathbb{P}$**13** [148] $\mathbb{P}$**3**	$\mathbb{S}$**2** [140, 152]
Block 2	Usability evaluation	$\mathbb{G}$**2.1**	$\mathbb{P}$**9** [137] $\mathbb{P}$**4**	$\mathbb{S}$**3** [138, 139]
	Mobile QoE evaluation	$\mathbb{G}$**2.2**	$\mathbb{P}$**2** [142] $\mathbb{P}$**10** [149] $\mathbb{P}$**11** [150]	$\mathbb{S}$**4** [135, 136]

Table 1.4 Summary of research goals, publications, and open-source software contributions in this PhD thesis.

1.7 Context of this PhD Thesis

This PhD Thesis has been carried out under the umbrella of the Cátedra SAES-UMU, a private initiative created by the company SAES (Sociedad Anónima de Electrónica Submarina) and the **University of Murcia**[8] and aimed at improving the development processes of the company, paying special attention to increasing the final quality of their software products. This research work has also been accomplished thanks to a six-month stay in the Telekom Innovation Laboratories in Berlin, Germany.

The **Cátedra SAES-UMU**[9] is also interested on developing and using open-source tools to validate the developments of SAES, as well as to provide the open-source community with most of its research results. This commitment with free and open-source tools clearly marked the character of the software solutions developed within this PhD thesis. All methods and tools resulting from our research were provided as an altruistic contribution to the **open-source community**.

SAES[10] is the only Spanish company specialized in submarine electronics and acoustics. It

[8] http://www.um.es
[9] http://www.catedrasaes.org
[10] http://www.electronica-submarina.com

is worldwide leader in the ASW segment, and is currently involved in the new deployment of programs of submarines and surface vessels, as well as in the modernization of MPA for Spanish Navy and foreign navies. The needs and problems proposed by a software development company like SAES motivated this research work to find new and realistic methods aimed at improving the efficiency and effectiveness of software testing. Working near such a company provides us with real constraints —specially technological ones— when researching, designing, and developing new solutions.

The **Telekom Innovation Laboratories**[11] (also called T-Labs) was established in 2004 as the central research and development institute of Deutsche Telekom. The aim of the research activities conducted in T-Labs is to develop innovative products and solutions while working in close cooperation with science and industry, as well as promoting the integration of the resulting innovations in the Deutsche Telekom Group. The user-centered approach of this research company, mainly focused on enhancing the usability of systems by analyzing the user-system interaction process, clearly influenced the research activities conducted during this PhD thesis.

1.8 Structure of this PhD Thesis

This PhD thesis is structured into seven main chapters corresponding to an introductory part, the description of the current state of the art, the four main contributions that represent the backbone of this research work and that were briefly introduced in Section 1.5, and a final chapter including the conclusions and the description of the main lines of future work.

Chapter 1 describes the context of this PhD thesis and some background information for the reader to understand the main problems tackled in this research work. All the contributions resulted from this PhD thesis, as well as the scientific publications directly related to this dissertation, are also described in this chapter.

Chapter 2 introduces the background needed for the better understanding of this PhD thesis. This chapter describes some of the most relevant approaches related to the research activity conducted in this dissertation. These approaches present some deficiencies and open research questions that are addressed in the following chapters.

Chapter 3 studies the current problems of Software Testing when trying to assure the quality, reliability, and robustness of a software component in an specific scenario of use.

[11] http://www.laboratories.telekom.com

This chapter emphasizes the problem of graphical user interfaces (GUI) where testing is critical before the software can be accepted by the final user and put in execution mode. The design and development of an open testing architecture called *Open HMI Tester* is described as well.

Chapter 4 addresses the problem of data verification, specially input data introduced by the users in GUIs. The pitfalls related to the integration of advanced verification processes into different scenarios are discussed, and some solutions aimed at reducing the complexity of this process are proposed. This chapter also describes the design and implementation of *S-DAVER*, a framework to ease the implementation of data verification in user interfaces.

Chapter 5 tackles the problem of describing the interaction between the user and the system in such a way that provides the experts with a basis for the implementation of usability evaluation processes. The most relevant problems of describing multimodal interaction preserving its dynamic nature and considering modalities at the same level are tackled as well. This chapter describes the design of *PALADIN*, a parameter-based metamodel aimed at describing multimodal interaction in a uniform and dynamic way.

Chapter 6 brings the problem described in Chapter 5 beyond *flat* environments. It presents a new approach that considers the changes of the surrounding context, as well as the user impressions during the interaction process, with the aim of evaluating users quality of experience (QoE) in mobile environments. This chapter presents the design of *CARIM*, a metamodel that provides unified criteria for the analysis of interaction and the inference of QoE in mobile scenarios.

Finally, **Chapter 7** includes the conclusions of this PhD thesis, and describes the lines of future research looming on the horizon.

2

Related Work

This chapter introduces the background that contextualizes the work in this PhD dissertation. It includes a summary of some of the most relevant approaches related to the research activity conducted in this thesis. The readers may benefit from familiarizing themselves with the described approaches and their contribution to the software development community and the HCI community.

Related works are structured into two main groups according to the two research blocks in which this PhD thesis is divided (see Section 1.2). Approaches in Group 1 (Section 2.1) are intended to assure the quality of interaction components individually. Approaches in Group 2 (Section 2.2) are intended to describe and analyze the interaction process as a whole, including also the environment and subjective information of users.

The approaches described in this chapter present some deficiencies and unaddressed questions that should be tackled. Several of these problems and open research questions are identified, described, and addressed in the following chapters of this PhD dissertation.

2.1 Group 1: Approaches Assuring Quality of a Particular Interaction Component

The approaches described in this group are intended to assure the quality of a particular component of the interaction process individually. Subsection 2.1.1 describes methods, tools and technologies implementing GUI testing processes aimed at validating software output. Subsection 2.1.2 describes several approaches aimed at checking the validity of the input data provided by software users.

2.1.1 Validation of Software Output

Testing is commonly used to assure the quality, reliability, and robustness of GUI-based systems. Different test cases are generated and then executed to validate the system output in the given context the application is intended to work. The GUI testing approaches analyzed in this section can be classified in three types, depending on how the test cases are generated:

- **Approach 1**. (Subsection 2.1.1.1) Tools that build a complete GUI Model including all the graphical components (GUI widgets) and their properties. This model is traversed in an automated test case generation process to explore all the possible interaction paths and generate test cases to test them.
- **Approach 2**. (Subsection 2.1.1.2) Tools that build a smaller model corresponding to the part of the GUI to be tested. As a result, the number of generated test cases is reduced. These tools often use modeling languages such as UML [177] to define and annotate the model in order to guide the test case generation process.
- **Approach 3**. (Subsection 2.1.1.3) Methods in which test cases are created by the testers by using script languages, or by recording the user-system interaction data into test scripts. The test cases are replayed later to check the validity of software output, e.g., by implementing observation-based testing [122]. These techniques allow developers to perform a lightweight testing process taking into account only the elements, actions, and properties of the GUI to be tested [204].

2.1.1.1 Methods Using a Complete Model of the GUI

Memon et al. described in [155] a process named *GUI Ripping*. In this process, all the windows of the GUI are traversed to analyze all of its widgets and events. Then, a model is automatically built to describe a *GUI Forest* (i.e., a tree composed of all the GUI widgets) and an *Event-Flow* Graph (EFG) (i.e., a graph describing all the GUI events). The model is verified, fixed and completed manually by the developers.

Memon et al. described in [157, 228] the memon03dart framework, which is based on the process described above. memon03dart uses the GUI model to explore all the possible test cases, from which the developers select those ones identified as meaningful. Then, an oracle generator is used to create the expected output. A *test oracle* [229] 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, the test cases are automatically executed and their output compared to the results expected by the oracle.

White et al. described in [223, 224] a similar technique using a GUI model based on reduced FSM (finite-state machine). The authors introduced the concept of *responsibility* to denote a desired or expected activity in the GUI, and defined a *Complete Interaction Sequence* (CIS) as a sequence of GUI objects and actions that will raise an identified responsibility. The model is used to automatically generate the test cases, which are then executed to find "defects" (i.e., serious departures from the specified behavior) and "surprises" (i.e., user-recognized departures from the expected behavior, but not explicitly indicated in the specifications of the GUI).

The tools belonging to this approach focus part of their efforts on building a model of the GUI from which the test cases are generated automatically. Creating and maintaining these models is a very expensive process [154], and specially dangerous if the models are completed and validated manually. A GUI is often composed of a complex hierarchy of widgets and properties, many of which are irrelevant for the developers (e.g., window containers or base panels). Therefore, the result of the auto-generation process might include a vast amount of useless/senseless test cases.

This leads to other problems such as scalability or low tolerance to modifications. For example, adding a new element to the GUI (e.g. a new widget or event) would have two worrying side effects. First, the number of generated test cases grow exponentially at the same time new elements are added. Second, updating the GUI model would involve the

manual verification and completion of the change in the model, as well as the regeneration of all affected test cases. As a result, even a minimal change in the GUI appearance might cause a lot of errors during the validation process if the test cases or the oracle outputs remain obsolete [153].

2.1.1.2 Methods Using a Partial Model of the GUI

Vieira et al. described in [215] a method in which UML use cases and activity diagrams are used to respectively describe which functionality should be tested and how to test it. The UML model is enriched by improving the accuracy of the activity diagrams (i.e., decreasing the level of abstraction) and by making annotations using custom UML stereotypes to represent additional test requirements. Finally, the test cases are automatically generated from the enriched UML model.

The scalability of this approach is better than the offered by the methods described above, as the process is now focused only on a section of the model. Even though, it might result in a very large number of test cases. The refinement of the diagrams helps also to reduce the final number of generated test cases. However, as happens with the tools belonging to the first approach, the main limitations of this approach are again that the developers have to spend so much effort building, refining and annotating the model and the low tolerance to modifications.

2.1.1.3 Methods Based on GUI Interaction

Steven et al. presented [204], a tool for capturing and replaying Java program executions. This tool uses an unobtrusive capture process to record the interactions (i.e., 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 recording stage. steven00jrapture interacts with the underlying operating system or windowing platform to build a *System Interaction Sequence* (SIS). A SIS includes the sequence of inputs to the program together with other information necessary for the future replay.

Caswell et al. described jfcUnit [38], a framework to create automated tests for Java Swing GUIs based on the JUnit testing framework [90]. It provides automatic capture of test cases, which are recorded into XML-based script files. The scripts can be edited or created manually as well. Then, the test cases are automatically executed into a GUI. jfcUnit

provides a mean to locate Swing objects on the screen and to interact with them by using a set of functions.

Chang et al. presented in [40] a new approach for GUI testing based on computer vision. The test cases are generated manually by the testers, who include action statements in a test script to interact with the GUI and to verify whether the desired response is obtained. The authors use Sikuli Script [230], that provides a library of functions to automate user inputs such as mouse clicks and keystrokes. These functions take screenshots of the GUI widgets as arguments. Then, during script execution, it is automatically checked if the visual outcome matches the expected one. This approach improves the readability of test cases and provides platform independence. However, it is unable to detect unexpected visual feedback and to test internal functionality of the GUI.

El Ariss et al. presented in [61] a method for Java GUIs combining a model based testing approach with a capture and replay approach. First, the testers model the behavior of the application using its functional specifications. Then, they use capture and replay functionality to automatically record and execute the test cases. For this they use jfcUnit. Finally, the model is used to improve the coverage criteria of the capture/replay process, and therefore ensure that all the GUI widgets are exercised fully.

The tools belonging to this approach present a better scalability because the test case generation process is guided by the tester. Furthermore, the developers do not need to build or maintain any model of the target application. However, using a human-guided method may result in problems with the testing coverage, as some GUI functionality might remain out of the testing process. The maintenance of test scripts may be expensive if the GUI design changes frequently.

2.1.2 Validation of User Input

The validation of user input is essential to provide integrity and robustness into an application during execution. Runtime Verification (RV) approaches are characterized by monitoring the current execution of a program to determine if it satisfies correctness properties. They verify the application behavior or the input data within the runtime context. This section analyzes the most relevant works related to RV and the verification of user input data, organizing them into the three following approaches:

- **Approach 1**. (Subsection 2.1.2.1) Approaches using formal logic for the verification process. These logic are often created for a specific verification purpose.
- **Approach 2**. (Subsection 2.1.2.2) Approaches using *property monitors* that are usually integrated into the application code directly (e.g., by using metadata annotations) or by using aspect-oriented programming technologies (AOP).
- **Approach 3**. (Subsection 2.1.2.3) Approaches aimed at verifying GUI properties and input data in web-based user interfaces.

2.1.2.1 Data Verification Using Formal Logic

Barringer et al. [17] presented a small and general monitoring framework based on EAGLE. EAGLE is a general logic for reasoning about data which supports a mixture of future- and past-time operators. The monitoring process is performed state-by-state, i.e., without storing the execution trace. The validation rules can be parametrized with formulae or data. However, the interpretation scheme was complex and difficult to implement efficiently [18].

More recently, Barringer et al. introduced two rule-based systems for on-line and off-line trace analysis. RuleR [18] is a conditional rule-based system that uses single-shot rules. It uses a step-by-step basis to check a finite trace of observations against the rules at runtime. Developers can use RuleR along with a wide range of temporal logic and other formalisms. It uses AspectJ [47] to hook the verification code to the application code. LogScope [16] is a derivation of RuleR for monitoring logs offline. It is based on a higher-level pattern language and a lower-level RuleR-like automation language. Developers have to write mostly patterns. Then, the patterns are internally translated to automata to provide the process with extra expressive power. LogScope works offline and standalone.

Kim et al. described MaCS [107] to check the execution of a target program and to perform runtime correction of system behavior. The requirements are defined as a set of constraints specified in a language with formal semantics, and linked to events that denote when a requirement has to be checked. Then, this specification is used to generate a filter, an event recognizer and a runtime checker. MaCS uses the filter and the event recognizer to keep track of changes. The runtime checker determines if the current execution history satisfies the requirement specification. If not, the information collected during the process is used to steer the application back to a safe state.

Zee et al. described in [232] a runtime checking process incorporated into the Jahob verification system described by Kuncak [115]. The process combines static and runtime checking. It uses an inference engine that automatically computes a loop invariant to prove that there are no errors during the execution. The specifications are based on the Jahob language, allowing developers to use formulae in higher-order logic. The developers directly write the requirements specification in the source code.

2.1.2.2 Data Verification Using Formal Property Monitors

Chen et al. described MOP [160], a specification language to define safety properties. The specifications are converted then into property monitors, which are integrated as separated aspects into an application by using AOP. MOP is an event-based language, where an event is a state snapshot extracted from the running program. JavaMOP [97] is an implementation of MOP that generates AspectJ code for monitoring. When the specification is violated the corresponding handlers are called. These handlers are Java code snippets implementing logging routines, runtime recovery, etc. JavaMOP allows multiple differing logical formalism.

Bodden described in [25] a lightweight tool to implement verification into Java applications. It uses next-time free linear-time temporal logic (LTL) formulas that are directly specified into the source code as metadata annotations. Later, this author presented J-LO [26], a tool using AspectJ pointcuts to specify the assertions written in LTL. These solutions do not require an enhancement of the Java language. However, they imply a high intrusiveness in the application code, specially the solution using metadata annotations. More recently, Bodden presented MOPBox [27], a Java library to define parametric runtime monitors through a set of API calls. The events are bound to the monitors by using AspectJ aspects, or by using other notification sources like, e.g., an application interface. When the monitors reach an error state, this error is notified through a call-back interface.

Falcone et al. presented a unified view of runtime monitors both for properties monitoring and enforcement [63]. The *enforcement monitors* halt the underlying program if the execution sequence deviates from the desired behavior. They are based on the *Security Automata* (Schneider [198]) and the *Edit Automata* (Ligatti et al. [123]). The former was presented as the first runtime mechanism for enforcing properties. The latter is able to insert new actions by replacing the current input when safety properties are violated.

2.1.2.3 Data Verification in GUIs and in the Web

RAVEN [65], described by Feigenbaum and Squillace, is one of the few verification approaches specially oriented to GUIs. However, it is aimed at verifying design properties. It works like a property inspector and uses validation rules written in a XML-based language. The rules are associated with particular Java types. Then, they are checked against the GUI widgets in design time (i.e., offline). The rules can also contain code from script languages like, e.g., Jython [188]. RAVEN is implemented in Java, and provides the *Architecture interface* to extend some aspects of the validation process.

Data verification also represents a very important concern in web technologies. For example, ASP.NET [110] uses *validators*. The validators are visual controls (a control is an input element, e.g., a button or a text field) that check a specific validity condition of another control. It uses five types of validators whose code is inserted along with the code that describes the forms. The process can be also supported by script methods that are executed on the server. The validators can provide immediate feedback, allowing users to correct bad input dynamically.

HTML5 [1] uses annotations (i.e., pieces of HTML code) inside the form specification to check basic properties of the user input before the form is submitted. It provides style annotations and script code bindings to visually enhance the feedback to users and thus providing a more dynamic verification process. It also provides a constraint validation API to statically validate more advanced constraints, e.g., value ranges, pattern matching.

A different solution for web applications was described by Hallé et al. in [79]. BeepBeep is a Java-based runtime monitor that uses a filter in the server to intercept incoming messages. Data in the messages is validated according to an interface contract written in a separate text file. Invalid messages are blocked. It uses an extended Linear Temporal Logic called LTL-FO+ specifically designed to address web application interface contracts. Therefore, the requirements can only be defined using this logic and the checking process is limited by the functionality it provides. It allows to enable and disable the rules at runtime according to the verification needs.

2.2 Group 2: Approaches Describing and Analyzing the User-System Interaction Process as a Whole

The approaches included in this group are intended to describe and analyze the interaction process as a whole, including also the environment and some user characteristics. Subsection 2.2.1 summarizes some of the most relevant works aimed at describing and analyzing multimodal interaction. Subsection 2.2.2 describes approaches aimed at collecting subjective and cognitive data from the users. Finally, Subsection 2.2.3 includes the description of some relevant approaches analyzing and modeling the context of applications.

2.2.1 Analysis of User-System Interaction

The interaction between the user and the system has been described, modeled or analyzed in the literature for very different purposes. It includes the development of multimodal interfaces, as well as the evaluation of the quality of interaction and the usability of systems. This section summarizes some of the most relevant works related to the description and analysis of multimodal interaction. The analyzed approaches are organized into three different groups according tho their main purpose:

- **Multimodal development**. (Subsection 2.2.1.1) This subsection includes a survey about notations and languages aimed at supporting design and development processes of multimodal interfaces.
- **Interaction analysis**. (Subsection 2.2.1.2) This subsection describes different approaches located in the fields of evaluation of multimodal interfaces and interaction analysis. Works in this subsection show different ways to assess interaction within a unimodal or multimodal system.
- **User experience analysis**. (Subsection 2.2.1.3) This subsection includes the description of several approaches modeling interaction with the aim of evaluating the users behavior and analyzing their experiences with the software.

2.2.1.1 Analysis for the Development of Multimodal Systems

Model-based approaches are very present in Human-Computer Interaction (HCI) and have evolved in order to suit the design and development of new user interfaces. Paternò et al. distinguished in [184] between four different generations:

- (G1) Approaches focused basically on deriving abstractions for graphical user interfaces (e.g., UIDE [68]).
- (G2) Approaches focused on expressing the high-level semantics of the interaction, e.g., by using task models (e.g., ADEPT [98] and GTA [212]).
- (G3) Approaches focused on dealing with different characteristics and input/output modalities of interactive platforms to facilitate the development of multiple versions, e.g., for mobile applications. In this subsection we mainly focus on approaches of this generation.
- (G4) Approaches dealing with ubiquitous environments in which applications, distributed everywhere, exploit a variety of sensors and interaction modalities (e.g., [44] and [213]).

Descriptions based on markup languages

Many approaches focused their efforts on finding a common notation to describe the different aspects of interaction for the development of multimodal systems. Some of them presented new notations based on markup languages, as the ones described in the following.

The Multimodal Utterance Representation Markup Language (MURML) [113] represents multimodal utterances as a composition of speech output augmented with gestures, and tries to define the correspondence between these two modalities. MURML combines the representation of cross-modal relationships within hierarchically structured utterances with the description of gestural behaviors.

Another example is the Multimodal Interaction Markup Language (MIML) [12] aimed at describing semantic interaction for different platforms. This language provides a three-layered description of multimodal interaction systems focusing on the tasks related to the user, the system and the data models, the interaction (i.e., input and output) in different modalities, and the devices in use in the target scenario.

The Device-Independent MultiModal Markup Language (D3ML) [72] is an example oriented to web-based applications. This domain-specific language is used to describe web-based user interfaces regardless of input and output modalities. Such descriptions allow the system to dynamically adapt the user interface depending on the current input and output capabilities. D3ML does a remarkable effort in aggregating all meta-information that is relevant for rendering content on arbitrary devices and/or modalities.

The eXtensible Markup language for MultiModal interaction with Virtual Reality worlds (XMMVR) [176] integrates voice interaction in 3D worlds, allowing users to manage objects by using speech dialogs. This language does not provide a common "vision" of different modalities, but it is a hybrid markup language embedding VoiceXML [179] for vocal interaction and VRML [37] to describe 3D scenes and actors.

The Extensible Multimodal Annotation markup language (EMMA) [99] represents a set of W3C recommendations within the Multimodal Interaction Framework [118]. It describes an XML markup language for annotating the interpretation of user input in different modalities. EMMA is focused on annotating single inputs from users, also combining information from multiple modes. Inputs are structured along with their interpretations.

Descriptions based on interaction models

Other approaches used models instead of markup languages to describe the different aspects of multimodal interaction, and thus ease the development of multi-device user interfaces.

Palanque and Schyn presented in [183] the ICO (Interactive Cooperative Objects) notation.

The authors presented a formal description technique that provides a formal and precise way for describing, analysing and reasoning about multimodal interactive systems. It uses concepts from the object-oriented approach to describe the structural aspects of a system. It also uses high-level Petri nets to formally describe the behaviour of the system, and thus allow its formal analysis and reasoning about the model. ICO allows also to model and verify the fusion mechanisms for input in multimodal systems. ICO models are executable, providing simulation capabilities already before an application is fully implemented.

Vanacken et al. described NiMMiT (Notation for Multimodal Interaction Techniques) [214], a graphical notation for modeling multimodal interaction techniques in 3D environments. It is a state- and event-driven notation that allow designers to describe the application by using high-level diagrams. Interactions are described as state diagrams, which use events that can be generated by different types of devices. Multimodality is accomplished using combinations of events from different families. NiMMiT allows the creation of abstract representations of interaction that can be easily adapted according to the multimodal interaction scenario.

Manca and Paternò described in [129] a XML-based logical language modeling multimodal interaction. It was mainly aimed at supporting the development of graphical-vocal user interfaces. It uses components, which are first defined using an abstract level. The components are defined by using an authoring environment, in which designers can work through logical descriptions of the user interface and choose the most suitable combination of modalities. These components are then translated into more specific ones depending on the modalities used and the CARE properties of the target system.

This language is based on MARIA (Model-based lAnguage foR Interactive Applications) presented by Paternò et al. in [184]. This language enables designers to create more specific user interface languages according to the CAMELEON Reference Framework. These new languages take into account the target GUI platform and the interaction modalities used in the target system. At design time, the language is used to create user interface-related annotations and thus provide hints for its development. Then, at runtime, the language is exploited to support dynamic generation of user interfaces. MARIA can be used independent from used modalities or concrete GUI platform implementations.

CAMELEON was described by Balme et al. in [15]. It describes a reference framework for classifying interfaces supporting multiple targets or multiple contexts of use. This framework is structured into four levels of abstraction: description of the tasks and concepts, the abstract user interface (AUI), the concrete user interface (CUI) and the final user interface (FUI). These four levels are identified and organised independently with respect to the context in which the FUI is used. CAMELEON covers both the design and runtime phases, structures the development life cycle and provides a unified understanding of context-sensitive interfaces. Context is defined by the authors as a triple <user, platform, environment>.

Another approach aimed at modeling multimodal interaction is Damask, proposed by Lin and Landay in [124]. Damask enables designers to sketch and generate multidevice user interfaces. Designers have to define patterns and layers to indicate high-level concepts of their design. Then, Damask uses them and a set of pre-built UI fragments to generate designs for several devices, which have to be finally refined by the designers.

Meskens et al. presented a similar solution in [161] called Gummy. This tool works as a multiplatform UI builder that, by adapting and combining features of existing user interfaces of an application, can generate an initial design for a new platform trying to keep all user interfaces consistent.

2.2.1.2 Evaluation of Multimodal Interaction

This subsection describes evaluation approaches for Speech Dialog Systems (SDS) and Multimodal Dialog Systems (MMDS). These approaches can be classified into four different groups according to the nature of the metrics used for the evaluation process:

1. Parameters quantifying user-system interaction
2. Parameters describing user efficiency and task success
3. Parameters measuring modality efficiency
4. Metrics based on the observation of user behavior

Evaluation based on interaction parameters

Fraser proposed in [70] a common set of metrics to measure the performance of SDS based on the EAGLES (Expert Advisory Group on Language Engineering Standards) recommendations [120]. The author defined key aspects of the system, the test conditions and the test results. His aim was to arrive at criteria which could facilitate comparison across systems, describing also what to evaluate and report, and how to do it. Dybkjær et al. discussed some problems related to such metrics in [58]. One of them argues that the methodology can be difficult to follow and may not fit equally well into projects with different contexts.

Möller provided in [167] an overview of interaction parameters which have been used to evaluate SDS in the past 20 years. The author presented a characterization of these parameters including the interaction aspect each one addresses, as well as the measurement methods required to determine them. This work included also an overall description of the parameter extraction process and the level these parameters are instrumented during the dialog. Furthermore, it is standardized in ITU-T Suppl. 24 to P-Series Rec. [166].

Parameters to evaluate SDS have also been used as a basis to define new metrics to evaluate MMDS. Möller and Kühnel et al. recommended in [114, 168] a set of parameters to describe user interaction with multimodal dialog systems. These parameters are aimed at quantifying the interaction flow, the behavior of the system and the user as well as the performance of input and output devices. This approach was not only aimed at transferring some spoken dialog parameters to a multimodal context, but the authors also proposed new parameters inherent to multimodal interaction. The authors presented an experiment

within a multimodal smart-home system, in which such parameters were used to evaluate user interaction.

Evaluation based on efficiency and success parameters

Other approaches focused on assessing usability in a more predictive way, basing on parameters describing user efficiency and task success. This is the case of PARADISE [217], a framework to compare different spoken dialog strategies in a SDS. This framework considers user satisfaction as a measure of system usability, which is objectively predicted by measuring task success and dialog costs. Task success is measured using attribute value matrices (AVM), which describe the aim of a dialog. Dialog costs are calculated using cost functions. Then, the relevance of these values for the system performance is weighted via multiple linear regression.

PROMISE [22] extended PARADISE for the evaluation and comparison of task-oriented MMDS. The authors described a new way to define system performance. They split the performance function of PARADISE into two parts, reducing the formula to normalized cost functions first. Then, the authors defined an alternative way to calculate task success. The result was a new formula to evaluate multimodal systems performance, since different AVMs with different weights can be computed.

Perakakis et al. also used SDS efficiency parameters to assess interaction in MMDS [185, 186]. The authors described two new objective metrics (i.e., efficiency and synergy) to identify usability problems and to measure the added value from efficiently combining multiple input modalities. Their results demonstrated how multimodal systems should adapt in order to maximize modalities synergy and to improve usability and efficiency of multimodal interfaces.

Evaluation of modality efficiency

Other approaches focused on defining new parameters to determine the most suitable combination of modalities and thus maximize system quality. For instance, the CASE (Concurrent, Alternate, Synergistic and Exclusive) properties [173] describe a classification space that describes the properties of both input and output interfaces of a multimodal system. This classification is based on the concurrency of data processing and the fusion of input/output data.

This work was extended to the CARE (Complementarity, Assignment, Redundancy, and Equivalence) properties [50] to assess aspects of multimodal interaction, specially for user

input. *Complementarity*, *assignment*, *redundancy* and *equivalence* are the properties used to denote the availability of interaction techniques in a multimodal user interface, as well as to predict usability during the design of a system.

Lemmelä et al. described in [121] a 5-step iterative process for identifying issues affecting the usefulness of interaction methods and modalities in different contexts. This process was mainly focused on evaluating applications using tactile and auditory cues. It provides a description of those parameters affecting the suitability of a particular modality in a specific context. The process uses these parameters to select the best option for each case.

Evaluation based on the observation of user behavior

Many approaches based the assessment of usability of MMDS on the observation of user behavior. For example,Balbo et al. described in [14] an approach to record user interaction data aimed at detecting concrete user behavior patterns, e.g. direction shift, action repetition or action cancellation. The discovered patterns were used to analyze deviations from a data flow-oriented task model in order to identify potential usability problems.

Damianos et al. presented in [52] a methodology for evaluating multimodal groupware systems. The authors use the *Multi-modal Logger* described in [19] to record user behavior. These records are then combined with human observations and user feedback to detect usability glitches in systems under development.

Strum et al. used the MATIS (Multimodal Access to Transaction and Information Services) system [205] to explore ways to maximize usability by combining speech and GUI interaction. The authors automatically logged user interactions to measure usability at different levels (i.e., unimodal and multimodal) and described to what extent users notice and use the extra interaction facilities that are available in MMDS.

Martin and Kipp described in [132] Tycoon, a theoretical framework to study the multimodal behavior of recorded human subjects. Tycoon specifies four different types of cooperation between modalities: equivalence, specialization, complementarity and redundancy. It offers a coding scheme and analysis metrics aimed at analysing, e.g., how redundant the behavior of a user is, how much the user relies on specific modalities, the different switches between modalities, etc.

Serrano et al. described in [201] a component-based approach for developing and evaluating multimodal interfaces in mobile phones. This approach captures usage data in realistic situations and later implements in-field evaluations. Data is captured at different levels of

abstraction, i.e., device, interaction, composition and task. The process supports continuous user evaluation in an iterative development process.

2.2.1.3 Evaluation of User Experiences

Some approaches model the interaction between the user and the system with the aim of evaluating the users behavior and analyzing the quality of their experiences with the software.

One example is the CUE-Model, presented by Mahlke and Thüring in [127]. This model integrates data concerning user interaction, the user experience and overall judgments of system quality. UX is divided into three components: perception of instrumental qualities (e.g., controllability, learnability) and non-instrumental qualities (e.g., visual aesthetics), and emotional reactions caused by these perceptions (e.g., subjective feelings). The authors also conducted an study from which concluded that both quality aspects (i.e., usability and aesthetics) significantly influence emotional reactions.

Schulze and Krömker described in [199] a conceptual framework to provide a uniform basis for UX measurement. Their goal was to analyze UX influencing factors (i.e., human and system aspects, emotions, a spatiotemporal dimension and motivation) and to obtain indicators for product optimization. The authors designed an evaluation method by using techniques to collect qualitative (via observation and open questions in interviews) and quantitative user data (via Likert scales and Semantic Differentials).

Ali et al. described in [7] a conceptual framework to measure software quality in mobile learning contexts. This framework is based on the ISO/IEC metrics (e.g., scalability, service quality, etc.). The evaluation process combines structural factors, the dimensions of learning context, and design issues aimed at addressing the learning objectives of the users or the platform. The authors remarked the need of using metrics related to the context of use and quality in use.

Other approaches do not use any model and are based on the direct observation of the user. This is the case of OneClick, a framework presented by Chen et al. in [42] that aims at capturing the users perceptions when using network applications. The authors used a simple method in which users are asked to click a button whenever they feel dissatisfied with the quality of the application in use. By quality of the application the authors meant all the QoE dimensions that affect users satisfaction (e.g., poor voice quality, screen freezing,

etc.) The method was really intuitive, time-aware, and had an application-independent nature.

2.2.2 Analysis of Subjective Data of Users

The analysis of subjective opinions and impressions of the users is essential to evaluate the quality of the experiences (QoE) of these users with a software or a system. It is also necessary the analysis of some other features like users mood or their attitudes. Several of the most relevant works aimed at collecting subjective and cognitive data from the users are summarized in the following. The different approaches are divided into two groups according to the nature of analyzed data:

- **Analysis of user ratings**. (Subsection 2.2.2.1) This part describes several approaches aimed at collecting the user perceived quality of a software, as well as at measuring the user experiences with the software. It is also described some approaches discussing the appropriateness of such methods for the evaluation of multimodal systems.
- **Analysis of users state and attitudes**. (Subsection 2.2.2.2) This part introduces first an approach describing the influence of users mood and attitude toward technology use on the final user-perceived quality. Then it is summarized several approaches aimed at analyzing and measuring these two influencing factors.

2.2.2.1 User Ratings Collection

Questionnaires represent an effective mean for extracting subjective information from users. One example is AttrakDiff [83] presented by Hassenzahl et al. This questionnaire aims at measuring the attractiveness of interactive products. Based on the Hassenzahl's model of UX, it uses 28 pairs of opposite adjectives to evaluate attractive (e.g., ugly–beautiful), hedonic (e.g., cheap–premium) and pragmatic (e.g., unpredictable–predictable) attributes of a product. There exists a lighter ten-item version called AttrakDiff mini [84].

SUS (System Usability Scale) [30], presented by Brooke, is a simple ten-item attitude scale (five positives, five negatives) giving a global view of subjective assessments of the usability of a product. It presents statements such as "I thought the system was easy to use", which are evaluated by using a five-point Likert scale. It has been used for both research and industrial evaluations.

Kirakowski and Corbett described SUMI (Software Usability Measurement Inventory) [109], a fifty-item questionnaire aimed at measuring user satisfaction and assessing user perceived software quality. It provides measures of global satisfaction, as well as of five more specific usability areas, including effectiveness, efficiency, helpfulness, control, and learnability. It contains questions such as "Using this software is frustrating" that are rated using the *agree*, *undecided* and *disagree* values.

SASSI (Subjective Assessment of Speech System Interfaces), described by Hone and Graham in [87], is a questionnaire aimed at providing reliable measure of users subjective experiences with speech recognition systems. It is composed of fifty attitude statements. Each statement is rated according to a seven point scale from *strongly agree* to *strongly disagree*.

Wechsung and Naumann compared in [220] the aforementioned questionnaires, plus a self-constructed one, in order to investigate to which extent these methods for the evaluation of unimodal systems are appropriate for the evaluation of multimodal systems. The results showed that, in general, questionnaires designed for unimodal systems are not applicable for the evaluation of usability in multimodal systems, since they seem to measure different constructs. Nevertheless, AttrakDiff showed the most concordance when evaluating unimodal and multimodal systems. It was mainly due to its rating scale, that is applicable to all systems, and to the use of pairs of bipolar adjectives, which are not linked to special functions of a system.

In [172] the authors did a similar comparison to study the correlation between objective data (i.e., log files) and subjective data (i.e., questionnaires) in multimodal interaction environments. The contradictory findings showed again that, in general, questionnaires designed for unimodal interfaces should not be used as the only data source when evaluating multimodal systems. AttrakDiff presented again the best correlation results. Thus the authors concluded that this questionnaire provides a proper basis to implement a reliable, valid, and more specific questionnaire for multimodal interfaces.

In [219] Wechsung et al. conducted two new studies to investigate the relationship between user subjective ratings of multimodal systems and user subjective ratings of its single modalities. In this case the authors used the AttrakDiff questionnaire basing on previous successful results. They argued that it is the only questionnaire, among the analyzed in their previous work, yielding valid and reliable results for the evaluation of multimodal systems.

In order not to overwhelm the user during test sessions, Wechsung et al. used in a subsequent work [221] the "mini" version of AttrakDiff to collect retrospective quality judgments of the interaction with the system. This shorter version is more suitable for repeated assessments during a test session.

Other authors use self-constructed questionnaires to evaluate users perceived quality and particular aspects of interaction according to the goals of the study. One example is the work of Callejas and López-Cózar [34], who presented a report including empirical results extracted from the evaluation of interactions with UAH (Universidad Al Habla) [32], a SDS providing spoken access to academic information. They studied how quantitative aspects of interaction (i.e., objective measures) affect the quality judgments of users (i.e., subjective measures). Interaction parameters were used to measure system performance and the dialog course. Moreover, the authors used a self-constructed questionnaire to allow users to express their opinions about different interaction aspects. The questionnaire is composed of eleven questions about previous expertise, interaction issues, and perceived performance. The questions are rated by using 5-item, custom scales.

The authors also used self-constructed questionnaires with the aim of evaluating different dialog management strategies in UAH [35]. The subjective measures provided by the questionnaires, along with other quantitative measures, were used to assess the quality of the strategies learned, and thus to select the optimal ones.

2.2.2.2 Users Mood and Attitude Measurement

In [221] Wechsung et al. analyzed the influence that attributes like users personality, attitude, mood and cognitive abilities have in interaction quality perception. The authors showed that, while cognitive abilities and personality traits do not influence quality perceptions and interaction behavior, attitude towards technology does. They also showed that a positive mood is linked to positive quality judgments. As conclusion, they highlighted the importance of these two attributes when analyzing evaluative judgments of users.

In order to measure users mood, the authors used an adapted version of the seven faces scale described in [9]. Faces scales have been largely used in the literature to measure human satisfaction, mood or even pain. Its effectiveness measuring users emotions has been showed in many works. For example, Wong and Baker defined his own face scale to measure pain of patients in [226]. Agarwal and Meyer used a face scale in [5] to collect emotional response data during a usability study, and then explore the differences between

responses of users. In [78] Gulur et al. used a face scale to evaluate pain and mood of young patients.

Others methods use larger and more complex evaluation scales. This is the case of SAM (Self-Assessment Manikin) [117] developed by Lang. This method uses three scales to measure pleasure, arousal and dominance. Each scale is composed of five graphic characters representing emotions. During the study, the user places an "x" over any of the five figures in each scale or between them, resulting in a nine-point rating scale for each dimension. SAM has been largely used during years to measure emotions of users. For example, Mahlke and Thüring used SAM in [127] to measure the quality and intensity of emotional reactions of users. Minge used SAM as well in [163] to analyze the dynamics of UX, i.e., the changes in user experiences over time.

Other works aimed at measuring users attitude towards the use of information and communication technology. Ray et al. analyzed in [189] the attitude of men and women by using an inventory aimed at identifying attitudes associated with gender issues reflected in the literature. The inventory was composed of nine statements. The user had to select the level of agreement with each statement in a five-point scale that ranges from Strongly Agree (5) to Strongly Disagree (1).

Ogertschnig and Van der Heijden presented in [175] an study to measure attitudes towards using mobile information services. As a tool for conducting the experiment the authors proposed a new, short-form version of an attitude scale containing two subscales: hedonic and utilitarian value. Each subscale is composed of five items, each one evaluated using a five-point scale ranged from "Not at all" to "Extremely".

Hassad conducted in [81] an exploratory study to examine the attitude of instructors toward technology integration for teaching. The author described a preliminary scale for measuring attitude toward technology integration. This scale, called ATTIS (Attitude Toward Technology Integration Scale), is composed of an homogeneous cluster of six attitude items which are rated by using a five-point scale. Each of these items represents a different facet of attitude. The results showed that this scale returns acceptable levels of internal reliability (consistency) and validity.

2.2.3 Analysis of Interaction Context

Understanding the surrounding context is essential for the evaluation of the user experience with a product. This is particularly true in mobile environments, where applications are dynamically used in different scenarios and social contexts. This section describes different approaches aimed at analyzing and/or modeling the context of use of applications. The summarized works are divided into two different subsections according to their main purpose:

- **Analysis of context factors**. (Subsection 2.2.3.1) This part describes approaches aimed at analyzing and discussing those contextual factors influencing the experience of users with a software.
- **Interaction context models**. (Subsection 2.2.3.2) This part includes the description of several approaches aimed at defining and structuring data related to the mobile context within a model.

2.2.3.1 Interaction Context Factors Analysis

Zhang and Adipat proposed in [233] a generic framework to guide the selection of research methodologies and attributes for testing usability in mobile environments. The authors also identified a set of challenging features of mobile interaction including the changing context, multimodality, connectivity issues, screen and display limitations, restrictive data entry methods and a limited processing capability and power.

Coursaris and Kim presented in [49] a review of 45 empirical mobile usability studies analyzing the interaction context. As a result, the authors described a framework of contextual usability including influencing factors (i.e., user, activity, environment, and product), the usability dimensions to be measured and a list of consequences being impacted by usability.

Wigelius and Väätäjä analyzed in [225] those contextual factors affecting the experience of mobile workers. The authors described a context divided into five dimensions: social, spatial, temporal, infrastructural and task. Their findings emphasized the influence of social and infrastructural factors in the final worker experience.

Korhonen et al. presented in [112] a detailed context categorization structured into eight categories: physical context, user, social context, location and time, device, connectivity, task and service. They studied also how contextual factors influence the UX of personal

mobile products. The authors also introduced the concept of *triggering context* referring to those contextual factors that significantly affect the users experience, and the concept of *core experience* to denote the most meaningful experience for a user in an episode of usage.

Wac et al. analyzed in [216] the QoE of users in different daily life situations. The authors identified six main factors influencing user ratings: user location and current time, user previous experience, alternative devices availability, appropriateness of the application for mobile devices, social context and attitude towards using mobile devices. The authors used the Context Sensing Software (CSS) to automatically collect context parameters that influence users QoE in Android devices, especially for highly interactive applications.

Ickin et al. analyzed in [91] the experiences of different users using mobile applications, as well as the external factors influencing these experiences. The authors used a blend of quantitative and qualitative interaction data. To extract the data they used the CSS for Android to collect quantitative interaction values, and the Experience Sampling Method (ESM) to collect qualitative and subjective data from users.

2.2.3.2 Interaction Context Modeling

Henricksen et al. described in [86] a model of the context in pervasive computing environments. This model definition is structured around three main entities: people, communication devices and communication channels. The model provides a set of properties for each entity (e.g., the id of a communication channel) which are linked to the entities by means of static (i.e., fixed) and dynamic (i.e., changing) associations.

Ryan and Gonsalves modeled in [194] the context on mobile usability. They considered the interaction between the user and the device, as well as the physical environment and the runtime software environment. The authors emphasized the importance of device and software environment parameters in the model, mainly due to the influence of the application performance over the final user performance.

Thurnher et al. proposed in [210] a context framework to evaluate mobile usability through the observation of user behavior. The authors described a model composed of parameters derived from the five human senses, even including taste and smell. The values are gathered by using environment sensors, thus avoiding additional distraction of subjects caused by observers. This method provides a different approach to evaluate mobile usability from sensed data. The authors discuss about its potential for explaining phenomena that could not be explained by traditional in-situ methods.

Chepegin and Campbell presented the NEXOF-RA [43], a model describing the context-of-use of a user interacting with a particular system. The model is an aggregation of the following three entities: user, platform and environment. The state of each entity is described using a set of properties (e.g., position, age, CPU speed) that are obtained from different context providers. A sub-model is used to describe the platform entity, in which the interaction is performed. It is represented as an aggregation of different components (e.g., device, web browser, network, camera).

Jaroucheh et al. described in [96] a 5-layer architecture to model contextual situations, as well as an algorithm to track and identify such situations. The architecture uses information from the user behavior and the context events to recognize contextual situations. Such situations are described as a set of different states considering the context history. The authors used linear temporal logic (LTL) to check alignment between the observed situation and the expected one.

In an attempt to categorize approaches like the described above in this subsection, Bolchini et al. described in [29] a set of features to characterize context models. The authors also described a 5-type classification according to their main use. For this purpose, different context-aware approaches were analyzed focusing on the problem of data tailoring. The authors also presented a framework to choose among the available models, or to define the requirements of a new context model instead.

3

Evaluating Quality of System Output

System output is a main component of the human-computer interaction process. Testing is essential to validate the system response and thus assuring the quality of the output provided to the user. This is specially true in graphical user interfaces (GUI), the mean by which the output is provided to the users in most applications nowadays.

Applying GUI testing into a project is not straightforward. It is time consuming and often involves activities that are implemented manually by specialized personnel. The integration of GUI testing tools is also troublesome, mainly due to the use of different execution contexts (e.g., different GUI platforms).

This chapter presents OpenHMI-Tester (OHT), a new application framework for the development of GUI testing tools. Its open design is aimed at supporting major event-based GUI platforms. Its implementation is cross-platform. Furthermore, it can be easily integrated into ongoing and existing applications due to it being not code-intrusive. As a result, the OHT provides an adaptable, extensible, scalable, and robust basis to support the automation of GUI testing processes.

3.1 Introduction and Motivation

Testing is used to assure the quality of the system output as well as the reliability and robustness of a software in the given context in which it is intended to work. This is specially true in the case of graphical user interfaces (GUI), where the testing stage is critical before the software is accepted by the final user and put in execution. These days GUIs are present in most of developments, and can constitute as much as 60 percent of the code of an application [153]. Therefore, integrating GUI testing in current developments is essential to achieve quality.

Nevertheless, the integration of advanced GUI testing tools in most of developments is not a fact nowadays. It is known that testers are used to apply good testing practices such as unit testing to check the functionality of their applications, but their scope is frequently limited to the parts without GUI [40]. Moreover, techniques developed to test the core functionality of a software can not be directly applied to test the information and the features of a GUI [158].

Many authors claim against current approaches for GUI testing and highlight a special need for more efficient approaches [6]. GUI testing is often time-consuming, as well as expensive in terms of human resources because it usually depends on highly specialized personnel [182]. Many of the available methods for automated testing are incomplete or involve activities that, in most of cases, have to be implemented manually [154].

Due to the iterative process of GUI development, manually designing and maintaining the tests requires too much effort [6]. A simple change like rearranging the GUI widgets may involve, for example, to modify all the test scripts [40]. GUI testing methods should be more robust and provide a better tolerance to modifications.

Moreover, integrating current GUI testing approaches into a development is not straightforward. Normally, applications do not allow the intrusiveness of testing tools into their source code. Thus, such tools should provide a non-invasive mechanism to test a GUI. Applying GUI testing is also troublesome due to the different execution contexts for which applications are created. E.g., the developers can choose from a wide range of available GUI platforms. If a GUI testing tool is aimed at testing interfaces from a specific platform, then it can not be used into a project with a different GUI system. GUI testing architectures should be open to features like this.

Additionally, related to the problems described above, we would like to pose the following research questions:

> **Q1**: Can a GUI testing method and its processes be agnostic to a core concept such as the GUI platform?
> **Q2**: Can a GUI testing method simulate the interaction of a human tester in order to enhance the tolerance to modifications in the GUI while executing a test case?

To attempt to solve the aforementioned problems and to give an answer to the research questions, Section 3.4 describes the design of OpenHMI-Tester (OHT), a new application framework for the development of GUI testing tools. Before this, some testing requirements and previous design considerations are described in Sections 3.2 and 3.3, respectively.

The OHT provides a basis to automate testing processes based on runtime GUI instrospection and behavior observation (e.g., functional and regression testing, GUI convergence checking). A capture/replay approach [178, 192] is used to extract GUI properties and user behavior to create test cases at runtime. Then, the test cases are executed in the real application to validate the software response. It enables the implementation of a scalable testing process.

The OHT is not code-intrusive because it implements a transparent application hooking thanks to the use of DLL injection. It provides also high tolerance to modifications in the GUI design during development, as the human interaction is simulated "from the deep" during test case playback. Moreover, the OHT provides an open and adaptable design aimed at supporting major event-based GUI platforms.

A cross-platform version of the OHT framework was implemented and provided as a contribution to the open-source community. The details of this implementation are described in Section 3.5. Finally, the proposed solution is discussed in Section 3.6 and some conclusions are provided next in Section 3.7.

3.2 GUI Testing Requirements

Before discussing the details of the OHT architecture design, the requirements that provide the driving philosophy behind this design are introduced. These requirements have been ex-

tracted from our experience in collaborating with SAES [202], a military software company specialized in medium and large GUI-based developments.

The OHT is aimed at providing a fully functional capture and replay framework to implement GUI testing. According to the problems described above, the main requirement is to provide an open design in terms of the operating system, the windowing platform, the capture/replay functionality and the representation of the test cases. The OHT should be also easily integrated into new and existing developments.

Cross-platform. To provide a cross-platform architecture is mostly an implementation concern, which is later discussed in Section 3.5. However, it is a design concern to propose a flexible and adaptable architecture to allow the use of potentially any windowing system (e.g., Qt [174], GTK+ [208]). Thus, the proposed architecture should be agnostic to this feature, at least, when implementing its core functionality.

Functionality. The basic functionality of a capture and replay tool should be provided. These processes should be open enough to be enriched with new functionality if needed (e.g., to implement automatic validation of the software response during test case playback). The replay process should work into a real execution of the application and "from the deep", i.e., reproducing as faithfully as possible the interaction of a real human with the tested application. This process should be also robust and tolerant to changes in the GUI and missing objects, as well as to window moving and resizing actions during playback.

Additionally, the OHT architecture should be open to the implementation of new and advanced testing features like, e.g., property picking to allow the tester to check the properties of a selected object, screenshots to support visual validation, etc.

Test cases representation. The OHT architecture should enable the integration of different test case representations. The developers should be allowed to use their own method to encode the test cases (e.g., based on a markup or script-based language). Regardless of the representation in use, the framework should provide generic storing and retrieving functionality during the capture and replay processes, respectively. The developers should also be enabled to add new events or actions to extend the functionality of the test cases (e.g., pauses, breakpoints, messages, etc.)

Integration. It is also required that the OHT has to be integrated into new, ongoing and already existing developments. Therefore, the proposed framework should provide a transparent, automatic and not code-intrusive application hooking to the tested application.

3.3 Preliminary Considerations for the Design of a GUI Testing Architecture

This section includes some preliminary considerations to the design of the OHT architecture. Concretely, it discusses who are going to be the main participants in the GUI testing process, how the generated test cases have to be structured, and what kinds of data and information will be generated and exchanged in such a GUI testing architecture.

3.3.1 Architecture Actors

In order to avoid confusion for the reader, in this work we distinguish between two different roles or actors within the GUI testing process.

On the one hand we have the figure of the *tester*. This actor represents the human that interacts with the GUI testing tool to test the target application. He/she uses this tool to create and organize the test cases, and later to execute them against a real execution of the application to test.

On the other hand we have the figure of the *developer*. This actor is responsible for adapting the GUI testing tool to a specific testing environment in the case that some of its features change (i.e., operating system, GUI platform and data model). This actor needs to have a detailed knowledge of the GUI testing architecture for its adaptation.

3.3.2 Organization of the Test Cases

Many of the testing approaches analyzed in this research work (see Section 2.1.1) used test suites to arrange a set of test cases related to the same target application (e.g., CppUnit [64] or jUnit [90]). A test suite will include, therefore, all the information needed during the test case playback process. Encapsulating the whole description of a set of test cases into a single object simplifies the internal testing processes in a testing architecture, as well as the communication between its internal modules.

A test suite can be structured into three levels, as described in the following:

- **Test Suite**: it includes a set of test cases referring to the same application and, usually, with a common testing goal. This element may also include some meta information and a reference to the tested application.

- **Test Case**: this element describes a set of ordered test items to be systematically executed into the target application. It may also include meta data such as a test case description, its purpose, etc.
- **Test Item**: it is the smallest element in the description of a test case. Each test item includes the information of a single action that can be executed in the target application.

3.3.3 Interaction and Control Events

In a GUI testing architecture there are several modules sending information to each other to notify, e.g., GUI actions, control signals, etc. For this purpose, the architecture can use "events" as a mechanism to encapsulate and exchange information in a generic and uniform manner.

Each event may include information from a different nature. This nature can be determined by using a `type` and a `subtype` values of the event object. The events in a GUI testing architecture can be classified in four groups according to their main purpose:

- **GUI Events**: contain information related to GUI actions (e.g., layout changes, mouse clicks, keystrokes). These events normally are related to a single GUI widget.
- **Non-GUI Events**: these events contain other relevant information about the interaction with the target application (e.g., timer events).
- **Meta Events**: custom events defined by the developer to implement actions that are not natively supported by the windowing platform (e.g., pop-up messages, a pause during test playback, to play an alert sound).
- **Control Events**: these events are also defined by the developer to implement control signaling between the internal modules of the GUI testing architecture (e.g. the "playback process start" event).

3.4 The OHT Architecture Design

The OHT architecture is structured into two main software elements, each one with a different purpose. On the one hand the *HMI Tester* (HT), that is aimed at controlling the record (capture) and playback (replay) processes. This element also manages the creation

and maintenance of the test suites. On the other hand the *Preload Module* (PM), the software element that is "injected" into the tested application with the aim of capturing user interaction data as well as executing new actions into the GUI. Both modules communicate with each other.

The OHT architecture can be also divided into final and adaptable modules. Final modules are those implementing generic functionality (e.g., the module implementing the record logic). These modules (depicted using non-colored boxes in Figure 3.1) represent most of the proposed architecture and their functionality never change. Otherwise, adaptable modules are those whose functionality can be adapted to support new testing features such us the operating system or the GUI platform (e.g., to include a specific GUI actions filter). These modules are represented by colored boxes in Figure 3.1.

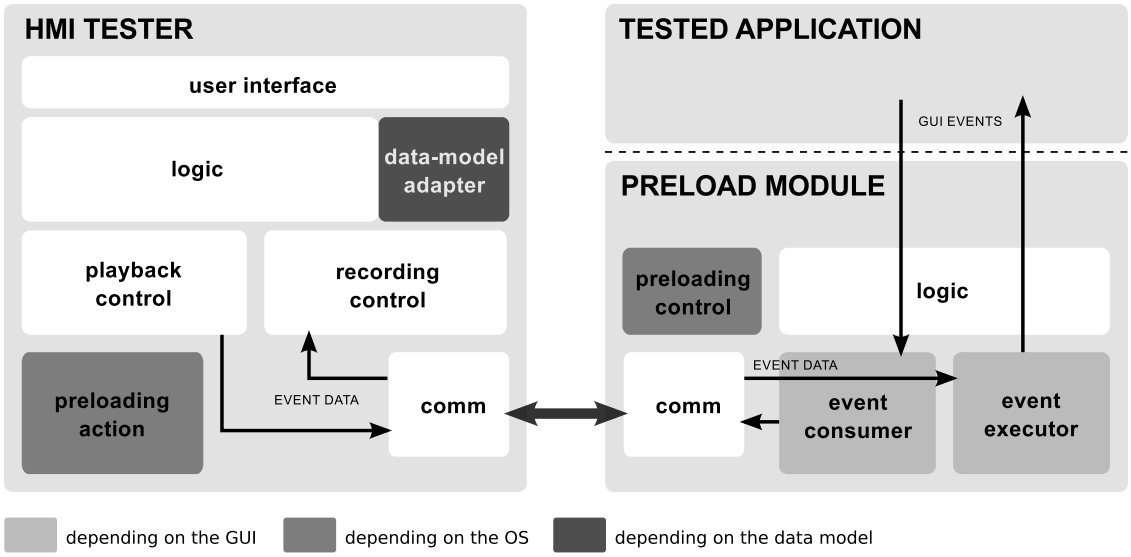


Figure 3.1 HMI Tester and Preload Module architecture.

As the reader can see in Figure 3.1, the capture/replay process involves the communication between the HT, the PM and the application to be tested (i.e., the target application). The HT and the PM can communicate directly, e.g., by using a mechanism like sockets. During the capture process, the PM sends to the HT those events generated by the user interaction with the target application that were captured by the event filters. During replay, the HM sends events to the PM for their execution into the target application.

However, how is it performed the communication between the PM and the target application? How can two independent software elements be connected to each other in a non-intrusive manner? The answer is by using library preloading.

The library preloading method [89, 171] enables the inclusion of new functionality into a "closed" application by preloading a dynamic library. For this purpose, the PM is implemented as a dynamic library including, among others, the functionality needed to perform the preloading action. Then, before the HT launches the target application for a capture or replay process, it first enables the `preload` option of the operating system and sets the PM as the library to preload. As a result, when the target application is launched, the PM is automatically loaded and deployed into the target application.

3.4.1 The HMI Tester Module Architecture

The *HMI Tester* (HT) is the software element used by the tester during the testing stage. It provides a graphical user interface to allow the tester to control the recording (capture) and playback (replay) processes. This element has two main duties. First, it is responsible for controlling the recording and playback of test cases. It also provides a mean to manage the lifecycle of a test suite, including its creation, adding new test cases, fill the test cases with the information received from the PM, etc.

As shown in Figure 3.2, the HT is composed of a set of modules including the functionality necessary to manage the recording and playback processes. It also includes a module aimed at controlling the preloading process, which depends on the operating system in use. Additionally, another adaptable module lets the developers adding to the architecture their preferred representation of a test suite, including load and save functionality. The most significant modules of this part of the OHT architecture are further described in the following.

DataModelAdapter. It is used to integrate a custom representation of the test suite into the OHT architecture. The developers can implement their own *DataModelAdapter* to provide the *DataModelManager* with the functionality to manage such a new representation.

PlaybackControl. This module is used to manage the test case playback process. It uses the *ExecutionThread* to send GUI events to the PM and, when necessary, including control signaling events to remotely guide the playback.

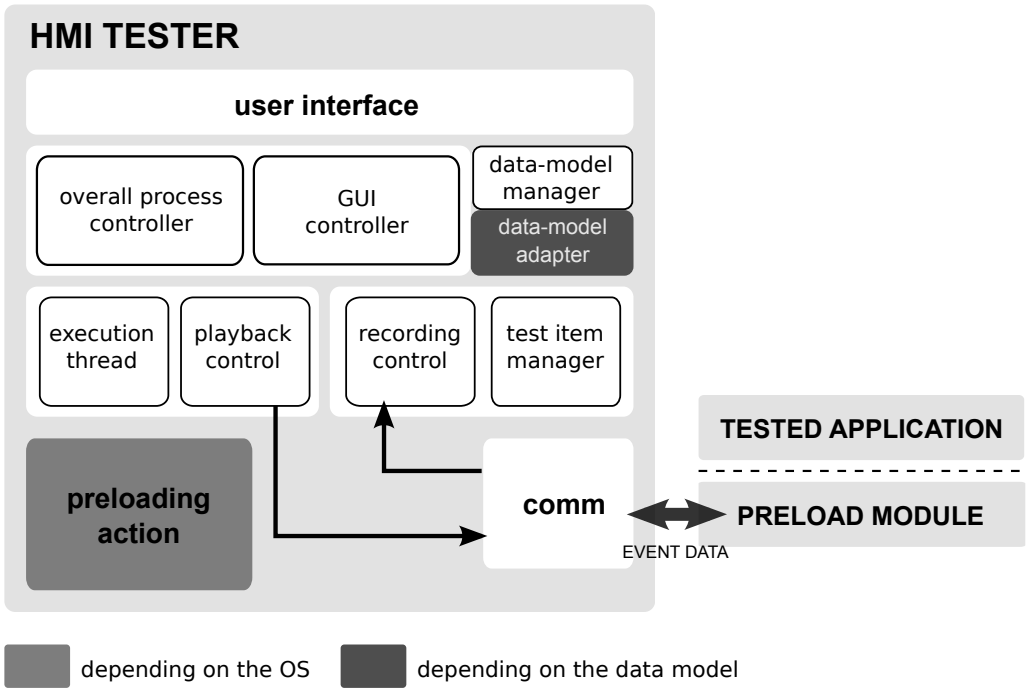


Figure 3.2 HMI Tester module detailed architecture.

RecordingControl. This module is used to manage the test case recording process. It sends control signaling events to the PM to control the process. It also receives from the PM the events captured during interaction, and stores them in the current test case.

PreloadingAction. This module is intended to control the preloading process both during test recording and playback. First, it enables the `preload` option in the operating system and establishes the PM as the library to preload. Then, it launches the target application.

Comm. This module encapsulates all the functionality related to communications. It is used in the HT to send new events to the PM during test case playback, and to receive event data from the PM during the test case recording process.

3.4.2 The Preload Module Architecture

The *Preload Module* (PM) is the software element that is hooked up to the target application to capture GUI events as well a to execute GUI actions received from the HT. Figure 3.3 depicts the architecture of the PM. It is distributed similarly than in the HT: some modules implement final and generic behavior (i.e., communication to the HT and main logic)

and other modules may be adapted, if necessary, depending on the GUI platform or the operating system. The modules implementing generic behavior are described next.

Logic. Its main task is the initialization process, during which the *Comm*, *EventConsumer* and *EventExecutor* modules are created and deployed into the target application. After initialization, the PM is hooked up into the target application.

Comm. This module works similarly as described above in the HT architecture. During recording it is used to send the captured events data to the HT. During playback it receives from the HT the events to be executed. In both processes, this module properly delivers control events to the *Logic* and the rest of the events to the recording or playback modules.

Regarding the requisite of designing an open architecture, some of the modules of the PM may be adapted to fit a different operating system or a different windowing platform. These modules are described in the following.

PreloadingControl. This module is responsible for detecting the application launching and, once it occurs, call the initialization method included in the *Logic* module. Since this module might use non-cross-platform methods, it is probably that it has to be extended depending on the operating system.

EventConsumer. It captures the events generated as per user interaction with the target application. This module properly manages the data included in these events, which is sent to the HT later on. This module is also responsible for installing the event filters for the capture process.

EventExecutor. This module executes the GUI events received from the HT. Each time a new event is received, it extracts relevant data from the event and either posts a native event to the target GUI, or it executes an equivalent action to perform the requested behavior (e.g., posting a mouse-click event vs. executing a method implementing a click on a GUI widget).

At this point we can give an answer to research question **Q1**. The OHT architecture uses final modules to include all the generic functionality of the testing process (e.g., tester interface, management of recording and playback processes, synchronization of the PM and the HT). Then, adaptable modules are used in the "boundaries" of the architecture to add the functionality depending on specific technologies (i.e., GUI platform or operating system) or specific format (i.e., the test suite representation). As a result, much of the framework functionality is reused regardless of the specific execution context in use.

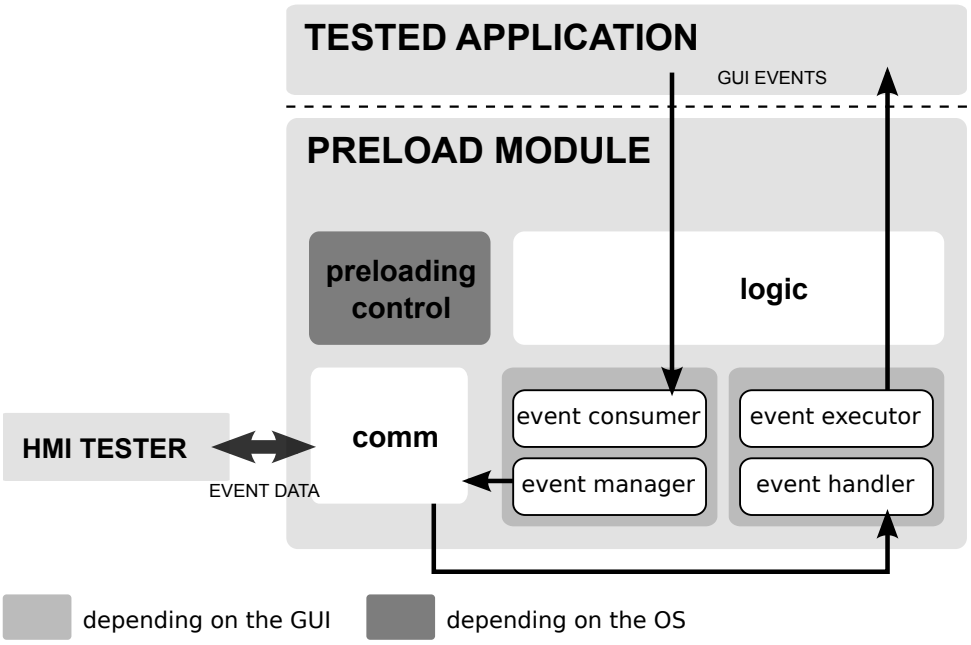


Figure 3.3 Preload Module detailed architecture.

3.4.3 The Event Capture Process

The event capture is the process by which the HT gets the events generated by the target application during the user interaction. In order to provide a non-invasive process, the HT uses the PM, which is preloaded and deployed into the target GUI to record the events when a test case is being recorded. The event capture process can be summarized in the following steps:

1. **Event generation**: while the tester interacts with the GUI of the target application (e.g., clicking a button, pressing a key) several GUI and non-GUI events are generated. The data included in these events has to be captured and sent to the HT.
2. **Event capture and Control signaling**: the PM gets the events generated in step 1, extracts their relevant data and encapsulates it on new test items. Then, the test items are sent to the HT. During this stage, control signaling is used to notify the HT about the target application state (e.g., the application was closed).
3. **Event handling**: the HT get the test items sent by the PM. In the case it is a control event, an action will be executed according to the notification. Otherwise, the test item will be stored in the current test case.

4. **Event storing and Control events handling**: control events are properly handled (e.g., when it is notified that the target application was closed, the current test case is finished and stored into the current test suite). GUI events are stored into the current test case respecting its order of arrival.

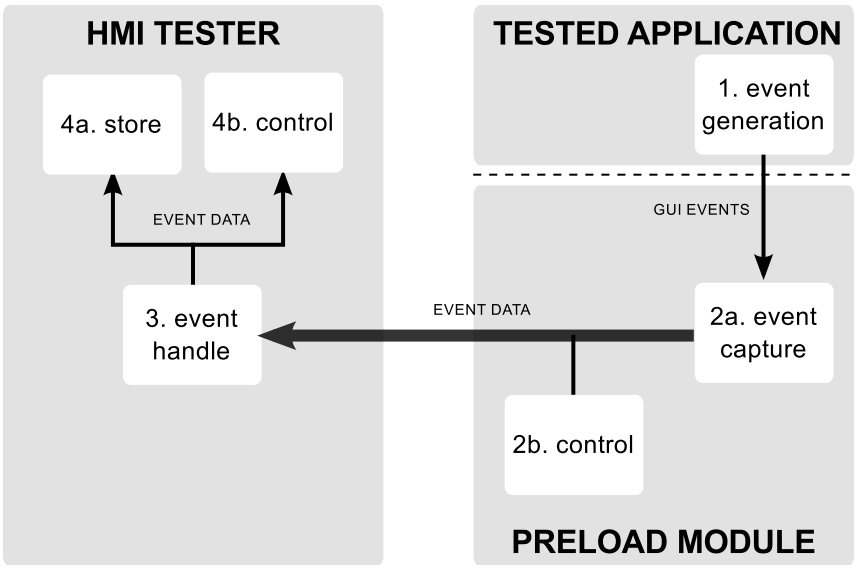


Figure 3.4 Diagram of the event capture process.

3.4.4 The Event Playback Process

The event playback is the process by which the HT sends the events stored into a test case to the PM for their execution. Control events are also sent in order to manage the process. When the PM receives new events, it will post them into the event system target application. These events describe the actions to be performed over the tested GUI. The event playback process is divided into the following steps:

1. **Event dispatching and Control signaling**: the test items (i.e., the events) stored in the current test case are sent to the PM. Control events are also sent to notify about the process state (e.g., the test case playback has finished) or about other actions to be taken (e.g., close the target application).
2. **Event pre-handling**: when a new event is received in the PM, its *type* value is used to decide whether the event is a GUI event, and therefore it has to be posted to the

target application, or it is a control event, and thus it has to be handled by the PM *Logic* module.

3. **Event posting and Control events handling**: the PM executes the received event (which represents an action in the GUI) either by posting the corresponding native event into the target GUI event system, or by executing an equivalent action (e.g. call the click method of a GUI button).
4. **Event handling**: the GUI events posted by the PM (or indirectly posted due to the execution of the equivalent action) arrive to the GUI event system and are executed.

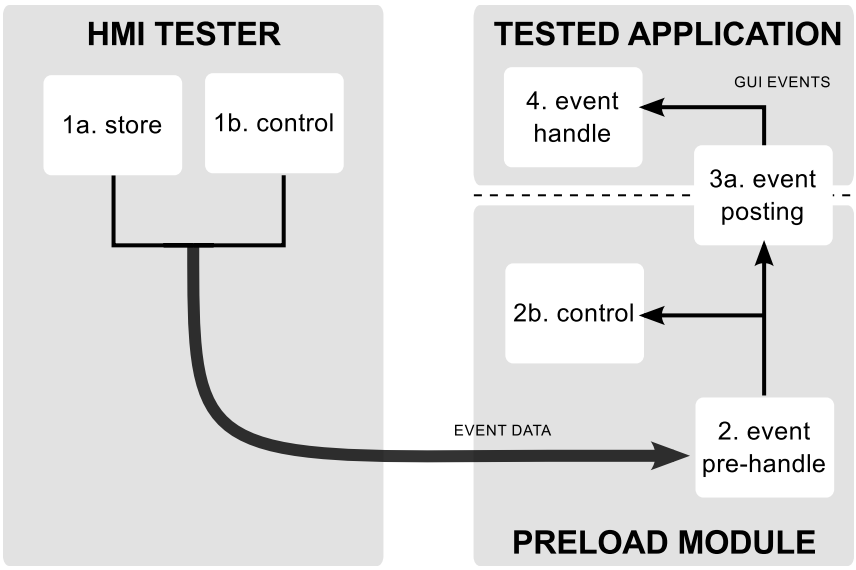


Figure 3.5 Diagram of the event playback process.

3.5 The OHT Implementation

As stated above, the OHT architecture is composed of several modules implementing generic and final functionality, and other modules that may be adapted to fit the OHT to different testing environments. This section describes first the implementation details of the generic functionality. Then, it is described how the proposed architecture should be adapted to a given windowing platform or operating system by adapting some modules. Finally, the implementation details of a fully-functional OHT-based capture and replay tool for the Qt GUI platform and Linux systems is described.

3.5.1 Implementation of Generic and Final Functionality

3.5.1.1 Generic Data Model

As described in Subsection 3.3.2, the OHT architecture uses test suites to arrange a set of test cases related to a target application. Such test suites are described by a generic data model, depicted in Figure 3.6. A test suite is linked to a single binary application and is composed of a set of test cases. Each test case represents a set of ordered test items (i.e., events) to be executed into the target application.

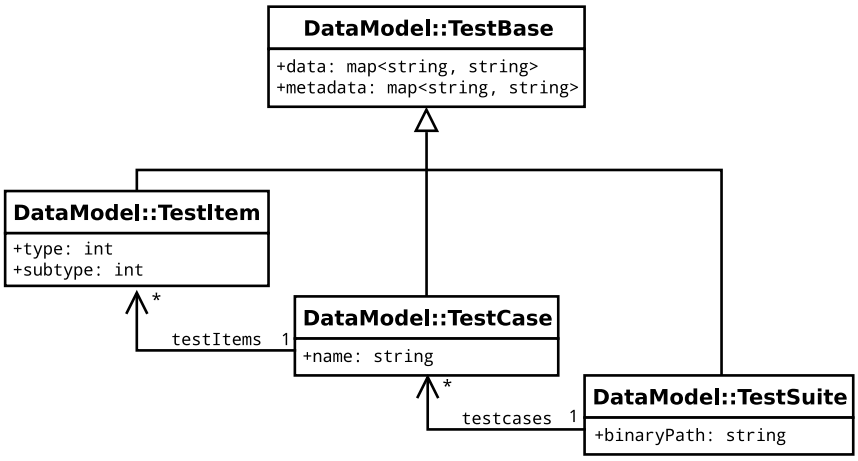


Figure 3.6 Generic Data Model Hierarchy.

Such a model design is flexible and generic enough to represent a set of test cases for any GUI platform. The `TestItem` object may be extended by both generic and specific modules to carry those data needed during the GUI testing process (e.g., control signals, user action events, etc.) The *DataModelAdapter* module will be responsible for translating a generic test suite object (and its internal objects) to a specific representation according to the project needs, as described in Subsection 3.5.2.1.

3.5.1.2 Generic Recording and Playback Processes

First of all, the OHT architecture needs a way to synchronize the interaction between the HT and the PM during the recording and playback processes. For this purpose, it is proposed the use of a control events hierarchy that is based on the `TestItem` object defined on the generic data model described above. A partial view of this event hierarchy is depicted in Figure 3.7. All the control signaling events are defined as derived of a generic `ControlTestItem`.

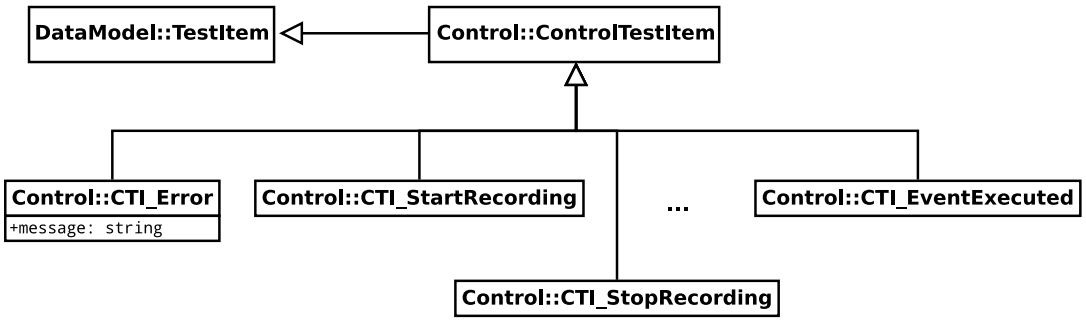


Figure 3.7 Control Signaling Events Hierarchy.

During the recording process, the GUI events produced during the user interaction with the tested application are captured by the PM and stored by the HT. The HT uses the control events to signal, for example, the start of the recording process, a pause, or to notify that the process has to finish immediately. The PM uses control events to notify about the state of the application (e.g., the application crashed).

During the playback process, the HT sends to the PM sequences of events (which were captured in a previous recording process) for their execution in the target application. These events are injected into the application as if a real human tester would be using it. Again, the HT uses control events to control the recording process (e.g., to notify that it is going to start sending new test items). The PM executes the events described by each of the test items, and then answers with a `CTI_EventExecuted` control event to synchronize the process in both sides.

3.5.2 Implementation of Specific and Adaptable Functionality

As stated above, a few modules of the OHT architecture may be adapted to suit the specific features of different testing environments. The adaptation encompasses implementing just the specific behavior required to interact with that particular environment. By leveraging the common architecture, the adaptation modules allow the OHT architecture to be flexible enough to support a wide range of existing operating systems and windowing platforms, as well as different test suite representations.

How these modules are adapted and *plugged* into the OHT architecture with this purpose is described in the following. Additionally, it is provided a detailed description of the

particular adaptations used to create a final version of the Open HMI Tester for Linux systems and Qt-based applications. More details related to this implementation are described next in Subsection 3.5.3.

3.5.2.1 Using the *DataModelAdapter*

One of the requisites of the OHT architecture was to allow the developers to use their own representation of the test cases (see Section 3.2). In accordance with this purpose the OHT provides the *DataModelAdapter*, a module that serves as a bridge between the generic description of test suites that is used internally in the OHT (described in Subsection 3.5.1.1) and the custom representation used by the developers. This adapter allows the *DataModelManager* to manage the information in the test suites in a uniform and generic way.

The developers have to provide in this module the functionality to translate all the data within a test suite in their custom format into a test suite in the generic format, and vice versa. In this manner, different representations of a test suite (e.g., stored in a local script file, in a XML file in a remote web server, or even using a URL) can be seamlessly integrated and properly managed into the OHT architecture.

For the final version of the Open HMI Tester described in Subsection 3.5.3, XML files were used to describe the test suites. With this aim, two methods had to be implemented in the new class `XMLDataModelAdapter`. The method `file2testSuite()` is used to create a generic test suite object from a given file path. It uses a DOM Parser to read the XML files describing the test suites. The method `testSuite2file()` creates a XML file from a generic test suite object. It uses XML visitors to extract the data from the test suite objects and return the corresponding XML string.

3.5.2.2 The Preloading Process

One of the main features of the OHT architecture is using library preloading to transparently hook the *Preload Module* (PM) into the binary application to be tested. This process is divided into two stages:

1. The `preload` option has to be enabled in the operating system. The HT has to indicate that the library in which the PM is encapsulated has to be loaded along with the target application.

2. The start of the target application has to be detected anyhow by the PM to start deploying the capture and replay services.

During the first stage, the *PreloadingActionModule* in the HT is in charge of configuring the preload action. This functionality depends on the operating system. Since the final OHT implementation provided as proof of concept is intended to run in Linux systems, then it was used the `LD_PRELOAD` environment variable for this purpose. This variable can be used in most UNIX systems to modify the normal behavior of the dynamic linker.

During the second stage, the *PreloadingControlModule* in the PM has to detect the target application start. For this purpose, the provided implementation includes in this module an overridden version of the method. This method is called during application startup in X11 based systems. The PM uses this method to start its deployment into the target application.

3.5.2.3 Adapting the GUI Event Recording and Playback Processes

First of all, we need to represent the GUI events that are sent between the HT and the PM during the event capture and playback processes. Since most of event-based GUI platforms like Qt [174], GTK+ [208] or JavaFX [111] arrange GUI events by using a hierarchy, a similar but simpler and more generic event hierarchy was created to be internally used within the OHT architecture.

The OHT uses the generic GUI events hierarchy depicted (partially) in Figure 3.8. This event hierarchy is based on the `TestItem` object included in the generic data model described in Subsection 3.5.1.1, and it is structured into three levels. Level 0 represents the base of the hierarchy. Level 1 includes different event sources in a GUI-based application. Level 2 includes all the different events that may be capture/executed in a GUI.

During the recording process, the native GUI events are captured by the *EventConsumer* module in the PM, and then translated into events of the generic hierarchy. The capture of native events has to be implemented depending on the windowing platform in use. For example, developers can install event filters in platforms like Qt and GTK+, or they can use event listeners in platforms like JavaFX. For the final implementation of the OHT, that is intended to test Qt-based applications, an event filter is installed into the main GUI events loop during the preloading process.

During the playback process, the HT sends generic GUI events to the PM, which executes them by means of the *EventExecutor* module. This module first translates the generic events

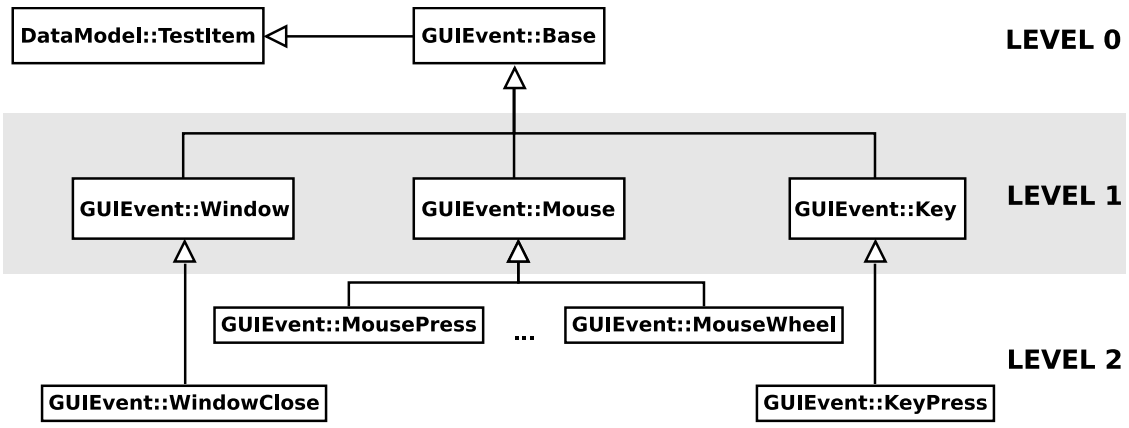


Figure 3.8 Generic GUI events hierarchy used in the OHT architecture.

into native ones, and then post them into the GUI events system. Again, to get access to the native GUI events the process have to be implemented depending on the windowing platform in use. In the final implementation of the OHT the native events are created using the data included into the generic ones. Then they are directly posted to the main Qt event loop. Some functionality is directly simulated over the target application instead of posting the native events to provide a better testing experience (e.g., to simulate mouse movement).

At this point we can give an answer to research question **Q2**. As described above, the OHT architecture is intended to inject GUI interaction events directly into the main event loop of the target application. It uses the adaptable modules to create native interaction events and post them into the native event system, in the same manner the use of mouse and keyboard devices is managed in most of event-based GUI platforms. As a result, the playback process simulates the interaction of a real human, improving the tolerance to changes in the GUI (e.g., resize or rearrange the GUI widgets) as further described in Subsection 3.6.4.

3.5.3 Technical Details About the OHT Implementation

A final implementation of the Open HMI Tester was created as proof of concept for the approach proposed in this chapter. As described above in Subsection 3.5.2, this final version of the OHT is intended to work in Linux operating systems and with applications based on the Qt4 [174] windowing platform. XML is used to store the test suites into text files.

Standard C++ was used along with the Qt system to provide a cross-platform tool that can be used in Linux, Windows and OS systems.

This version of the OHT is ready to use and to be incorporated into production. It is provided as a contribution to the open-source community and can be downloaded from [133]. It allows the testers to record test cases from the interaction with Qt-based applications. It includes the functionality of mouse and keyboard (e.g mouse click and double click, mouse wheel, key presses) as well as some window functionality such as close events.

The test cases are arranged into test suites, which are stored into XML files to be executed later. The tester can load a test suite file and choose among all the available test cases for the selected target application. The GUI events included into the test cases are executed as if a real human tester would be using the mouse and the keyboard. This is the reason why the OHT is tolerant to changes in the GUI design, to missing objects as well as to window moving and resizing actions during playback.

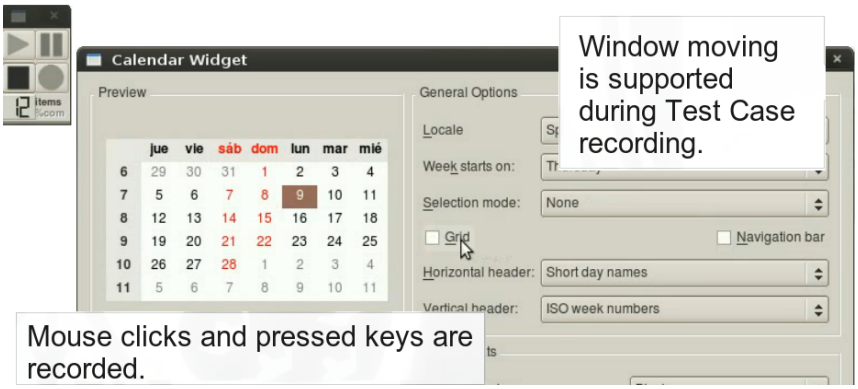
Figure 3.9 depicts how the final implementation of the OHT works with a calendar application extracted from [211]. (1) Depicts the OHT in *record* mode. Before recording a new test case, the tester has to provide a name to identify it in the current test suite. (2) During recording, the actions of the tester are recorded into the test case. The tester can also move and resize the windows participating in the test. (3) Depicts the OHT in *ready* mode. In this state, the tester can create a new test suite, add a new test case to an existing test suite, or select a previously recorded test case for its playback as shown in the figure. (4) Depicts the OHT in *playback* mode. The selected test case is executed automatically until it is finished or the tester (or an application crash) halts the process.

3.6 Discussion

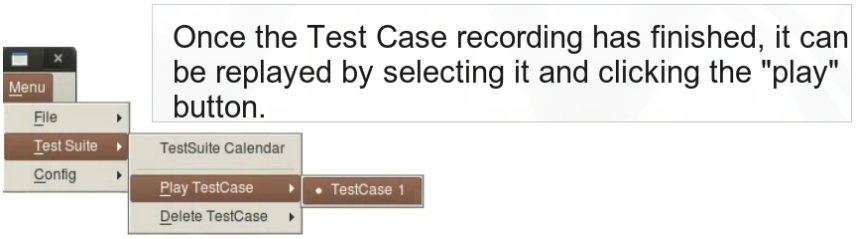
The Open HMI Tester (OHT) provides a framework to develop GUI testing tools based on runtime GUI introspection. It uses a capture and replay approach to generate test cases directly from user interaction and without building any model of the application. There are some examples in the literature using this approach. However, the OHT provides a cross-platform and open architecture that can be adapted to work in different operating systems or with different GUI platforms. Moreover, its functionality can be easily extended to

(1) The OHT before recording a test case.

(2) The OHT during recording.



(3) Selecting a test case for replay.



(4) The OHT performing the actions previously recorded.

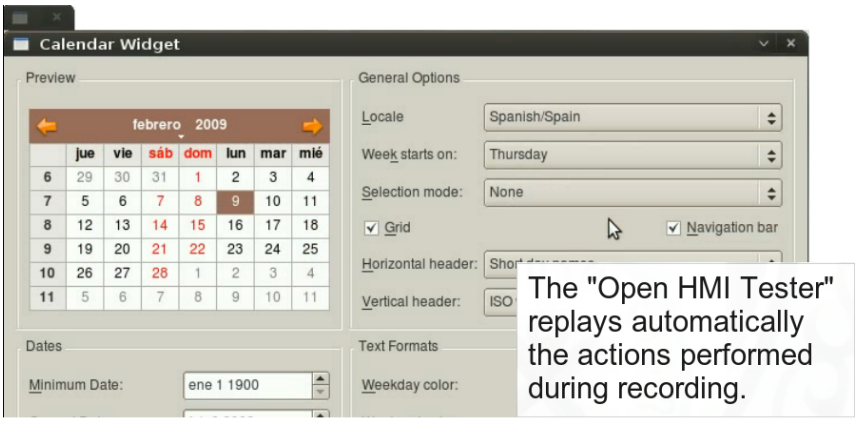


Figure 3.9 *Open HMI Tester* for Linux+Qt at work.

implement many different GUI testing functionality. The main contributions and limitations of the approach proposed in this chapter are discussed in the following.

3.6.1 Architecture

The OHT architecture describes a framework to automatically generate and execute test cases in GUI-based applications. The developers do not need to build or maintain any model of the target application. The test cases are automatically generated while tester interaction with the GUI application to be tested. They are then stored into a test suite for the later implementation of functional and regression testing.

The OHT is intended to serve as a basis for the development of GUI testing tools. It is fully portable since it is written using standard C++ and its definition is not linked to any operating system or GUI platform. The OHT is open-source. The developers can easily extend its implementation to support new windowing platforms, as well as to customize the testing process according to their needs (e.g., to capture only a subset of GUI events, to test performance by executing test cases repeatedly). As a result, the OHT architecture can be used regardless of the GUI platform or operating system used in the target application, as posed in research question **Q1**.

3.6.2 The Test Case Generation Process

Many related approaches implement automated test generation by searching all possible test cases in a model of the whole GUI [104, 155, 157, 223, 224, 228, 231] or a part of it [164, 215]. The OHT uses a capture and replay approach instead, in which the test case generation process is guided by the tester. It is the tester who, during the test case recording process, decides what widgets and actions are relevant for testing, thus determining the coverage criteria.

Such a process allows the tester to focus their efforts only on the relevant parts of the GUI, avoiding to build and maintain a whole and heavy model of it. In most cases this leads to a smaller set of generated test cases. This set may also grow incrementally in parallel with the development, thus reducing the number of modified test cases due to a change in the GUI design or functionality.

However, using a human-guided method may result in an incomplete testing process due to missing test cases. Some GUI functionality may remain out of the testing process.

Therefore, it might represent a risk that the whole responsibility of creating a complete test suite falls on the testers.

Moreover, some authors argue that the maintenance of capture/replay tests is expensive due to the GUI is constantly changing during the development [75]. The OHT tries to mitigate this problem by simulating the human actions directly in the core of the GUI system. Other authors argue that capture/replay methods postpone testing to the end of the development process when the GUI is already functional [182]. The OHT improves tolerance to changes in the GUI design (see Subsection 3.6.4), which reduces the number of obsolete test cases due to GUI design modifications.

3.6.3 Validation of Software Response

Some testing approaches use test oracles to validate the application output. Oracles include a reference to the results expected after the playback of a test case. Atif Memon introduced in [153] the usage of test oracles in GUI testing. He proposed using test oracles to validate the GUI step by step instead of validating the results once the test case playback has finished, because the final output may be correct but intermediate outputs might be incorrect. However, building test oracles for all the conditions to be validated in a GUI at runtime may result in a time-consuming and expensive process in terms of human resources.

The OHT architecture allows testers to use replay functionality to validate the application output by performing observation-based testing [122]. Visual verification can be used during test case playback to implement functional and regression testing. GUI convergence can be also checked. This might also represent a tedious process for testers. It might be dangerous as well if the whole validation process rests on the ability of human testers. To attempt to solve these problems, a semi-automated alternative for validating GUI output based on the idea described in [146] is proposed.

Thanks to the open nature of the OHT architecture, the test case playback process can be easily extended with further functionality to automatically check GUI properties at runtime. The events set used internally in the OHT can be extended with new meta-events called *verification points* to describe the expected state of some GUI properties at a specific time.

Such meta-events, which could be integrated into the test cases, e.g., by using an external editor, are then checked during playback. As a result, the observation process performed by

the testers is now supported by automatic checking of GUI properties. This represents a main concern for future work (see Section 3.7).

Regardless of how the application output is validated, the OHT architecture enables the implementation of the testing process directly over the target application at runtime. This allows testers to perform a more accurate and faithful testing process than if an intermediate model is used instead. Using a running instance of the target application to create and execute the test cases allows to test those "hidden" options configured within the application code or in the GUI design specification, which are only available at execution time.

3.6.4 Tolerance to Modifications, Robustness, and Scalability

During the test case generation process, new elements might be added to the GUI design. The tester can deal with this problem by creating a new test case involving the new GUI widgets, or by editing an existing test case to add the new actions to be executed. When GUI elements are removed, the tester can replace the out-dated test cases by new ones. The out-dated test cases may also be edited to remove the obsolete actions.

However, the testers might not take any of the proposed solutions. In this case, in order to avoid malfunction in the target application, the playback process in the OHT architecture foresees the missing GUI elements and do not perform the corresponding actions. This feature provides the OHT with high tolerance to changes in the design of the target application.

It is also necessary to provide tolerance to those actions of the tester or system features that may make the playback scenario different to the recording scenario (e.g., resizing the window of the application being tested, using a different screen resolution). In order to try to mitigate the side effects of these "external" actions, the OHT architecture follows the following two rules.

On the one hand, during recording, it is stored only essential data of the interaction process (e.g., the button in which to perform a click). Thus, other aspects of the GUI or the user interaction can be different during playback (e.g., the current position and size of the button). On the other hand, during playback, the user actions are replayed by posting events deep in the GUI (i.e., in the main event loop). This allows the OHT to simulate the interaction of a real mouse and keyboard devices as posed in research question **Q2**.

The OHT architecture is also designed for robustness. The whole lifecycle of a test case (i.e., record, storage, load and playback) is implemented within the architecture. Human intervention is reduced to the actions performed by the tester while recording a test case. The fact that the developers or testers do not have to build, verify or complete a GUI model increases robustness as well. However, many times in this chapter it has been mentioned that the test cases are open to be edited or added with new events or meta-events by using external tools. This might represent a source of inconsistencies and errors in the test cases. To avoid putting the robustness of the OHT at risk, the editing methods should be safe and trustworthy.

Scalability is another of the strengths of this architecture. During development, a GUI may be added with new elements, even with new sections or windows. To face GUI changes, the tester can (a) add new test cases, (b) replace out-dated test cases by new ones, or (c) update obsolete test cases by edition. In the worst case scenario, the number of test cases in a test suite will grow linearly with the GUI size. In other approaches in which test cases are automatically generated from a model, the number of test cases grows exponentially when new elements are added to the GUI. As conclusion, the testers will be responsible for the final amount of test cases in a test suite.

3.6.5 Performance Analysis

In order to evaluate the OHT architecture, the implementation described in Subsection 3.5.3 was used with some of the Qt-based desktop applications available at [211]. A set of applications including a GUI with many of the basic widgets (e.g., buttons, line edits, check boxes) as well as other more complex ones (e.g. calendar widgets) were selected for testing.

In general, the evaluation of the proposed architecture presented promising results. The OHT was able to capture the interaction in the binary applications used in the experiment, creating as a result one test suite for each application. The test cases were executed later simulating the previously recorded keyboard and mouse events with accuracy. There were some problems when simulating some advanced actions performed by the tester, which are described at the end of this section. Since no models nor representations of the GUI are used, then the overhead introduced by additional test case or oracle generation processes was nonexistent.

The recording process implies capturing the native GUI events, their transformation to the internal test item format, sending them to the *HMI Tester* and store the items into the current test case. During this process, no symptoms of time delay were observed. The response of the target application was as usual. Note that the process is carried out between one action of the tester and the next one, thus the observed delay is negligible.

The playback process implies reading the events from a test case, send them to the *Preload Module*, translate the events to the native format and finally execute them. The process was executed fluently, specially thanks to the idle time introduced between every two GUI events to faithfully simulate the interaction of a real user. Note that the set of events captured during recording (i.e., mouse, keyboard and some window events) do not even represent, in most of cases, the third of the events generated in a GUI.

Nonetheless, the performance of the OHT might be compromised if it is not used an efficient implementation of event capture and playback processes. The developers should provide efficient event filtering during capture process to avoid bottlenecks. It is essential discarding dispensable events such as timing or other non-GUI events. They should also provide a faithful replay process. With faithful we do not mean to replay executions with the highest fidelity, specially considering that capturing the features of the whole execution environment during recording is inconceivable. In most of cases, a small set of GUI events is enough to simulate testers actions faithfully and efficiently.

Regarding the fidelity of the playback process, some aspects of the simulation of the human tester are still unsupported. It is mainly due to the low maturity of the OHT architecture implementation. As mentioned above, there are some advanced interactions like drag and drop, as well as some advanced GUI elements like widget tables and modal windows that, in order to be supported, need further implementation in both the capture and execution modules. The future implementation of such features should not comprise the current efficiency of the testing process.

3.7 Conclusions

The GUI represents the mean by which the users interact with the application in most of cases nowadays. It is also the mean used by testers to check if the design and the functionality of an application are the expected.

Testing the GUI is essential before the final user accepts a software. Nevertheless, the testing stage requires important time and human resources. GUI testing requires also special tools for the automation of its processes. There is a lack of such a tools, specially in the open-source community. Moreover, this stage is particularly troublesome in the industry, where GUIs tend to be larger and more complex, and the execution environment often changes for every different project.

The OHT architecture presented in this chapter tries to mitigate these problems. It provides a cross platform, adaptable, and extensible framework that supports the automation of many different testing processes based on GUI introspection, the capture of user interaction, and the execution of actions in the tested application. It enables the implementation of robust testing processes that simulate the tester actions from the core of the GUI system, which also increases the tolerance to changes in the GUI design.

Its design has been driven by the requirements found in GUI-based industrial developments, and its implementation has been successfully integrated into some of them. The OHT is provided as an open-source and ready-to-use framework. It can be directly used within Qt-based applications and can be easily adapted to work in different testing environments.

4

Evaluating Quality of Users Input

Chapter 3 presents a solution to improve the quality of system output. Besides system output, users input is the other main component of human-computer interaction. Validating the input data provided by the users is essential to assure the quality of an application, enhance its robustness, and prevent malfunction.

Runtime Verification (RV) provides essential mechanisms to implement data verification. However, RV often entails complex and formal processes that might be inadequate in scenarios in which only invariants or simple safety properties are verified. This is particularly frequent when verifying users data in GUIs.

This chapter describes S-DAVER, a lightweight framework to implement separate data verification that can be easily integrated into a GUI. It decouples the whole verification logic from the business logic, and includes the verification rules in separate files. The rules are written in interpreted languages and can be changed at runtime without recompilation. Visual feedback is used to assist developers during testing, and to improve the experience of users during execution.

4.1 Introduction and Motivation

Software verification and validation processes (V&V) are essential to provide integrity and robustness into an application. Runtime Verification (RV) is a special case of V&V in which the execution of a program is monitored to determine if it satisfies correctness properties [191]. The data is verified within the runtime context as soon as it is produced instead of using a static representation of it [26].

GUIs are a particular case in verification. A GUI is an aggregation of widgets that hold data introduced by users. These data have to meet a set of constraints and content guidelines in order to avoid unexpected results or application crashes. Applying V&V processes into a GUI is, therefore, essential to verify the data constraints in real-time. However, the properties to be verified in a GUI are usually simpler than in other scenarios in which RV is commonly applied (e.g., a security password in the welcome screen vs. race conditions in a multithreaded application).

Applying RV into a project often involves the usage of complex languages and formal logic. These languages may present limitations in their expressiveness when writing complete specifications [119]. RV may also involve the formal specification of the acceptable runtime behavior of the application to be verified [46]. This process may impose too much overhead in a development, and is specially troublesome in those stages of the development in which the application design and behavior change frequently.

Moreover, many RV approaches use aspect-oriented programming (AOP) languages (e.g., [17, 28, 97]), which present disadvantages like code bloating and maintainability problems [23]. AOP languages lack a dynamic nature as well, as most of them need to be recompiled each time the rules are slightly changed.

For the reasons described above, applying RV into a GUI to verify users input data might not result in an efficient solution in terms of functionality, time, and development effort. It should be found, therefore, a more effective way to verify data in such scenarios instead of integrating advanced, formal, and thus more complex RV processes. Furthermore, the developers should be free to choose an appropriate specification language for data verification according to the project needs and its dimensions.

The main goal of the research in this chapter is to reduce the overhead introduced by common RV approaches into GUI data verification scenarios. It focuses on finding a lighter, more dynamic, and easier-to-integrate method. This new method will be based on

fundamental pillars of RV such as runtime operation, time efficiency, and encapsulation of the verification code. However, the fact that formalisms are not used in order to provide a more lightweight and effective method might comprise the correctness and completeness of the verification process.

It is also a goal in this chapter to improve the experience of developers and users while using a verification method. We would like to provide a dynamic verification process as well as to explore a unified way to provide interactive feedback to them.

Additionally, related to the problems and challenges described above, we would like to pose the following research questions:

> **Q1**: Can the verification logic code and the verification rules be completely pulled out of the business logic avoiding AOP languages?
> **Q2**: Can the verification rules be changed at runtime avoiding the recompilation of the whole software to provide a dynamic verification process?

Aiming to solve the aforementioned problems and to give an answer to these research questions, Section 4.2 presents a new approach for lightweight runtime data verification in GUIs. It describes a verification layer located between the GUI and the business logic, which uses rules written in scripting languages. The main features of this approach are described in Sections 4.3, 4.4 and 4.5

As a result of this research, the design of a new verification framework named S-DAVER (*Script-based DAta VERification*) is presented in Section 4.6. Some considerations about its implementation and its integration into developments are described next in Section 4.7. Section 4.8 describes a practical use case in which the implementation of S-DAVER was integrated into two FOSS (Free and Open-Source Software) applications. Section 4.9 includes an analysis of the performance of the proposed tool.

Finally, Section 4.10 discusses the main innovations of the proposed approach as well as its limitations. This section also compares S-DAVER to other related works previously described in Section 2.1.2. Section 4.11 draws the conclusions of this chapter.

4.2 Practical Analysis of Common GUI Data Verification Approaches

In our previous experience working with GUI-based applications, as well as collaborating with some software development companies as, e.g., SAES [202], two prevalent approaches to verify input data in GUIs were observed:

- **Approach 1**. The verification rules are tangled inside the business-logic code, normally within the widget-event handler methods (e.g., `onClick()` method). This approach considerably reduces the quality of the logic code. Moreover, recovering from a data violation is time-consuming. When invalid input is detected, the event handling process is aborted and the user has to be notified about the violation. Then, the user restarts the process by editing the field again.
- **Approach 2**. The rules are included into the GUI specification (e.g., `max` and `min value` limits of an integer spinbox widget). As an advantage, input values are verified before they are brought to the business logic, thus the event handling process is not started in case of data violation. However, the verification code is now scattered across the GUI specification, which pollutes the graphical design of the application.

These two approaches might not represent the best way to proceed in most scenarios with data verification into a GUI. They include the verification rules all over the source-code, even into the GUI specification, instead of having a separate place to store them properly. There exists not a common point in which the data is verified and from which data violations are notified to the user. Moreover, feedback to users is nonexistent, or at best, it is hardcoded along with the verification code.

To improve these limitations and to improve the final quality of the source code, the development and the whole verification experience, a verification method should include at least the three following features:

1. A **runtime monitor** that automatically and transparently checks the rules against the current GUI execution as response to the changes made by the user during the application use. The monitoring effort will be executed in conjunction with the underlying GUI system to minimize the verification overhead. Integrating this monitor into a development should be straightforward.

2. A clear and organized **set of verification rules** completely decoupled from the GUI and business logic. The developers should be free to choose the language to define the verification rules. Runtime changes in the verification rules should be allowed.
3. A **feedback system** in which all verification operations are registered and data violations are uniformly and interactively notified to developers and users.

The results of our research efforts, which aimed at giving an answer to these three main concerns, are described in the following.

4.3 Monitoring GUI Data at Runtime

A runtime monitor represents the core component of the verification process. This element is in charge of loading the verification rules, detecting changes in the GUI data and looking for the corresponding rules, checking them against the current execution of the program and, finally, providing the user with the appropriate feedback.

The runtime monitor supports the verification process almost in its entirety. Therefore, it has to be designed to incur in minimal overhead with respect to the overall performance of the GUI as well as of the whole application. To reduce this overhead, the proposed approach takes advantage of the idle time between the actions composing the interaction process between the user and the system in a GUI. The user performs a set of actions manually (e.g., mouse clicks, keystrokes) to which the system automatically responses with output (e.g., a sound, visual information). This idle time between the user actions is used for the verification process to check the rules, thus reducing the perceived overhead.

For this purpose, the runtime monitor is included as a lightweight verification layer located between the GUI and the business logic. The layer can intercept the events generated because of the user actions to detect changes in the GUI data. These events are analyzed and can be truncated in case of data violation. The verification process, that is depicted in Figure 4.1, involves three steps (GUI changes detection, rule finding and checking, and decision) that are further described below.

Step (1). The verification layer analyzes GUI events generated as per user interaction to determine if any change occurred in GUI data (e.g., the user introduced a new value into a widget). An event filter is used to detect these changes as well as additional information like the interacted element and the action performed on it. The filter provides the lightweight

layer with independence from the rest of the application because it is used to transparently keep a watch on the GUI events flow. As a result, each change in the widgets data triggers a checking process.

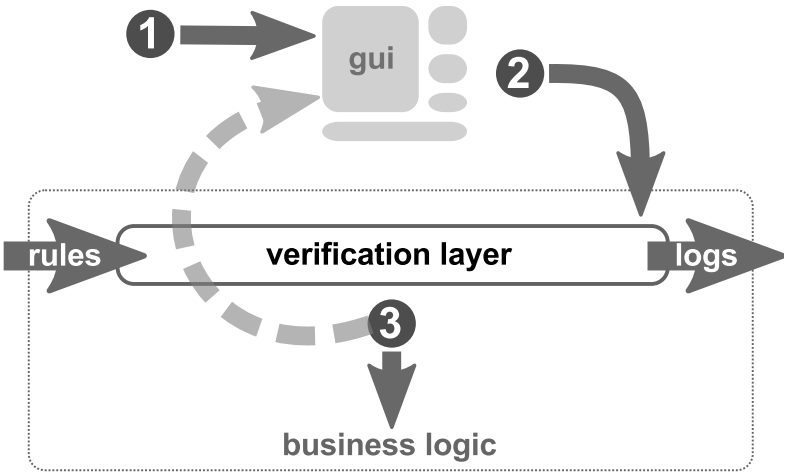


Figure 4.1 Overall behavior of the proposed V&V scenario.

Step (2). The verification layer looks for those rules involving the GUI data that were modified in *Step 1*. Once found, the rules are checked against the current state of the GUI (i.e., the current values of the data within the widgets) to ensure that the change fulfills the requirements. The interpreted code in the rules is directly executed by the layer using the data into the GUI widgets, therefore without involving any code of the business logic.

Step (3). Finally, the layer makes a decision depending on the results obtained from the checking process. If all of the verification rules were met, the event generated by the user action continues towards the business logic to be handled. Otherwise, the layer may truncate the event if configured to do so (see *stop events* in Subsection 4.7) and uses *GUI Intervention* to provide the user with dynamic feedback (see Subsection 4.5).

The approach is dynamic by definition: the verification rules are checked at runtime while the user is interacting with the GUI, similarly as how an invariant is checked in the *Design By Contract* approach [162]. Single-state checks are made to verify a set of safety properties (i.e., the data within the widgets) and thus to ensure that the GUI is in a consistent state. The data in the widgets is assumed as the state of the application at a specific time, and its correctness implies that the user behavior in the GUI is correct as well.

Figure 4.2 depicts an example showing how the lightweight framework works:

1. The user writes 25 within *Buffer Size*, which only supports integer values lower or equal to 20. A *FocusLost* event is generated just when the user leaves the widget. This event is intercepted by the verification layer.
2. Once the event is intercepted, the verification rules corresponding to the interacted widget are selected and then...
3. ...checked against the current execution of the GUI.
4. Finally, a new entry including the verification results is added into the log files. GUI intervention is used to decorate the widget with a red background color to indicate a non-allowed change. The data within the widget is changed to a default value to restore the GUI to a consistent state.

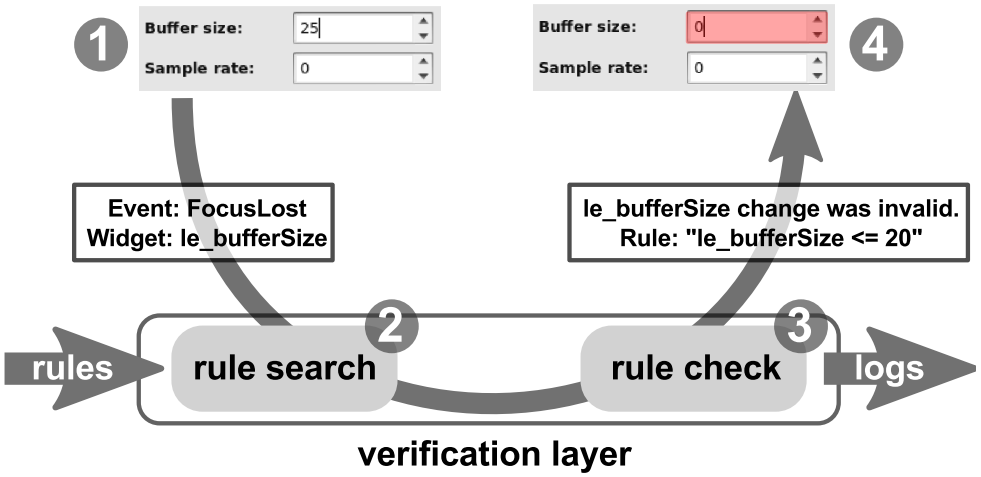


Figure 4.2 Wrong GUI input data verification example.

At this point we can give an answer to research question **Q1** posed at the beginning of this chapter. The verification layer is a single, lightweight, autonomous and independent object encapsulating all the verification logic code. It is "connected" to the application by intercepting and altering the GUI events flow. All the information for the verification process is extracted from these events. Then, AOP languages and other similar techniques are not needed. The verification rules are included in a separate location from which they are loaded (see Subsection 4.4). As a result, the verification logic is completely pulled out of the application business logic.

4.4 Verification Rules

The verification rules represent the requirements for input data. The requirements are usually specified in the Software Requirements Specification (SRS) to describe the constraints to be met by the data into the GUI widgets. They specify the properties to be checked as well as the expected values.

4.4.1 Rule Definition

The rules may involve data from one or more GUI widgets. They may also include guard conditions or complex operations like database queries and calls to external functions, e.g.:

```
if (checkBoxStarted.isSelected)
   var MAX_SPEED = getValue("max_speed")
   return spinBoxSpeed.value >= 0 && spinBoxSpeed.value < MAX_SPEED
```

Any rule specification should be allowed, provided that it is defined as a function returning a boolean value and regardless of how much code was executed to reach this result.

This approach proposes using interpreted and general purpose languages to define the verification rules. Interpreted languages are those in which the code is executed at runtime by an interpreter instead of being compiled first and then executed. Such languages are flexible and provide a high expressiveness to define potentially any rule. Furthermore, they allow the creation of lighter and more readable rule specifications. However, the main reason of choosing interpreted languages is that pieces of interpreted code are loaded and executed at runtime. They can be modified, added or removed during execution without recompiling and relaunching the application.

There are many of interpreted languages that can be chosen depending on developers needs. For example, restriction languages as OCL [218] provide a more precise and formal checking process, while scripting languages as Ruby [67] or JavaScript [187] provide more flexibility. In the following the reader can see an example of rules written in different general-purpose languages and notations such as OCL, Octave [59] and Lua [94]:

```
-- a rule in OCL
context Ui
inv: spinBoxSpeed->value >= 0
inv: spinBoxSpeed->value < 100
```

```
# a rule in GNU Octave
function result = simpleRule(speed)
   result = (speed>=0 & speed<100);
endfunction
```

```
-- a rule in Lua
function simpleRule()
   return spinBoxSpeed:value() >= 0 && spinBoxSpeed:value() < 100
end
```

4.4.2 Using the Rules to Apply Correction

Scripting languages also allow developers to include correction actions within the rules definition. This process, called *Runtime Enforcement* by some authors, is an extension of RV aiming to circumvent property violations [63]. In this manner the rules are not only used to check whether the GUI data meets the requirements, but they can be used also to correct the invalid data into the widgets.

Correction actions are used to bring the GUI to a consistent state before the input data reaches the business logic, as showed above in Figure 4.2 (step 4). The scope of the correction actions is limited to the GUI data. Thus these actions can not be used to modify the business logic or any application behavior out of the GUI. The following rule shows an example of GUI data correction. It always evaluates to true because, after its execution, `spinBoxSpeed` has always a valid value:

```
if (spinBoxSpeed.value > 100)
   spinBoxSpeed.value = 100
return true
```

4.4.3 Rule Arrangement

The rules should be stored in such a way as to avoid a lack of structure in the requirements. For this purpose, it is proposed using a set of separate files arranged using a well-structured file tree that matches the structure of the GUI. Figure 4.3 depicts an example of how the rule files are organized for a GUI composed of one main dialog (`MainWindow`) and two nested dialogs (`SaveDialog` and `FileDialog`). Each folder represents a dialog composing the GUI. A folder will contain zero, one or more rule files including the requirements related to the corresponding dialog. A folder can also contain subfolders, which represent nested dialogs.

Using this arrangement the rules are defined following the logical structure of the target GUI. It eases rule search in future debugging and maintenance processes. Furthermore, it

provides an implicit namespace to uniquely identify all the widgets compounding a GUI. For instance, in the example depicted in Figure 4.3, the *save_rules.lua* file could include a rule involving widgets from the current dialog `SaveDialog` (no namespace required) and from the parent dialog `MainWindow`:

```
1 fileName.value == MainWindow::projectName.value
```

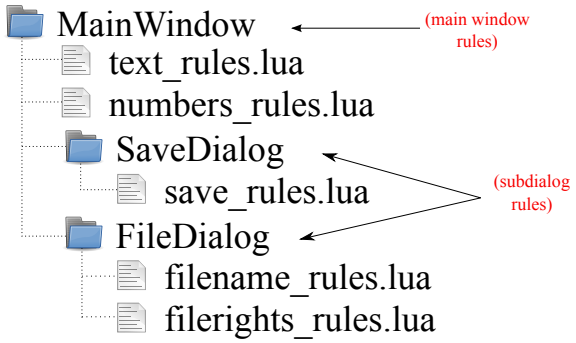


Figure 4.3 Example of the file structure proposed to arrange the rule files.

As a result, the code of the verification rules is completely detached from the application source code. Using a set of separate rule files helps developers to protect the source code against changes in the GUI data requirements. Thereby, changes in the requirements will affect only the corresponding rule files. Moreover, using interpreted languages allow us to make changes in the verification rules and reload their specification at runtime without recompiling the application source code. Changes will take effect immediately. Furthermore, existing rules can be completely removed at runtime, and new ones can be added on the fly as well. This answers research question **Q2**.

4.4.4 Rule Management

This subsection describes some considerations about the lightweight management of the verification rules. It is described how the rules are loaded into the target application, as well as how the verification layer handles their evolution during the application execution.

4.4.4.1 Loading the Rules

In order to provide an efficient verification process during the application execution, the rules are loaded and linked to the corresponding GUI widgets in application start. The

loading process is divided into the following three stages:

1. The directory/ies containing the specification is traversed to extract all the rules in the verification files.
2. In parallel, the dialogs composing the GUI are traversed as well to register the widgets into the verification layer.
3. After 1 and 2, GUI widgets are linked to the verification rules to denote what verification rules have to be checked when the data within a specific widget is changed.

The matching process between the rules and the widgets follows one simple rule: a widget is linked to only the rules in which its data appears. Therefore, when the data into a widget is changed, only the rules involving this widget are checked. This approach is scalable, as only a small group of rules are checked at a time. The performance of this process would be easily enhanced if the rules are checked only when the value of the interacted widget actually changes. It implies to keep up the current state of all the GUI widgets, which might comprise the scalability of the verification system.

4.4.4.2 Evolution of the Rules and the GUI

As said above, the verification rules are written using interpreted languages and included into external text files. This allows developers to change the rules specification at runtime. The rules code can be fine-tuned while the application is running without recompiling the software and relaunching it. New verification rules can be also dynamically attached/detached to the verification engine if a new dialog or window is added/removed to the GUI (e.g., when changing from the basic to a premium version of an application).

This feature is specially useful in testing stages. Take, for example, an industrial development including a GUI-based control panel. During the testing stage, the developers are checking if the constraints related to input data in the GUI are fulfilled. Suddenly, a mistake is found in the limits used by one rule verifying an integer spinbox widget. The rule is changed on-the-fly by modifying the rule code on the corresponding text file. The developers can see the effect of the introduced change immediately.

As the reader can see, the process is dynamic and direct. After a change, developers do not have to waste time recompiling or deploying the software. The reader can note that this process may take even several hours in complex industrial developments. Furthermore, the

application source code as well as the verification layer remain intact while the verification rules are changed or fine-tuned.

4.4.5 Correctness and Consistency of the Rules

The proposed approach uses scripting languages for the definition of the verification rules. These languages provide the developers with a high flexibility to define the rules and enable their modification on the fly. However, the semantics of scripting languages are normally more vague than formal languages. Furthermore, these languages usually do not include built-in mechanisms to check the correctness and consistency of their code. This may lead to correctness and consistency problems in the rules code.

The correctness of the rules is checked during the loading process. Rules including syntactic mistakes or unrecognized elements (e.g., an unrecognized external function or a nonexistent widget) are not included into the verification engine. This ensures that all the rules verify existing properties, thus they are safely checked even after a modification of the rules specification or the GUI.

However, checking the internal consistency of the rules is not straightforward, specially if the scripting language selected for the rules specification lacks a formal definition. In this situation, even to find an inconsistency like the existing between the two following rules (see constraints for value 0) would be fairly complicated:

```
1: spinbox_speed.value >= 0
2: spinbox_speed.value > 0 && spinbox_speed.value <= 100
```

Since the proposed approach ignores *a priori* the language used for the rules specification, then a method to check their internal consistency can not be provided. Therefore, developers should use an external tool or mechanism to carry out this task.

For example, they can use a framework for Constraint Logic Programming (CLP) like cKanren [8] or [11]. The verification rules are translated into a CLP specification. Each widget value is treated like a variable to be checked. As a result, inconsistencies in the rules can be found for each variable. Using CLP would be a valid solution. However, this contradicts the lightweight approach proposed in this chapter because creating a CLP specification from the verification rules is not straightforward. Finding an effective way to implement the process described above is one of the main goals as future work.

Moreover, in order to improve the robustness of the verification framework, the rules are checked as a set. As described above, when the data within a widget is modified, the rules linked to this widget are checked all together to determine the state of the widget. All the rules are checked to make a final decision and evaluate to true or false. If only one rule of the set fails, the widget remains in invalid state. In the example above, using the value `0` into the widget `spinbox_speed` would result in data violation because of rule `2`.

This provides a means to avoid states in which the GUI (and the entire application) may remain unusable due to an inconsistency in the verification rules. Furthermore, it provides also a way for the developers to "visually" check potential rules inconsistencies. The fact that a widget never reaches a valid state is a symptom of an inconsistency between rules.

The possibility of changing the rules specification at runtime might also be a source of correctness and consistency problems. Aiming to overcome this, each time a change is found into a file in the rules specification, all the existing rules linked to this file are removed from the verification engine and then reloaded following the preventive actions described above.

Finally, as said above, the rules are able to include correction actions in their code. These actions are directly applied to the GUI data in case of rule violation in order to bring the GUI to a consistent state. However, this feature might lead to consistency problems in the GUI if corrections are not applied consciously and carefully. The potential side effects produced by the correction actions implemented within the rules can not be known before its execution. Thus, these are responsibility of the developers and their code.

4.5 The Verification Feedback

As stated above, one of the main goals of this chapter is finding a dynamic and uniform way to provide the developers and users with all the relevant information generated during the verification process. This process should be completely automatic and transparent to developers. Therefore, we used the output data generated within the verification layer to automatically provide feedback through two different mechanisms: log streams and *GUI Intervention*.

On the one hand, an independent log stream exclusively related to the verification process is proposed. This log stream includes information related to the user interaction, the checking process results, incidences during the verification process, etc. The developers

can configure whether the log output is redirected to one or several text files, to other text stream or both (see an example in Section 4.8). Having all this information in a specific place (e.g., in separate logging files) eases debugging and auditing processes.

On the other hand it is proposed using GUI interventions, a more dynamic and attractive way to provide feedback about the verification process. The verification results are used to make visual changes in the GUI. The users are provided with "live" feedback superimposed directly over the interface with which they are interacting. Examples of these visual changes are, for example, highlighting invalid fields in red or showing unfulfilled rules in a floating box (e.g., see Figure 4.6).

GUI interventions notify the users about the checking results in real-time, helping them to notice input errors just in the moment the data are being introduced. It improves users experience while using the software as well as their efficiency. Using such interventions establishes a uniform way to provide dynamic feedback to users. Furthermore, the use of hardcoded painting routines along the business logic code is avoided.

4.6 S-DAVER Architecture Design

According to the lightweight verification approach described above, this section describes the architecture design of the S-DAVER framework. How some of its modules can be adapted to fit S-DAVER to a specific execution and verification environment (EVE) is described next.

4.6.1 Architecture Details

The proposed architecture is depicted as a UML diagram in Figure 4.4. As the reader can see, the S-DAVER architecture is composed of a set of modules revolving around the `VerificationEngine`. This central element works as the runtime monitor, analyzing the changes in the GUI data and checking the rules against them. On its left the reader can find those modules used to set up the verification process, load the rules specification and register the GUI elements to be validated into the engine. On its right the reader can see those modules in charge of bringing to light the verification results.

Figure 4.5 depicts how the different modules in Figure 4.4 collaborate to each other to implement the whole verification process at runtime. In this figure the reader can clearly

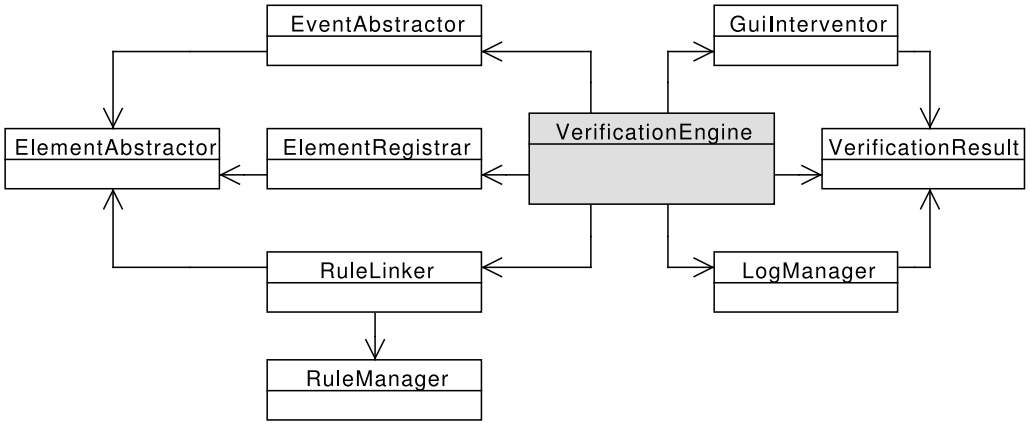


Figure 4.4 UML diagram of the S-DAVER architecture.

differentiate between two main processes. Steps 1 and 2 correspond to the verification layer initialization process. Steps 3 and 4 correspond to the runtime verification process.

In the first step the `ElementAbstractor` creates an abstract representation of the GUI widgets to be validated. The full name of a widget (i.e., including its namespace within the GUI) is used as its unique identifier during the whole verification process. Then, the `ElementRegistrar` uses the unique identifiers to provide the `VerificationEngine` with a link to the widgets to be validated, in order to make them accessible during the verification process.

In parallel, the `RuleManager` loads the verification rules from the specification directory. Then, once the unique widget identifiers are calculated and the rules are loaded, the `RuleLinker` performs a matching process between these two elements to link each rule to the data it validates (Figure 4.5, Step 2). The matching process is further described in Subsection 4.4.4.1. The resulting pairs are given to the `VerificationEngine`. All the information related to the rules and the widgets is kept updated at runtime to support changes in the rules as well as in the data sources.

Once the target application has been launched and the verification process configured, the runtime verification process starts. During user interaction, the `EventAbstractor` uses a filter to capture the GUI events. These events are checked whether they are in the list of events which trigger a checking process. If so, a new verification step [1] starts (Figure 4.5,

[1] A *verification step* is each one of the checking processes composing the whole verification process, and in which one or more verification rules are checked.

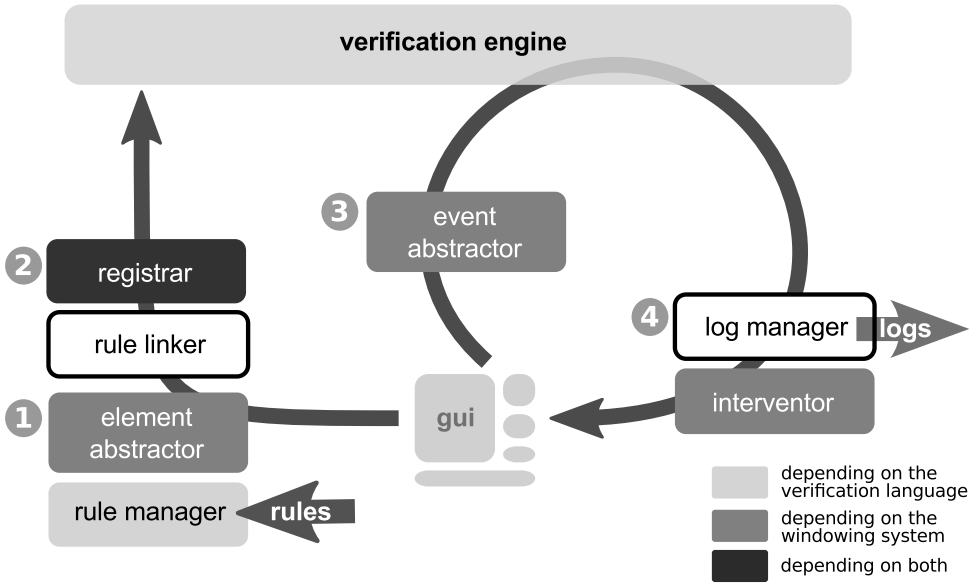


Figure 4.5 Module adaptations and interaction during the S-DAVER verification process.

step 3). The relevant information within the event (i.e., event type and destination widget) is sent to the `VerificationEngine`, which checks all the rules linked to the interacted widget against the current data it holds.

As a result, a `VerificationResult` object is built. It includes information related to the trigger event, the interacted widget, the rules that were fulfilled and those that were not. This object is given to the `LogManager` and to the `GUIInterventor` (Figure 4.5, step 4). The `LogManager` manages the log streams. The verification result is added to the logs according to the log configuration provided by the developers (see an example in Section 4.8). The `GUIInterventor` performs the GUI intervention process according to the checking results, e.g., turning into a red color the widgets involved in a rule violation.

4.6.2 Architecture Adaptation

As said above, some of the modules of the architecture can be adapted if the developers need to integrate S-DAVER into a different EVE. Figure 4.5 depicts those modules that should be adapted in case the developers want to change the GUI platform (*Case A*) or the verification language (*Case B*). These modules provide interface methods than can be implemented for this purpose.

Case A involves the modules that work directly on the GUI, i.e., `ElementAbstractor`, `EventAbstractor` and `GUIInterventor`. These modules have to be extended according to the GUI system in use, e.g., Qt [174], GTK+ [208]. *Case B* involves those modules depending on the verification language: `RuleManager` and `VerificationEngine`. They have to be extended according to the language in use, e.g., Lua [94], Ruby [67]. The `ElementRegistrar` depends on both the GUI and the verification language because it provides the `VerificationEngine` with access to the properties of the GUI widgets. Therefore, it has to be extended in the two cases described above.

Note that the core functionality of the framework is common to any EVE. Furthermore, the `RuleLinker` and `LogManager` modules are generic as well, thus they do not have to be adapted ever. As result, approximately 75% of the framework code is independent of the GUI system and the verification language.

The complexity of the framework adaptations directly depends on what is new in the target EVE, and which are the verification needs. Take, for example, a new development in which the GUI system changes. The developers are extending the `ElementRegistrar` to provide the `VerificationEngine` with access to the properties of the new GUI widgets. If, for example, the widgets do not have to be modified during verification (i.e., correction actions are not needed) only "getter" methods should be linked. If, for example, the rules will only verify the data within the widgets and not other properties such us their size, if they are enabled or not, etc., the adaptation is easily implemented by using a generic "getValue" method for each widget type.

4.7 S-DAVER Implementation and Integration Considerations

The S-DAVER framework was implemented and released as a contribution to the open-source community. It can be downloaded from [152]. The source code is distributed into a packaged file that can be built for different operating systems. It can be easily modified by developers that need to adapt the S-DAVER framework to their developments.

This version of the framework was implemented according to the execution and verification environment (EVE) described in the following. Standard C++ was used to implement the verification layer mainly due its performance for real-time systems. The Qt4 toolkit [174] was selected as the GUI system due to its cross-platform nature. Finally, Lua [94] was chosen

as the verification language. Lua is a scripting language widely used in the video game industry and in many commercial applications. Luabind [2] is used to create the bindings between Lua and C++. This version of S-DAVER was successfully compiled and executed in many different Linux distributions, including Ubuntu, Fedora and CentOS.

Section 4.6 described an open and adaptable design of S-DAVER, which can potentially be integrated into any development. The proposed design is agnostic to concepts such as the verification language, the GUI system and the event sources. Therefore, developers are free to adapt the original implementation of S-DAVER in order to fit the EVE of the development in which it has to be deployed.

As a proof of concept, a second version of S-DAVER was implemented using the ChaiScript [100] language. ChaiScript is an open-source BSD licensed scripting engine that can be easily embedded in C++ applications. In this new implementation (also available at [152]) only the modules including functionality related to the rules (load, register and execution) were adapted. The rest of the code was completely reused.

Beyond providing an open framework design and implementation of S-DAVER, this work was concerned with providing a lightweight integration and development process. By lightweight we mean a low code-intrusive method that is integrated into a development by using a few lines of code. Code intrusiveness means to have any verification code into the application code. By lightweight we also mean a transparent process to let developers to focus only on defining the rules. Finally, a lightweight framework should be integrated very early in the development, even when no code has already been written and no GUI design is available.

Based on these requirements, an integration process divided into the following three stages is proposed:

1. ***Adaptation to the environment (optional):*** The implementation of S-DAVER is adapted to fit the EVE of the development as described at the end of Section 4.6. In most cases, just two or three modules have to be adjusted to meet EVE changes (e.g., to use a different verification language). The rest of the implementation is reused.
2. ***Configuration:*** S-DAVER is easily configured by using a `VerificationContext` object. This object (further explained below) encapsulates all the configuration options for the verification process. The developers will create the `VerificationContext` object first, which has to be given to the `VerificationEngine` before deploying.

3. ***Deployment:*** During deployment, the verification layer uses the `VerificationContext` object to automatically configure itself. Then, it loads the rules and starts the run-time monitor. The whole process is performed by calling the `init()` method of the `VerificationEngine` object.

The `VerificationContext` object includes all the data needed to execute the verification process, e.g., rule files directories, the format of the log entries, etc. Table 4.1 shows all the options that can be configured using the current implementation of this object. The `VerificationContext` object can be instantiated from a configuration file or can be directly created in the source code (see example in Section 4.8).

Configuration option	Effect
`addRuleSource`	Adds a new directory or file to the rules sources
`removeRuleSource`	Removes a file or directory from the rules sources
`triggerStopEvents` `triggerWatchEvents`	Denotes events triggering a new verification step: - *Stop events* are truncated in case of rule violation - *Watch events* are never truncated, just notified
`updateRuleTime`	Selects how often the rules are checked for updates
`logFormat`	Sets the output log format
`addLogOutputFile`	Sets an output file for the log stream
`addLogOutputStream`	Sets an output stream for the log stream
`interventionOnError`	Selects the GUI intervention option for error cases
`interventionOnSuccess`	Selects the GUI intervention action for success cases
`interventionOnErrorColor`	Selects a color for GUI decoration for error cases
`interventionOnSuccessColor`	Selects a color for GUI decoration for success cases

Table 4.1 Configuration options supported by the `VerificationContext` object.

It is worth saying that, besides other options, the developers use this object to configure the *Stop* and *Watch* events. Both events trigger a verification process, but only *Stop* events will not be handled in case the checked rules are not fulfilled. This helps developers to protect the business logic against wrong data. Moreover, GUI intervention are easily configured to perform the desired behavior both when a checking process is successful or not. Not configured options will always considered to have default values.

Once the `VerificationContext` object is created, it is just a matter of creating the `VerificationEngine` object, provide it with the configuration object and call the `init()` method. Then, the verification layer will be initialized and the verification service will start working. During initialization the rules are loaded, the elements to be validated are registered and the event filter is installed to start monitoring the user interaction within the GUI. A real example of the whole configuration and deployment process is described in Section 4.8.

As a result, providing the verification layer with the configuration data, as well as switching among different configurations, is really easy by using the `VerificationContext` object. A couple of lines of code are enough to integrate and deploy S-DAVER into a development. The rest of the process (loading the rules, monitoring the GUI data and performing the verification processes) is addressed transparently and automatically. This reduces code intrusiveness to the minimum. Regarding the adaptation of the framework modules, the reader can see that the effort is greater the first time the framework is integrated into a new and unsupported EVE. However, this development effort is reused and amortized in future developments.

4.8 Practical Use Cases

To show the validity of the lightweight approach proposed in this chapter, the implementation of S-DAVER was integrated into two Qt-based FOSS applications. On the one hand we used Qt Bitcoin Trader[2], a popular open-source trading software that helps users to open and cancel Bitcoin orders very fast. On the other hand we used Transmission[3], a cross-platform open source BitTorrent client included by default into many Linux distributions, including Ubuntu.

This section first describes how S-DAVER is deployed into a development. Then, it is explained how and where the verification rules have to be defined using as example the two applications mentioned above. As a result of the implementation of these practical scenarios, Section 4.8.4 includes some considerations about the enhancements provided by S-DAVER during the development, testing and usage stages. Finally, the overall performance of this solution is evaluated in Section 4.9.

[2]Qt Bitcoin Trader: http://sourceforge.net/projects/bitcointrader
[3]Transmission: http://www.transmissionbt.com

4.8.1 Integration, Configuration, and Deployment of S-DAVER

As said in Section 4.7, the configuration of S-DAVER is based on the `VerificationContext` object. This object is indispensable for the creation of the `VerificationEngine`. Once these two objects are created and the rules source is provided, it is just a matter of calling the `init()` method to initialize the layer and start the verification service. The following code includes the four lines needed to implement this process into any Qt/C++ application. The verification layer has to be deployed on application initialization.

```
1  // create the verification context
2  VerificationContextPtr vc(new VerificationContext());
3  // add rule files or directory / set update time
4  vc->addRuleSource("rules/").updateRuleTime(10);
5
6  // (add extra configuration here)
7
8  // create and launch the verification framework
9  QtLua_VerificationLayer* vl = new QtLua_VerificationLayer(vc);
10 vl->init();
```

Additionally, the developers can set other options to configure the verification process to their liking. The next code snippet shows an example. *Stop* and *Watch* events are configured to set when the rules are going to be verified and if some GUI events have to be truncated (lines 2-3). The log output format and the output streams are configured to have all the verification information at a known place (lines 6-7). Finally, GUI interventions are configured to perform the desired behavior when a checking process is successful and when it is not (lines 10-12). Not configured options are always considered to have default values.

```
1  // configure trigger events
2  vc->triggerStopEvents(csu::vvl::framework::VVL_EVENT_MOUSECLICK)
3      .triggerWatchEvents(csu::vvl::framework::VVL_EVENT_FOCUSOUT);
4
5  // configure log stream
6  vc->addLogOutputStream(std::cout).addLogOutputFile("/tmp/verification.log")
7      .logFormat("[%tm] %wi at %en: %vr :: %fn");
8
9  // configure GUI intervention actions
10 vc->interventionOnError(VVL_GUI_HIGHLIGHT_ERROR
11         | VVL_GUI_HIGHLIGHT_RELATED | VVL_GUI_SHOW_ERROR_RULES)
```

```
12      .interventionOnSuccess(VVL_GUI_CLEAR);
```

4.8.2 Defining the Rules in Qt Bitcoin Trader

Qt Bitcoin Trader is an open source application developed in pure Qt. It helps users to open and cancel Bitcoin orders very fast with different traders and includes real-time data monitoring. This example focuses on the verification of the initial dialog. When the user enters to this application, he/she faces to the dialog depicted in Figure 4.6. The application requires the user to introduce an API key and secret, which is protected by a password using AES 256 encryption. All the data in this dialog must conform to a specific set of constraints, some of them involving more than one widget. All these constraints, which are directly implemented in the dialog code, have to be met before continuing by clicking the *OK* button.

Once deployed S-DAVER, the rules defined within the rules directory will be automatically incorporated into the verification engine. A new file `passwords.lua` was created in the directory `QtBitcoinTrader_Lastest/rules/NewPasswordDialog`. Since we are verifying an independent dialog that is shown at application start, the `NewPasswordDialog` folder is included in the first level into the rules directory. This file includes all the Lua rules used to verify the data in the widgets. For example, the following rule checks that the *OK* button needs not empty *key* and *secret* values:

```
1  rule linesNotEmpty
2     involves = @okButton:isEnabled()
3     return @restSignLine:text() ~= "" and @restKeyLine:text() ~= ""
4        and (@restSignLine:isVisible() and @restSignLine:text() ~= ""
5        or not @restSignLine:isVisible())
6  end rule
```

The rules may be more complex. The following rule is used to verify the format of the *profile name* field. This name has to be composed by using any lowercase or uppercase letter, any digit and any character from a specific set:

```
1  rule profileNameFormat
2     involves = @okButton:isEnabled()
3     allowed_profchars = {'(',')','+',',','—','.',';','=','@','[',']','^','_'
          ,'`','{','}','~',' '}
4     for char in string.gmatch(@profileNameEdit:text(),".") do
```

```
      local found = false;
      if string.match(char,'%l') or string.match(char,'%u')
         or string.match(char,'%d') then found = true end
      for k,v in pairs(allowed_profchars) do
         if char == v then found = true end
      end
      if not found then return false end
   end
   return true
end rule
```

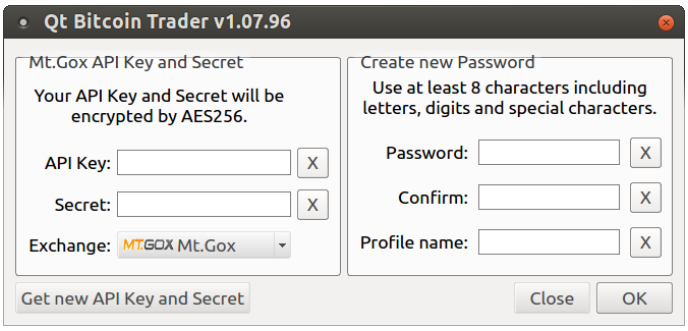
The results of integrating S-DAVER into Qt Bitcoin Trader are shown in Figure 4.6. This figure depicts the interaction sequence performed by the user to fill the dialog in (a). (b) depicts an error because *Secret* is empty. (c) depicts an error because *Confirm* is not equal to *Password*. (d) shows how new feedback information is interactively shown when the user places the mouse pointer over the wrong widget. All the widgets involved in the error are highlighted and the violated rule is shown in a floating blue box.

Methods like `isValidPassword())` or `checkToEnableButton()` included in the original application code have disappeared. After integrating S-DAVER the code is cleaner. For example, the following method will be called only if all the rules involving the *OK* button evaluate to true. Now, its code is free of data checks and painting routines:

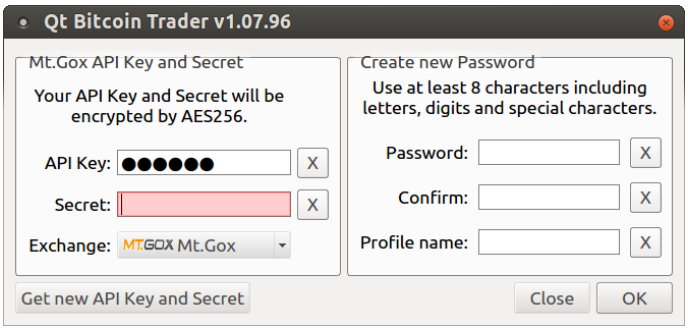
```
// old code
void NewPasswordDialog::okPressed(){
   if(isValidPassword())
      accept();
   else{
      QMessageBox::warning(this,"Qt Bitcoin Trader",
         julyTranslator.translateLabel("TR00100","Your password must be at
             least 8 characters and (...) special characters."));
      ui.confirmLabel->setStyleSheet("color: red;");
   }
}

// new code
void NewPasswordDialog::okPressed() {
   accept();
}
```

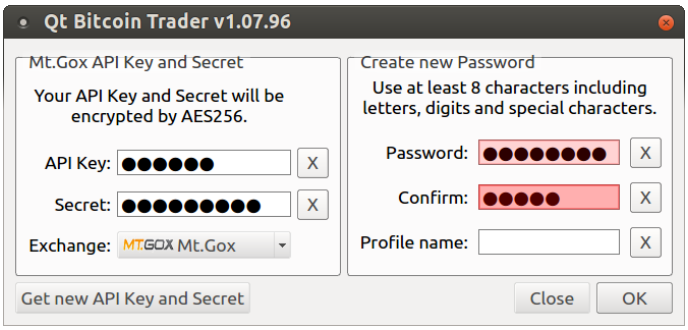
a) Initial password dialog in Qt Bitcoin Trader



b) Highlighting an error involving the data of a widget



c) Highlighting an error involving the data in two widgets



d) Showing verification feedback interactively on mouse hover

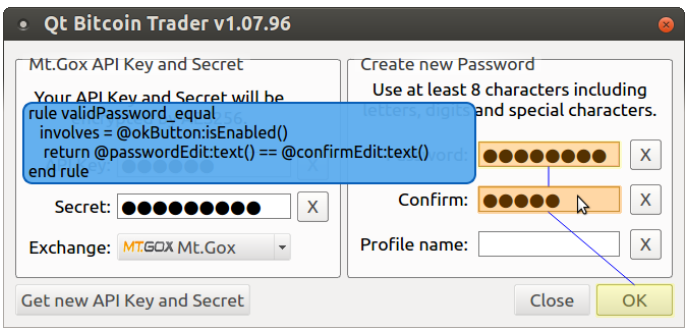


Figure 4.6 S-DAVER working within Qt Bitcoin Trader.

As a result, the Qt Bitcoin Trader application has now a uniform, more dynamic, more interactive and prettier feedback system. The original application code in this GUI dialog was reduced 37.8% (from 145 to 91 effective lines of code) resulting in lighter and cleaner methods.

4.8.3 Defining the Rules in Transmission

Transmission is an open source volunteer-based project. It was developed using standard C++ and Qt in order to provide a cross-platform solution. Transmission presents a simple GUI design that only requires few data from the users. Most of these data are included in the *Preferences* section, from which the user can configure the torrent sharing process and other features. Data constraints are, in general, simpler than those in the password dialog of Qt Bitcoin Trader.

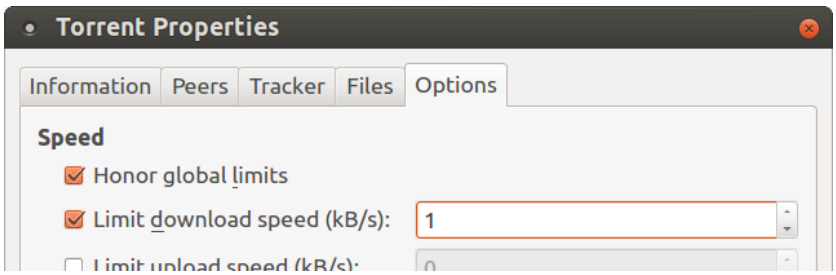
However, Transmission is not a good example of GUI design. Most of the GUI dialogs (except the main one) are hardcoded and created at runtime. Data constraints associated to the input widgets are included in the source code in the same place in which the GUI is defined (please, remember *Approach 2* described at the beginning of Section 4.2). This results in small pieces of verification code scattered in the whole application code, thus reducing the quality of the code and hindering development and debugging processes.

To mitigate this problem, S-DAVER was deployed into Transmission with the aim of bringing all the data constraints to a common place out of the application code, and thus try to lighten the original code. Due to the source code of Transmission is huge, and that S-DAVER works seamlessly with the application code, new rules were defined only for the two dialogs described in the following.

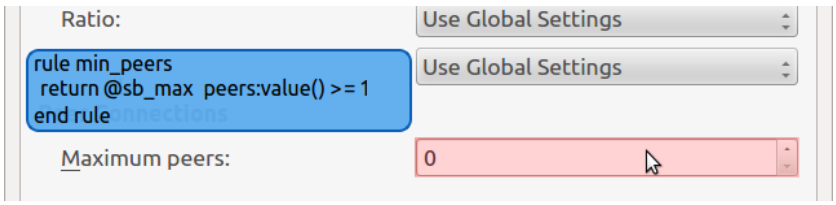
One of these dialogs is the `TorrentPropertiesDialog`, depicted in Figure 4.7. This dialog can be shown for each of the torrents that are being shared to other users. It is composed of several tabs showing information about the torrent. The last tab (the one showed in the figure) includes several options to customize the download/upload process for a specific torrent. The following example shows how the rules are defined in S-DAVER with and without correction.

Figure 4.7 shows two of the rules defined in the file `options.lua` included in the directory `transmission-2.82/rules/MainWindow/TorrentPropertiesDialog`. In the top of the figure the reader can see how the `download_speed` rule is used to bring the GUI to a consistent

state after an error. If the user sets a value lower than 1 for the value sb_download_limit, the rule directly applies correction and evaluates to true. This value never reaches a wrong state. Otherwise, the rule min_peers at the bottom of the figure does not apply correction actions. Instead, the rule checks the value sb_max_peers and, in case of an error, it evaluates to false and the error is interactively shown over the affected widget. This value reaches a wrong state, therefore it has to be fixed by the user manually.




```
rule download_speed
 if @sb_download_limit:value() < 1 then
    @sb_download_limit:setValue(1)
 end
 [...]
 return true
end rule
```



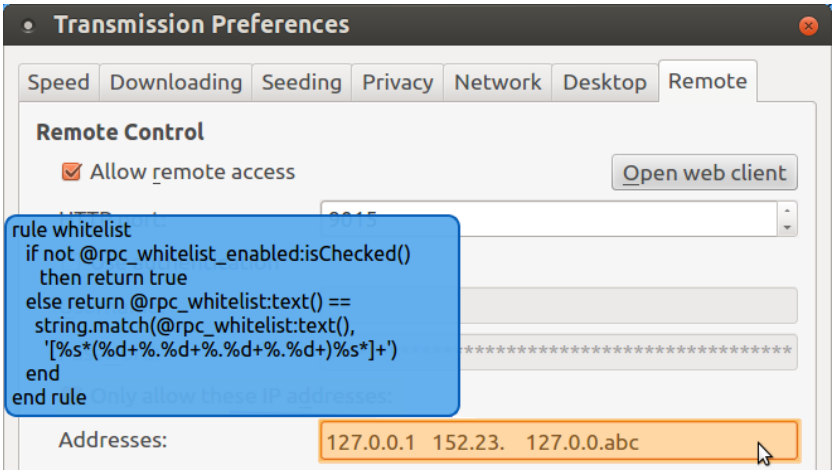

```
rule min_peers
 return @sb_max_peers:value() >= 1
end rule
```

Figure 4.7 Some of the rules validating the TorrentPropertiesDialog in Transmission.

Another dialog verified with S-DAVER is the PreferencesDialog depicted in Figure 4.8. This dialog is composed of a set of tabs in which the users can configure the torrent download/upload process, their privacy settings, etc. Here the reader can see another of the common verification mistakes in the implementation of Transmission. The content of

the text fields that are filled manually by the user is not verified until the data reaches the business logic (*Approach 1* described at the beginning of Section 4.2). Furthermore, Transmission does not provide any feedback in the case the user introduces wrong data.

Figure 4.8 depicts the `Remote` tab in the the `PreferencesDialog`. It includes the `rpc_whitelist` field, in which a set of valid IP addresses is expected. Originally, Transmission allowed to introduce any text in this field. Invalid or incomplete IP addresses, numbers, words, etc. were accepted without showing any warning at all. To face this problem, a new rule named `whitelist` was defined in the file `remote.lua`, that is included in the directory `transmission-2.82/rules/MainWindow/PreferencesDialog`. Now, the user is interactively notified if the input text does not match the regular expression included in the rule definition. Wrong data will not reach the business logic. Please, note that the rule in Figure 4.8 is a shortened version of the original one. Here, digits higher than 255 are allowed.




```
1 rule whitelist
2  if not @rpc_whitelist_check:isChecked() then
3   return true
4  else return @rpc_whitelist:text() ==
5   string.match(@rpc_whitelist:text(),
6    "[%s*(%d+%.%d+%.%d+%.%d+)%s*]+")
7  end
8 end rule
```

Figure 4.8 One of the rules validating the `PreferencesDialog` in Transmission (short version).

The three rules showed above are only some examples of the whole set of rules used to verify the data in `TorrentPropertiesDialog` and `PreferencesDialog`. The results in code reduction were not as good as with Qt Bitcoin Trader. In the best case, the code was reduced approximately 10%. However, the main benefits of applying S-DAVER into Transmission are to have a separate set of verification rules and to have a new interactive feedback system.

4.8.4 Development and Verification Experience with S-DAVER

The lightweight approach proposed in this chapter was concerned, from the very beginning, for improving the work of developers and the interaction of users with the application. Developers do not have to worry about the validity of input data while coding. The verification layer ensures that the data reaching the business logic is safe. Thus, the source code implements only the business logic functionality avoiding any verification statement. As a result, development is easier and code is cleaner.

Developers also benefit from changing the rules specification at runtime. Software testing becomes a more dynamic task, allowing developers to see the changes just when they are incorporated into the rule files. Feedback is also improved by using dynamic GUI interventions. Interactive help as shown in Figures 4.6, 4.7 and 4.8 provides extra information in real time. This allows developers to find errors in the requirements easily and fix them immediately.

The interactive feedback improves also users experience while using the software. Users can see immediately whether input errors are present, enabling a successful and more efficient interaction with the final application. This feedback can be used also to provide users with additional information, e.g., to suggest them the expected input format.

4.9 Performance Analysis of S-DAVER

Intensive tests were conducted to evaluate the performance of S-DAVER. Four different test configurations were created by combining different verification languages (Lua and Chaiscript) and the use or not of GUI intervention actions. The tests were run over a Qt-based GUI including 25 widgets guarded by rules. Each test comprised about 100 user interactions involving a verification step (VS), of which 70% evaluated to false. The verification steps where triggered by the *FocusOut* and *MouseRelease* events, which represent

Test setup	**Avg. number of VS / test**	100
	Avg. number of rules / VS	2.33
	Avg. number of widgets / VS	7.5
	% VS failed	70%

	Avg. time / VS (inc. GUI intervention)	**Avg. time / VS (exc. GUI interv.)**	**Avg. time / VS (inc. 70% GUI interv.)**
Qt+Lua	1.62 ms	0.32 ms	1.23 ms
Qt+Chaiscript	3.11 ms	1.40 ms	2.59 ms

Table 4.2 Test setup and performance results using Ubuntu 12.04, Intel Quad-core 2.83 GHz, 4GB RAM.

approximately 15% of scene events.[4] Additionally, two log outputs (the standard output and a file stream) were enabled during the test.

The time measured for each VS involved looking for the rules, checking them against the current data in the GUI and sending the results to the log streams. If GUI intervention was enabled, the time for highlighting the wrong widget is added, as well as the time for creating the decorations and the blue rule-box shown when mouse over. Out of the approximately 100 VSs performed for each test, 1/3 involved 3 rules and up to 5 widgets, 1/3 involved 1 rule and between 5 and 10 widgets, and 1/3 involved 3 rules and 10+ widgets.

Table 4.2 summarizes the results obtained from the tests. The reader can see that the difference between the time spent by the Lua and the ChaiScript engines is very remarkable. This shows that the performance of the verification language interpreter represents a key aspect when choosing a verification language for a project. Enabling GUI intervention has also a remarkable impact on the overall performance. This time overload might be reduced by using optimized painting routines.

Resulting from this analysis, 3 factors directly related to the performance of S-DAVER are identified: the number of widgets to verify, the average number of rules linked to a widget and the number of GUI interventions. The higher these values are, the higher is the average time spent for each VS. However, the proposed approach uses the time between one action and the next one during user interaction to perform the whole verification process. As a result, the user perceived overload is negligible.

[4] *Scene Events* are a subset of the GUI events set. This kind of events is directly related to user input methods like, e.g., mouse clicks, key presses, focus changes, etc.

4.10 Discussion

This chapter describes a lightweight verification framework specially oriented to GUI data. This kind of methods specially aimed at verifying user input data at runtime are not very common, as previously analyzed in Section 2.1.2. This work describes also the implementation of the S-DAVER framework and an usage example as proof of concept. An open-source implementation ready to be used is provided to developers. This implementation can be adapted to be used in many different verification scenarios. The main contributions and limitations of the approach proposed in this chapter are discussed in the following.

4.10.1 A Lightweight Data Verification Approach

S-DAVER describes a lightweight verification framework aimed at assuring the validity of user input data in GUIs. Its design is based on fundamental pillars of RV. GUI data is intended to be dynamically verified during user-system interaction, just at the moment and within the temporal context in which they are generated. However, there exist important differences with usual RV approaches, specially related to the use of formalisms to specify the rules. Furthermore, while RV is usually aimed at verifying the application behavior, the proposed approach is aimed at verifying user data correctness.

The two research question posed in Section 4.1 have been answered throughout the chapter. Regarding question **Q1**, the proposed solution completely detaches the verification concern from the business logic. All the verification functionality (inc. the enhanced feedback system) is encapsulated into an object. This object is deployed into a development by using a couple of lines of code and works transparently while the user interacts with the application. As result, a non-invasive verification method is provided. The verification logic is treated as a separate aspect clearly away from the application low-level behaviors. This improves encapsulation and reduces the application code, enhances its readability and eases the development process.

Furthermore, the verification rules are defined into a well-structured and independent set of rule files. They are automatically loaded by the verification framework. This improves modularity and encapsulation. Related rules are defined into the same file, and rules corresponding to different GUI panels are organized into different folders. AOP languages are not used, therefore the rules are not weaved into the application code. Instead, they are completely detached from the application to avoid code bloating problems and to ease

software maintenance [23]. The rules can now be coded from the very beginning, even before any application code has been written. It allows the early detection of errors in the requirements and the GUI, thus improving the application design [85].

The verification rules are checked as an invariant. Single-state checks are made to verify the set of safety properties composing the GUI state in order to ensure its consistency during execution. This approach is practical and eases tool implementation. However, it might reduce the expressiveness of the verification logic. At a specific time, only current values of properties are available to be verified. Therefore, checks related to the execution sequence as well as those based on previous values of data (i.e., the history) can not be implemented.

In response to research question **Q2**, general purpose and interpreted languages are used to write the verification rules. Unlike AOP languages, interpreted languages are not compiled along with the application code, but they are loaded and executed at runtime. This allows developers to change the rules code at runtime. These changes are immediately visible in the GUI because the verification framework automatically reloads the changed rules during application execution. This feature runs on the fly without recompiling the code nor relaunching the application. As a result, software testing becomes a more flexible, dynamic and thus more efficient task. It is worth noting that none of the approaches analyzed in Section 2.1.2 support this feature.

There are many of interpreted languages from which developers can choose the most suitable for verification. These languages are powerful and provide a high expressiveness and flexibility. They also allow to implement correction actions within the rules to bring the GUI to a consistent state. Developers feel comfortable with the use of programming languages to define assertions [119].

However, this flexibility and freedom provided by interpreted languages can bounce back against us during the verification process. Scripting languages do not usually provide formal mechanisms to allow developers to check the correctness and consistency of the rules code. As said in Subsection 4.4.5, developers should use an external method (e.g., constraint logic programming) if they want to ensure the consistency of the rules set. S-DAVER can not provide a mechanism for this purpose because it ignores, *a priori*, the language selected for verification. However, in order to mitigate this problem, its verification engine includes a set of built-in mechanisms to provide robustness when the rules are incorrect or incoherent.

Using the code in the rules to implement correction actions directly on the GUI data might also be dangerous. Despite it represents a powerful feature to bring the GUI to a consistent state after a rule violation, and despite the scope of correction is limited to the GUI, the actions implemented by developers may introduce inconsistencies in the GUI data. Again, the responsibility of a safe implementation goes to the developer. Besides the rules themselves, there exists not a method to guarantee that correction actions fulfill the requirements.

The lack of a method to check the correctness and consistency of the verification rules represents the main limitation of S-DAVER compared to most of Runtime Verification approaches. Attempting to solve this problem is one of our major concerns and the main priority as future work.

4.10.2 The S-DAVER Open-Source Implementation

An open-source implementation of S-DAVER is provided. This implementation can be downloaded from a public repository, an is ready to be integrated and used into Qt-based applications. The current implementation can be adapted by developers to be used in many different verification scenarios involving a different verification language, even a different GUI platform.

The S-DAVER implementation was successfully integrated into two FOSS applications as proof of concept. Resulting from this experiment, these two applications now have a separate set of rules to verify the input data instead of having the rules scattered in the application code. They also provide interactive feedback to users, which was a feature not present in the original versions. The original application code was reduced as well, reaching reduction rates up to 40%. The more restrictive and complex the input data constraints are, the higher is the number of removed lines of code. The performance of S-DAVER was tested presenting promising results. However, a more exhaustive analysis of its performance should be conducted in the future.

The provided implementation is fairly mature, but it presents some limitations that deserve to be described. First, the `VerificationEngine` treats rules as isolated functions. References from one rule to another can not be done, and external variables can not be used. This restricts the expressiveness of the selected language. Moreover, using a naming convention to define the rules would be also desirable. For example, automatically bind

a specific GUI event over a specific GUI widget to a rule with a name based on that <widget,event> pair. This interesting feature will be considered for future work. Finally, *GUI Intervention* modules are restricted by the own limitations of the functionality provided by the GUI system. Developers should choose carefully according to their needs. These limitations may reduce the effectiveness of the current version of S-DAVER. However, fix them is just a matter of implementation.

4.10.3 S-DAVER Compared with Other Verification Approaches

Compared to the verification approaches analyzed in Section 2.1.2, the main innovations provided by the approach proposed in this chapter are described in the following:

1. It describes a real lightweight data verification method. It does not require to formally describe the requirements or the expected behavior.
2. The requirements are directly translated into independent pieces of code. They are included in separate files that can be modified while the application is running.
3. The developers are free to choose between a wide range of interpreted languages to define the verification rules.
4. The framework is easy to integrate and use into a development, and works transparently to the developers.
5. The framework is concerned with the final experience of developers and users. Dynamic and visual feedback is used to interactively show information about the verification process during testing and execution stages.

Table 4.3 summarizes a brief comparison between S-DAVER and some of the most relevant implementations of the approaches described in Section 2.1.2. One example is RAVEN, which from the analyzed approaches, is the only one specially oriented to GUI. Unlike S-DAVER, its implementation is closed to the Java GUI platform. It uses a XML-based language to define the rules, which reduces considerably the expressiveness of the specification. E.g., defining rules in which a value is computed by using some data in the GUI would not be straightforward.

RuleR and LogScope use an application state against which the rules are checked. The idea underlying S-DAVER is rather similar, as the current execution of the GUI is considered as the application state. However, these methods consider the trace of execution (i.e., data

history) while S-DAVER does not. RuleR and LogScope use formal logic to specify the rules. Our framework proposes a less formal approach to gain flexibility when defining and maintaining the rules. However, it looses consistency in the specification. During verification, RuleR checks all the rules whose antecedents evaluate to true at a specific time. S-DAVER checks only the rules involving properties changed due to the user interaction. LogScope has a different nature. It works offline and standalone because it is aimed at analyzing behavior logs.

MaCS and the approach proposed by Zee et al. define the requirements by using languages specially designed for such verification processes. This implies that the expressiveness of these verification methods is reduced to the functionality provided by these languages. In contrast, S-DAVER is open to use general purpose languages to provide developers with more flexibility and higher expressiveness to define the requirements. A relevant limitation of the approach proposed by Zee et al. is that developers write the requirements directly in the source code, which makes this solution totally code-intrusive.

The MOP-based solutions and J-LO use the MOP language and a LTL logic, respectively, to define the verification rules. In contrast to S-DAVER, they provide a formal definition of the requirements. However, these solutions use AOP code (i.e., AspectJ) into the application code to bind the rules to the properties to be checked. It implies a high intrusiveness in the target application, which has to be always recompiled when any rule is changed.

ASP.NET and HTML5 include tools for verifying data in web-based user interfaces. The main difference between these approaches and S-DAVER is that their verification code is tangled and scattered across the GUI specification. This reduces the readability and encapsulation of the rules dramatically. Moreover, these solutions do not support rules involving more than one GUI element. The set of properties that can be validated is also highly restricted by the simple APIs these solutions provide. ASP.NET mitigates this problem by allowing the execution of script code at the server. In S-DAVER, the limitations in the expressiveness of the rules will be imposed by the capabilities of the chosen verification language.

BeepBeep describes a filter in the server to intercept incoming messages from a web application. The data in the messages is validated according to an interface contract that, as in S-DAVER, it is written in a separate text file. Invalid messages are blocked. The main difference with the proposed approach is that it uses an extended LTL logic to address the interface contracts. The requirements can be only defined using this logic and the

	GUI oriented	Formal languages	Script languages	Ad-hoc languages	AOP languages	Code intrusive	Correction actions	Dynamic feedback
S-DAVER	■	?	■	□	□	□	■	■
RAVEN [65]	■	□	⊞	■	■	⊞	□	□
RuleR [18]	□	■	□	□	■	⊞	□	□
LogScope [16]	□	■	□	■	□	?	□	□
MaCS [107]	□	■	□	■	□	?	■	□
Zee et al. [232]	□	■	□	■	□	■	□	□
JavaMOP [97]	□	■	□	□	■	⊞	■	□
J-LO [26]	□	■	□	□	■	■	□	□
MOPBox [27]	□	⊞	□	□	■	⊞	□	□
ASP.NET [110]	■	□	⊞	■	□	■	□	■
HTML5 [1]	■	□	⊞	■	□	■	□	■
BeepBeep [79]	□	■	□	□	□	⊞	□	□

■ *yes* ⊞ *partially* □ *no* ? *unknown*

Table 4.3 Comparison between S-DAVER and some of the most relevant implementations described in Section 2.1.2.

checking process is limited by the functionality it provides. As S-DAVER, BeepBeep is easily bootstrapped by adding a couple of lines in the header of the web-application main page.

4.11 Conclusions

Integrating the usually complex and formal processes of Runtime Verification may not result in a profitable and effective solution in some scenarios like, e.g., when validating input data in GUIs. The approach proposed in this chapter describes a lightweight verification solution intended for users input. It is easily integrated into a development thanks to its high encapsulation and low code intrusiveness.

AOP languages are not used to link the rules to the application. Formalizations of the requirements or the expected behavior are not created. Instead, the rules are directly written using scripting languages and transparently integrated into the application. The script code of the rules is included in separate files, which are arranged within a file tree according to the internal structure of the GUI. The rules can be changed at runtime during application testing. GUI *interventions* are used to provide visual and interactive feedback aimed at enhancing the efficiency and the verification experience of developers and users.

As a result of this research, a fully functional framework named S-DAVER has been designed and implemented. This implementation was integrated into two real FOSS applications resulting in a reduction of the original source code, as well as in a better organization and management of the verification rules. S-DAVER makes data verification an integral part of the development, testing, and execution processes. All the verification processes are encapsulated in a layer that, once integrated into an application, establishes a trust relationship between the GUI and the business logic.

5

Modeling and Evaluating Quality of Multimodal User-System Interaction

Chapters 3 and 4 deal with the quality of interaction components separately. However, to achieve the quality of the whole interaction process, its components (i.e., user input and system output) should be analyzed together, including the cause-effect relationships between inputs and outputs over time.

This chapter extends our research to the study of different sensory modalities to provide data. Multimodal interfaces are expected to improve input and output capabilities of increasingly sophisticated applications. Several approaches are aimed at formally describing the multimodal interaction process to evaluate usability and quality of such systems. However, they rarely treat interaction as a single flow of actions, preserving its dynamic nature, and considering modalities at the same level.

This chapter presents PALADIN, a model describing multimodal interaction. It arranges a set of parameters to quantify interaction as a whole, minimizing the existing differences between modalities. It uses a stepwise description to preserve the dynamic nature of the conversation. PALADIN defines a common notation to describe interaction in different unimodal and multimodal scenarios, providing a framework to assess and compare the usability of systems.

5.1 Introduction and Motivation

Today's applications are more sophisticated and demand richer and more advanced interaction between users and systems. In this context, multimodal approaches combine several types of sensory modalities to provide communication with a higher bandwidth [45] and flexibility [180], as well as to improve interaction robustness due to disambiguation [181] and to offer a better interaction experience [222].

In order to improve the usability and users satisfaction in multimodal systems, interactions need to be carefully analyzed and formally described. Current state-of-the-art approaches (analyzed previously in Section 2.2.1) are able to describe and/or analyze multimodal human-computer interaction in different scenarios. Nevertheless, we found some common limitations that should be overcome to properly assess multimodal interaction.

First, different modalities are often analyzed separately or at different levels of abstraction, e.g., when speech and GUI are instrumented using different tools and thus quantified separately. Data collected using a specific modality is not considered seamlessly with the rest of modalities. An equivalent method is not used when quantifying interaction in different modalities. As a result, multimodal interaction is not treated as a homogeneous flow of actions, as it is happening in reality. This makes difficult to assess multimodal interaction as a whole or to compare different modality combinations to each other.

Second, it is also a problem that current methods use different representations to describe interaction in different multimodal scenarios. In this context, in which there exists an evident lack of standardization, comparing interaction extracted from different systems (e.g., the same application running in different handheld platforms) is troublesome and it implies a higher effort during the analysis process.

Third, methods based on static parameters or average metrics do not capture the dynamic nature of the dialog process. These approaches do not allow the analysis of information that changes over time, e.g., when monitoring user and system activity or when analyzing the interface response in real time. Otherwise, dynamic approaches ease "live" interaction instrumentation and support its analysis at runtime.

According to these problems, and in the context of interaction analysis and usability evaluation in multimodal environments, the following research questions are formulated:

> **Q1**: How can different modalities be analyzed at the same level of abstraction when assessing multimodal interaction?

Q2: How can interactions observed in different multimodal scenarios be compared to each other?

Q3: How can multimodal interaction be represented to allow its analysis from a dynamic perspective?

To answer these questions, this chapter aims at providing a generic and dynamic way to represent multimodal human-computer interaction (MMI) in diverse scenarios. As a result, build a uniform method to analyze and evaluate usability of different interactive multimodal systems. The main contributions of this chapter are described next.

(a) Definition of MMI parameters (Section 5.2). A new parameter set quantifying multimodal interaction has been defined based on previous work [114, 168]. Furthermore, an existing turn-based approach was extended to provide a higher temporal resolution when analyzing interaction. They provide a more abstract description of interaction compared to previous works, expanding the range of possibilities for its analysis.

(b) Design of a MMI model (Section 5.3). The design of a new model named PALADIN (Practice-oriented Analysis and Description of Multimodal Interaction) is described. It structures the parameters defined in (a) into a runtime model to describe multimodal interaction dynamically. Instances of this model are valid to support analysis, comparison, transformation, adaptation, and decision processes. Its design provides also a common format to describe interaction in different multimodal scenarios.

(c) Implementation (Section 5.4). An implementation of PALADIN is provided along with a framework to ease its integration into multimodal developments. It is also described how these tools are integrated into real multimodal and unimodal applications to conduct user studies and to assess multimodal interaction.

As a result of this research, the PALADIN model and the tools developed around it provide a framework ready to implement instrumentation and assessment of interaction in multimodal environments. The scope is reduced to GUI (a.k.a., visual), speech, and gesture modalities.

The rest of this chapter is structured as follows. Section 5.5 describes two experiments with users and real applications in which PALADIN have been integrated. A discussion of the proposed solution is provided in Section 5.6. Section 5.7 includes some conclusions. Additionally, Section 5.8 describes all the parameters that are modified or newly introduced in PALADIN compared to [168].

5.2 A Model-based Framework to Evaluate Multimodal Interaction

Once analyzed approaches in Section 2.2.1, we found very interesting challenges to be overcome.

We first analyzed different approaches which, despite being intended for the development of multimodal interfaces, they present interesting features for our evaluation work. In general, those methods using markup languages tend to lack structure, while those methods using models often lack a dynamic nature or are not descriptive enough to implement evaluation processes. Furthermore, development methods tend to focus more on the system, usually ignoring the "user side" of interaction.

Unlike such approaches, PALADIN tries to provide a more structured and more precise description of multimodal interaction to support its quantification by instrumentation. Moreover, PALADIN aims at describing interaction as a single information stream, treating different modalities at the same level of abstraction (e.g., considering a screen touch and a gesture as two generic input information elements, regardless of the modalities used).

Section 2.2.1 analyzed evaluation approaches as well. Many of them use parameters to evaluate interaction. Several of the parameters proposed by these approaches are reused in PALADIN. However, most of them need to be adapted in order to improve their generality and to enable a dynamic description of interaction, a lack in these approaches. Other evaluation approaches are based on the observation of the user. Unlike these approaches, PALADIN is not intended for recording the user actions, but for supporting the description and quantification of multimodal interaction.

PALADIN structures parameters into a runtime model. Using a model instead of a markup language or a log file provides important advantages:

- the information is **better structured** and organized. The dependencies between data located in different sections of the model (i.e., reference and containment relationships) are explicit.
- the **runtime nature** of models like PALADIN (i.e., information is described in a stepwise manner) provides an implicit relationship between data and time, enabling the dynamic analysis and processing of data.

- a model is **based on a metamodel**, which provides a uniform structure for the model instances, as well as metadata about the information they can hold. This provides a proper basis for the automatic processing and management of the information.
- frameworks like EMF provide an **ecosystem of standard tools** that provide automatic functionality for data processing, code generation, model transformation, statistical processes, etc. (read Chapter 2 in [203]).

PALADIN tries to fill the gap between systematic models for system development and the available great quantity of parameters and metrics to measure user-system interaction. Thus, a model describing the course of dialog between the user and the system in a multimodal environment is presented. Its aim is to provide a basis for the analysis of interaction and the evaluation of usability of multimodal dialog systems (MMDS). According to the approaches described above, PALADIN can be classified as an evaluation method, based on a model structure, that follows a quantitative approach to describe the multimodal interaction process.

In this section we describe the parameters on which PALADIN is based. Starting with a short introduction on levels of information exchange, we ground the model on a turn concept, which chronologically structures the process of information exchange between user and system. The parameters used to semantically describe the interaction and quantifying it are described afterward.

5.2.1 Classification of Dialog Models by Level of Abstraction

A classification suitable for modeling spoken dialog systems using acoustic-, word- and intention-level was introduced in [195]. With respect to the context of multimodal systems, we modified this classification in order to make it more abstract and suitable for each modality. Thus, it is proposed to use *signal-*, *element-* and *concept-level* for the classification.

Each level describes a particular abstraction of the information transfer between user and system. The transfer is modeled with physical signals like sonar waves (speech) or light (GUI) at *signal-level*. The *element-level* is an abstraction of the signal-level. Here, a model uses elements (cp. Section 5.2.3) of the user interface or their actions for the description of the transfer. A still stronger generalization is the *concept-level*, which does not differ for any modality. User and system exchange just semantics units. One possible form are

attribute value pairs as used in PARADISE [217] or by Schatzmann and Young in their hidden agenda model [196].

5.2.2 The Dialog Structure

PALADIN structures a dialog as a sequence of alternate system and user turns. It is assumed that system and user turn do not overlap each other. Therefore, a turn represents the basic unit in which a dialog can be decomposed. Previous works (e.g., [168, 186]) proposed such a structure to model interactions with spoken dialog systems (SDS).

In order to describe the human-machine interaction in more detail, the system turn and user turn were redefined according to the new dimension of multimodal interaction. We propose dividing the user and the system turn each into three stages as showed in Figure 5.1.

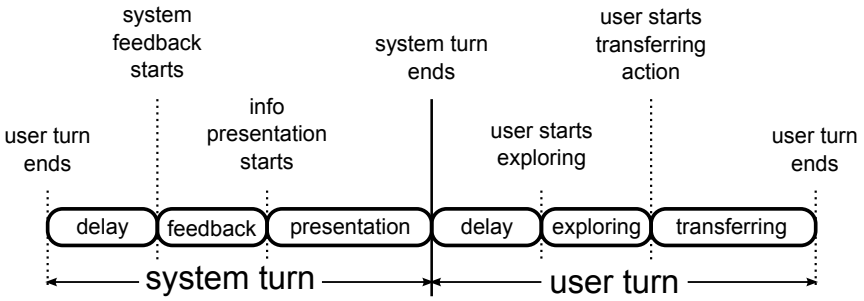


Figure 5.1 Interaction system and user turn in detail.

- In **system turn**, during *delay* the system does nothing from the users point of view. While *feedback* the system sends a signal to the user to indicate that the last input of the user is being processed, e.g., showing a clock symbol in the screen or playing a sound. Finally, in *presentation* the system response is provided to the user, e.g., a talking head using speech and gestures to show the available options.
- In **user turn**, *delay* ends when the user starts perceiving the information provided by the system. During *exploring* the user analyzes this information, e.g., scrolling in a GUI. While *transferring* the user supplies information to the system. That could be entering data in some fields followed by the usage of a send button in GUI, or uttering a sentence in speech.

This new definition of turn provides a uniform and symmetric perspective of user and system turns. It also separates elements carrying process information or feedback, both for

users and the system. Moreover, considering an exploring stage between the delay and the transferring action of the user helps us to distinguish between the time during which the user has no response, and the time he/she is taking the information in.

5.2.3 Using Parameters to Describe Multimodal Interaction

Parameter-based approaches have been successfully used to quantify and evaluate interaction in SDS for more than 20 years [114]. Related work [22, 58, 114, 168] showed a high correlation in SDS and MMDS in terms of how the interaction between the user and the system is performed. Therefore, most of those parameters used to evaluate SDS can be directly transferred —by adapting their definition— to a MMDS context.

In this situation, we decided to use a parameter-based approach to quantify user and system interaction in multimodal environments. The validity of these approaches for evaluating MMDS have been showed in previous works like [58].

The parameters described by Kühnel and Möller in [114, 168] are the basis for the work presented in this chapter. These authors described a first approximation of MMDS parameters based on those used to evaluate SDS [166, 167]. However, these parameters are too close to SDS evaluation, which hinders the assessment of multimodal systems that do not include speech modality, as well as the implementation of more abstract analysis of multimodal interaction. Therefore, a transformation of the base parameters was needed in order to describe multimodal interaction as a whole and regardless of the modalities used.

5.2.3.1 Adaptation of Base Parameters

The base parameters were included in PALADIN after some abstraction and specialization actions described in the following.

On the one hand the **abstraction process**, in which parameters related to the information exchange and the communication process between the user and the system were provided with a more generic definition. E.g., “words” were transformed into “information elements”. Such elements do not belong to a specific modality, but they can represent a word, a gesture, a widget in a GUI, an eye movement, or other elements carrying information.

A particular case of abstraction is the concept of “noise”. This parameter was adapted from its speech-based definition to denote potentially disturbing elements in any modality (e.g., advertisements in GUI, noisy sounds in speech modality or people moving in the

background of the scene in gesture modality). This parameter allows us to compute the percentage of information that is not relevant to reach the main goal of the dialog. It should be assumed that noise has to be annotated manually by an expert, since automatic detection of disturbing elements is an unsolved problem.

On the other hand the **specialization process**. First, the definition of those parameters related to specific features of speech interaction was left as is. Additionally, new specific parameters for GUI and gesture interaction (described later in this section) were incorporated. Specific parameters are used in two cases: when a particular aspect of interaction has to be annotated (e.g., it was an speech recognition error) or when additional information is needed to enhance generic parameters (e.g., to know how many words in "elements" are "unrecognized words").

After the transformation of the base parameters, and according to the new turn structure described above, the following parameters (which are further described in Table 5.9)[1] were added to PALADIN:

Dialog and communication: system feedback duration (*SFDu*), system action duration (*SAD*), user response delay (*URD*), user exploring delay (*UED*), user exploring duration (*UEDu*), user action duration (*UAD*), concepts per system turn (*CPST*), feedback per system turn (*FPST*), noise per system turn (*NPST*), concepts per user turn (*CPUT*), feedback per user turn (*FPUT*) and noise per user turn (*NPUT*)

5.2.3.2 Defining new *Modality* and *Meta-communication* Parameters

New parameters were defined to extend the information related to the different modalities used during multimodal interaction. At each of the turns described above, the user or the system uses one or several modalities to provide input or output data, respectively. Modality changes are performed, e.g., to improve the efficiency of the communication, due to a changing environmental context, etc.

New parameters were defined in order to annotate from which to which modality a change is performed. The origin —the user or the system— and the reason for that change were considered as well. Unfortunately it is not possible to automatically annotate the

[1] Tables 5.9, 5.10, 5.11 and 5.12 describe different interaction parameters including the modalities for which they can be applied (Mod.), the interaction level at which they are collected (Int. Lev.) and the measurement method (Meas. meth.) Table 5.8 further describes these abbreviations and their values.

reason for changing the modality (specially for the user), which might be asked to the user after interaction, e.g., in an interview.

With these parameters we can fully describe modality changes including all relevant data for analysis. E.g., it is possible to describe that the system switched the output modality from speech to GUI due to an environmental change. These parameters (described in Table 5.10)[1] summarize our contribution to better describe modalities usage in MMDS interaction:

Modality parameters: system modality change direction (*SMCD*), output modality change reason (*OMCR*), user modality change direction (*UMCD*), input modality change reason (*IMCR*), modality type (*MT*)

Additionally, new meta-information about the multimodal communication process was added to improve the analysis of dialog success and fail cases. These new parameters allow experts to better differentiate between dialog cancellation, that implies no task success, and dialog restart, in which user might reach task success. The cases in which a barge-in attempt is successful or not are also considered separately.

Furthermore, recognition errors are now considered also for GUI (e.g., input data that does not keep the expected format) and gesture (e.g., unrecognized gestures) modalities. The following parameters (described in Table 5.11)[1] were added to the model in order to include the scenarios described above:

Meta-communication: number of user cancel attempts (*#cancel*), number of user restart attempts (*#restart*), successful user barge-in rate (*SuBR*), number of data input validation rejections (*#DIV rejection*)

5.2.3.3 Defining new Parameters for GUI and Gesture Interaction

Specific features of GUI interaction were incorporated into the new parameter set as well. These parameters describe navigation and text input in GUI modality in terms of time, device usage, screen content and further specific properties. Their open definition supports potentially any input device of this type like, e.g., mouses, keyboards, touch-screens or eye tracking systems.

These parameters also distinguish between exploring and transferring actions. Exploring ones are used to explore and analyze the content provided by the system, e.g., scrolling down a web page. Transferring actions are used to provide the system with input data, e.g., a date is inserted. The validity of the data provided by the user, which has to conform to a

set of allowed actions, the input format [54] or other restrictions, has also been considered. The following parameters (further described in Table 5.12)[1] were added to the model to describe peculiarities of GUI input:

Input: keyboard usage percentage (*KUP*), mouse usage percentage (*MUP*), mouse movement length (*MML*), mouse move frequency (*MMF*), number of exploring actions (*#EAC*), exploring actions rate (*EAR*), number of transferring actions (*#TAC*), transferring actions rate (*TAR*)

Specific parameters for gesture interaction were not added to the PALADIN model. Therefore, gesture interaction is annotated within the model by using the generic parameters intended for describing the dialog and communication content. As stated in Section 5.7, to analyze gesture input and output in-depth and define a new set of parameters to describe the peculiarities of this modality is a priority in a short-term.

5.2.3.4 Classification of the Multimodal Interaction Parameters

The resulting parameters were classified following the classification in [166], which was extended by a new category: Modality-related parameters. The parameters *number of system output modality changes (#SMC)*, *number of user output modality changes (#UMC)*, *relative modality efficiency (RME)* and *multimodal synergy (MS)*, originally belonging to dialog- and communication-related parameters in [168], were moved into it.

Finally, PALADIN parameters are structured as follows:

- Dialog and Communication related parameters (Table 5.9 and [168])
- Meta-communication related parameters (Table 5.11 and [168])
- Cooperativity-related parameters ([168])
- Task-related parameters ([168])
- Input-related parameters (Table 5.12 and [168])
- Output-related parameters ([168])
- Modality-related parameters (Table 5.10 and [168])

Section 5.8 describes all parameters modified or newly introduced in PALADIN compared to [168]. Table 5.7 gives an overview of all parameters used in PALADIN, including references to their definitions.

At this point, research question **Q1**, formulated at the beginning of this chapter, can be answered. Multimodal communication content is described using generic parameters. Such parameters are used to quantify interaction seamlessly, as a uniform and homogeneous flow of information between the user and the system, and regardless the modality in use. Furthermore, interaction meta-data is collected at a communication-level, not at a modality-level.

As a result, inputs and outputs in different modalities are considered at the same level of abstraction. PALADIN uses *information elements* as the generic unit to quantify the communication between the user and the system instead of counting screen touches, window objects, spoken words or performed gestures, which are elements of a particular modality. Implications of this research question are further discussed in Section 5.6.

5.3 Design of PALADIN

The proposed design contains all data necessary to compute the parameters described above. This data set is subject to two conditions. First, it should be as small as possible in order to ease the extraction and computing processes. Therefore, the selected set of data includes only those required for automatically computing the aforementioned parameters. At the same time, we tried to maximize the number of data that can be automatically collected, and thus minimize those that have to be annotated by hand.

Data to be collected can be classified into five groups.

1. **Time metrics**, necessary to compute the duration of each stage in user and system turns (e.g., feedback stage duration).
2. **Communication content**, to describe the type of the information elements exchanged between the user and the system. Such data give us a very rough indication of how the interaction takes place (e.g., number of noise elements).
3. **Input and Output metrics**, to describe the peculiarities of each modality and to provide added value to the communication content data (e.g., speech parsing result or number of navigation actions).
4. **Meta-data** about the dialog process, aimed at quantifying the number of system and user turns which relate to a specific interaction problem (e.g., cancel-turn or speech recognition error).

5. **Modality** data, to describe the different modalities used during the communication process and to annotate features of each modality change (e.g., input modality type or reason for the modality change). The parameter *modality type* allows the annotation of fusion and/or fission of modalities used in system or user turn.

To inherit the dynamic nature of a dialog, the design of PALADIN is centered around the "turn" concept. Since a dialog can be described as a sequence of alternate system and user turns (see definition of turn in Subsection 5.2.3), the metrics described in the previous list are collected for each single turn. As a result, instances of this model are a stepwise representation of the interaction process between the user and the system in a dialog. Once the interaction process has finished, the data recorded at each interaction step (i.e., a consecutive pair of system and user turns) is used to compute global or average metrics, or in other words, the interaction parameters addressed in Section 5.2.3.

The reader should be aware that system and user turns are not discrete. Even some of their stages might not be present in specific situations. For example, the delay stage in a system turn may be imperceptible to the user, or there may be no feedback stage at all. In another case, it might be impossible to distinguish between the end of the delay stage and the start of the exploring stage during user turn if the exploring action involves actions such as reading, which can only be assessed through observation or eye-tracking.

As depicted in Figure 5.2, the basic structure of the model is represented by an aggregation of *turns* composing a *dialog*. While *dialog* holds data related to the whole process (e.g., task success information), each *turn* holds data corresponding to the interaction process at a concrete step of the dialog. A *turn* is composed of data related to stage timing and communication content common to the system and the user (e.g., number of feedback elements, duration of the *action* stage). Moreover, a turn is extended with further information (i.e., *meta-communication* and *input/output* data) depending on whether it is a *system turn* or a *user turn*.

Meta-communication data is partly common to the user and the system, e.g., information about help and correction turns. However, most of it refers only to user problems (e.g., cancel or restart turns, barge-ins attempts[2]) or to system problems (e.g., speech or gesture recognition errors, invalid input data errors). Annotating meta-data by turn provides a link

[2]A barge-in attempt occurs when the user intentionally addresses the system while the system is still speaking, displaying the information of a GUI, performing a gesture or sending information using another modality.

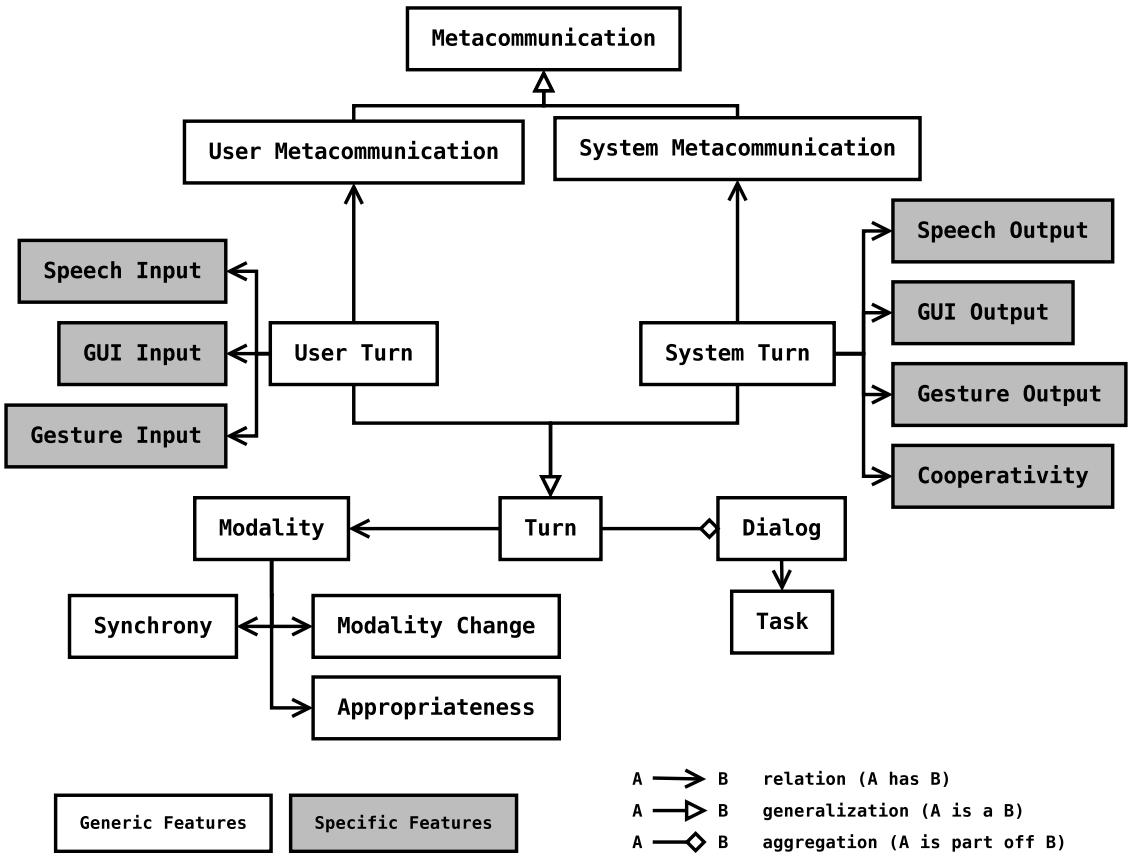


Figure 5.2 Arrangement of interaction parameters within PALADIN. Design illustrated as Ecore model diagram.

between an error, the time at which it happened, and the interaction conditions up to that moment.

Input data is only included into the *user turn*. It holds information related to the peculiarities of input in the different modalities used during the dialog. Our current model supports speech (e.g., automatic speech recognition metrics), GUI (e.g., usage of pointing and text devices) and gesture inputs (e.g., automatic gesture recognition metrics). **Output** data is the equivalent for *system turns*. It includes information about output peculiarities (e.g., correctness of system speech answers).

This part of the model provides a new level of abstraction to analyze input and output in a more detailed perspective than the provided by the *turn* object. Considering input and output separately provides higher flexibility to describe and analyze the different modality combinations used in a multimodal system.

Cooperativity data is considered only in *system turns*, and describes the contextual appropriateness of the information provided by the system (i.e., the response). This data is annotated by experts, who judge whether a system response is appropriate in its immediate dialog context. An adapted version of the Grice's maxims for cooperativity [76] is used to determine such parameters.

Finally, **modality** data is also described by turn, indicating which modality or combination of them is used at each time, as well as some properties of such modalities (e.g., appropriateness or lags between output of different modalities). This object of the model implicitly provides information about modality changes, e.g., the turn in which a change happened and from which to which modality it was performed. Moreover, recording modality-related data in a stepwise manner allows experts to evaluate the performance and usability of different modality combinations depending on the current dialog conditions.

As the reader can see in Figure 5.2, all objects in the model are grouped around the *turn* object, allowing them to inherit its dynamic character. This object acts as a link between data belonging to different sections of the model. Thus, data from different sources can be easily combined, increasing the expressiveness of the model. This feature eases the detection of errors and allow experts to easily draw complex conclusions; e.g., "the number of speech recognition errors increases when the number of inserted concepts is above average" or "60% of such errors imply a modality change".

Extending the model is easy as well mainly due to the turn-centered design. In most of cases, it is just a matter of adding new attributes to an existing object of the model, or

creating a new one. Anyway, significant changes would be made at the edges of the model, minimising the impact on its main structure.

The features of the proposed design give us an answer to question **Q2**. All the metrics described above are well structured within a model representation. Using this "common representation" to describe interaction in different multimodal —and unimodal— scenarios provides a basis to easily compare different interaction records (i.e., the model instances). All model instances have the same format and structure regardless of the way the data were collected and put into them.

Moreover, **Q3** is also answered at this point. The turn-based design of PALADIN provides a stepwise description of multimodal interaction. Data are quantified and annotated into the model in different turns, at specific points in time during the interaction process. This creates a relationship between data and time, providing experts with a basis to implement dynamic analysis of interaction.

In addition, this model design enhances the solution proposed above for question **Q1**, since two different levels of abstraction (i.e., a generic level and a more specific one, both annotated in Figure 5.2) are now used to describe the communication content.

5.4 Implementation, Integration, and Usage of PALADIN

PALADIN was implemented according to the design described above in Section 5.3. Its implementation is provided as an open-source contribution to the HCI community, and can be downloaded from [139]. The implementation of PALADIN was done within the Eclipse Modeling Framework (EMF), which provides automatic code generation, syntactical validation and model transformation functionality. Chapter 2 in [203] gives a compact introduction of the concepts behind and usage of EMF.

The data model of PALADIN was initially defined in an XML Schema Definition (XSD) by following the appropriate W3C recommendations [128, 209]. Then, the schema definition along with EMF were used to automatically generate the Java code for our model. Automatic code generation is available for several programming languages, e.g., Java, .NET framework, etc.

PALADIN is a metamodel intended to be used within an interaction evaluation environment. In order to ease its integration into research and production systems, a helping framework was developed. It is called Instantiation Framework (IF), its implementation

is open-source and can be downloaded from [139] as well. The IF is aimed at serving as a bridge between the interaction source (e.g., a filter extracting live interaction from an application, an application simulating user-system interaction, an interaction log) and the PALADIN instances. It eases the creation of model instances in real-time and helps to manage their life-cycle.

The IF works as an independent element during the interaction instrumentation process. It needs to be notified about actions of the user and the system during interaction (e.g., the user made a click, the system provided a feedback message). The IF uses the information provided by the interaction source to create and update the current instance of PALADIN. The permanently updated model instance can be accessed during the interaction for runtime analysis, or once the interaction is finished to implement off-line analysis.

The current implementation of PALADIN and the IF can be easily integrated in the source code of a Java application. In [139] it is carefully described how PALADIN can be used with or without the IF, and how these tools are integrated into an application to implement multimodal interaction analysis. The IF provides a facade[3] from which it is easily notified by an external tool instrumenting interaction. This facade class includes a set of methods describing different actions of the system and the user (e.g., `touch()`, `overallWords()`, `newGuiFeedback()`) and meta-actions that can occur during multimodal interaction (e.g., `incorrectlyParsedUtterance()`, `interactionStarts()`).

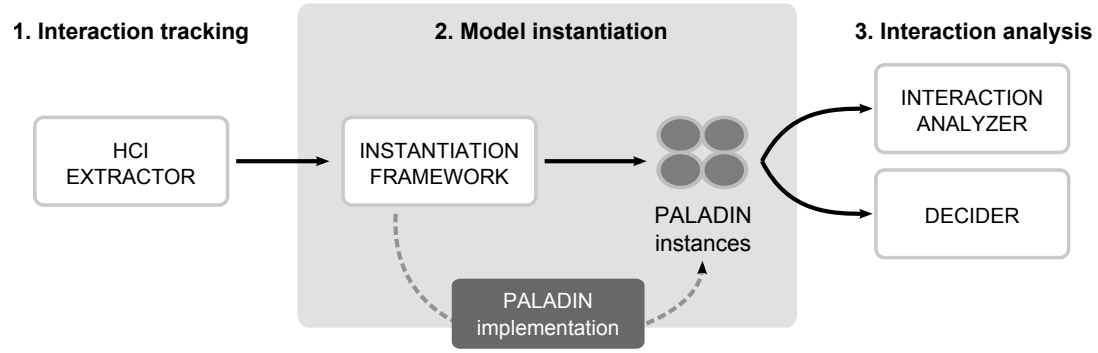


Figure 5.3 Overview of the PALADIN instantiation process and its context.

Figure 5.3 depicts a typical instantiation scenario for PALADIN, which is similar to those described in Section 5.5. This scenario can be divided into three stages:

[3] A facade is an object that provides a simplified interface to a larger body of code, such as a class library or a software framework.

1. **Interaction tracking**: in this stage the user-system interaction is tracked to capture the parameters necessary to describe it. Interaction events may be captured from real application usage in real-time (see the Android HCI Extractor in Section 5.5) or from other scenarios in which interaction is simulated (e.g., MeMo [62]). Interaction can be also extracted from logs, or anyhow artificial produced.
2. **Creation of model instances**: the IF is notified with the interaction events through its facade. This information is used to create live PALADIN instances.
3. **Interaction analysis**: the data stored in the model instances is used off-line to implement data analysis, comparison or transformation processes. Such instances can be also accessed through the IF at runtime if we are making live decisions for application adaptation.

In [139] it is also described how PALADIN and the IF can be extended and customized to implement additional features of multimodal interaction. As the reader can see in these tutorials, integrating the IF into a project to create live instances of PALADIN is really easy and does not require more than ten lines of code. However, adapting these tools to support new analysis features (e.g., to support a new modality) requires a higher development effort and more knowledge about the model and the instantiation architecture.

5.5 Application Use Cases

This section shows how PALADIN can be integrated in different scenarios in which user-system interaction is analyzed to improve usability of systems. PALADIN was integrated into four different Android applications (summarized in Table 5.1) in the frame of two experiments in order to instrument selected parameters of user-system interaction (described in Table 5.2). Beside an app's use case, the availability of its source code was a criteria for the choice.

Experiment 1 (described in Subsection 5.5.1) tries to show that PALADIN can be used to faithfully describe multimodal interaction, and thus provides a basis for its analysis and comparison as well as to make decisions. It also shows some preliminary conclusions drawn from the analysis of the resulting PALADIN instances. This experiment runs a restaurant search app (ReSA) into an smartphone. The app was developed at the institute of one of the

	ReSA	ReSA 2.0	Trolly	Vanilla Music Player
Usage	searching restaurants	searching restaurants	shopping list	searching and playing music
Developer	Stefan Schaffer	Xin Guang Gong	Ben Caldwell	Adrian Ulrich
License	proprietary	proprietary	GPL v3 [74]	GPL v3
Version	1.0	2.0	1.4	0.9.10
Source code	—	—	http://code.google.com/p/trolly	https://github.com/adrian-bl/vanilla

Table 5.1 Information about the four Android apps used in the two experiments.

authors for research on modality selection in multimodal systems. Exclusive usage of touch or speech are supported for input, as well as GUI for output are supported in ReSA.

Experiment 2 (described in Subsection 5.5.2) tries to show that PALADIN can be used to conduct a real study with users. Gong and Engelbrecht used PALADIN to analyze the influence of specific system and user characteristics on the quality of users-judgment prediction models [73]. This experiment integrates PALADIN into the tablet applications *ReSA 2.0*, *Trolly* and *Vanilla Music Player*. ReSA 2.0 bases on ReSA and is also an internal development. Trolly and Vanilla Music Player are full functional open source apps and available for free at Google play (the official marketplace for Android apps). It was not the goal of the experiment to examine multimodal interaction, but the influence of an app's complexity on users' judgements. For that reason, each app had GUI as output modality and touch for input. The speech functionality of ReSA was not used in this experiment. Nevertheless, this experiment proofs the usage of PALADIN in everyday apps, that are developed without any intention to be used in a usability evaluation study.

5.5.1 Assessment of PALADIN as an Evaluation Tool

This section describes an experimental set-up and procedure based on the use of the *ReSA* application. This Android application is used by the participants to search for a restaurant

		Experiment 1		Experiment 2
Type	**Parameter**	**GUI**	**Speech**	**GUI**
Dialog and Communication	*EPST*	■	■	■
	EPUT	■	■	■
	SAD	■	■	■
	SFD	■	■	■
	SFDu	■	■	■
	UAD	■	■	■
	UFD	■	■	■
	UFDu	■	■	■
Modality parameters	*IMCR*	■	■	■
	UMCD	■	■	■
Input	*#EAC*	■	■	■
	#TAC	■	■	■
Speech-input	*CER*	□	□	□
Modality parameters	*MT*	■	■	■
Meta-communication	*#ASR rejections*	□	■	□

■ *yes* □ *no*

Table 5.2 Parameters recorded in the two experiments, grouped by parameter type.

according to the indicated preferences. This experiment is used to prove the validity of PALADIN for evaluating the usability of multimodal systems. It is also described some initial results obtained from the analysis of the interaction data extracted in this experiment.

5.5.1.1 Participants and Material

Fifteen native German speakers (average age 29.5 years, SD = 4.6 years, 5 women, 10 men) participated in the study. All participants were students or research colleagues of various disciplines, mainly engineering. None of them had any former experience with the used application. The "Restaurant Search App" in version 1.0 (ReSA) was already used in former experiments on modality selection and perceived mental effort. We used it for our evaluation study, because the complete source code was available for us and the application offers touch and speech input to the user. Table 5.1 gives further information about the app.

All users interacted with ReSA installed on an Android-based smartphone (HTC Desire, Android 2.2, 1GHz, 512MB RAM). The available input modalities were speech (Wizard-of-Oz setting) and touch (via the smartphone's touch-screen). GUI was the only output modality used during the interaction. Users were able to use any of the available input modalities to interact with the application, but not both in the same turn. Modality changes were automatically recognized, thus the user had not to explicitly activate speech recognition, e.g., by pressing a push-to-talk button. In case the user input was not recognized, a feedback message indicating the error was presented to the user, and the application returned —if not there— to the main screen (Figure 5.4, a).

ReSA was originally developed for studying users' behavior in input modality selection in dependency on recognition errors by the system. The possibility to control the system error rate is essential for such studies. For that reason, ReSA does not actually use an implemented automatic speech recognition (ASR) but a Wizard-of-Oz approach (e.g., as described in [71]) for speech input processing). The ASR as well as the natural language understanding were substituted by a trained assistant (i.e., the wizard) in our setting. He/she performed speech interaction steps by means of a specially-designed Java application, which was deployed on a notebook connected to the mobile device via wireless LAN. During the experiment wizard and participant stayed in separated rooms. Not until the participant had fulfilled all tasks, he/she was informed about the wizard.

Wizard-of-Oz approaches have been successfully integrated in testing scenarios involving speech and other input modalities, e.g., in [114, 197, 200]. Our wizard application included

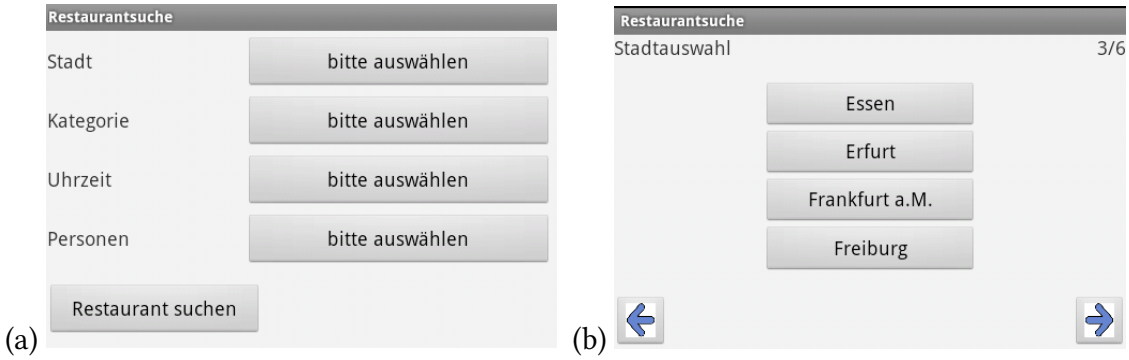


Figure 5.4 Restaurant Search App running on Android. See Table 5.3 for translations.

all the options and commands the user could say, as well as functionality to simulate speech recognition errors. The ASR simulation rejected a user utterance with a probability of 10 %, independently from the actual form and content of the utterance. The participants assumed that ASR worked with an open microphone, since he/she could talk to the system at any time, without pressing a button.

ReSA was added with the *Android HCI Extractor*[4]. This prototype tool —implemented within this work— is able to automatically extract and quantify the interaction between the user and the system in multimodal Android environments.

As depicted in Figure 5.4 (a), ReSA presents four restaurant search criteria in the main screen (i.e., a city, a food category, a desired time and the number of persons) that have to be answered by the user. When one of these options is selected, ReSA shows a set of screens (Figure 5.4, b) including a list with some of the available values. As ReSA uses German as default language, Table 5.3 shows the translation into English of the main speech commands used during interaction.

The user can use touch or speech to select any of the options. The user can show more values touching the arrows on the bottom left and right, or by saying *next* or *previous* if he/she is using speech input. The items are ordered alphabetically or numerically. An option can be selected by touching the corresponding button on the screen, or by saying the written text label, even if the option is not being displayed.

In this manner we can define in ReSA tasks with a different difficulty level. E.g., if the requested value is showed in the first screen (difficulty 1) the user can select it directly by

[4] An open-source implementation of the Android HCI Extractor can be downloaded from the PALADIN website [139]. More information related to this tool and its integration with the model and the framework described above can be found in this page as well.

German (speech)	English (speech)	German (GUI)	English (GUI)
Weiter	*Next*	bitte auswählen	*please select*
Zurück	*Previous*	Kategorie	*category*
Restaurant suchen	*Search Restaurant*	Personen	*persons*
Neue Suche	*New Search*	Restaurant suchen	*search restaurant*
Beenden	*Quit*	Stadt	*city*

Table 5.3 Translations of speech and GUI commands used in ReSA.

using speech or touch. However, if the value to be selected is showed in the third screen (difficulty 3) the user may proceed as follows: either he/she uses touch to navigate until the third screen and then selects the value, or he/she uses speech and utters the value directly without navigation. The tasks in ReSA range from difficulty 1 (easiest) to difficulty 5 (hardest).

This application design represents a benefit of speech input: the higher the number of interaction steps (i.e., screens) to reach a concrete option is, the greater the benefit of using the speech modality. Once all the options are provided, the user can select *search for restaurant* to send the request to the server and reach the last screen. The last screen allows the user to make a new search with *New Search* or finish the process with *Quit*.

5.5.1.2 Procedure

A single experiment took approximately 15-20 minutes. At first demographic data (i.e., age and gender) was gathered using a questionnaire. After that, the system was explained and the usage of touch and speech was demonstrated.

Each participant performed three training trails: touch usage only, speech usage only and multimodal using mixed modality. The real test comprised 6 trials. The tasks were presented in written form (e.g., "Please look for a Sushi restaurant in Berlin at 8 pm for 12 persons"). Difficulty of such tasks was systematically varied between 1 and 6 interaction steps for touch input and always 1 for speech input (if no speech errors are simulated). A trial was finished if all specified information was collected correctly and the request was sent to the server.

As a result of a complete test we obtained, for each participant, 3 PALADIN instances —described within XML-based files— corresponding to the testing trails, 6 instances corre-

sponding to the real test, and a set of audio records about the speech input provided by the user during the trails.

5.5.1.3 Data Analysis

In order to show the validity of PALADIN for multimodal interaction assessment, this subsection describes the implementation of different analysis processes based on the experiment described above. Examples of interaction analysis, task comparison and runtime decision are described in the following.

As a first step, a prototype analysis tool was implemented to provide experts with abstract representations of multimodal dialogs. This tool uses PALADIN instances to draw the "interaction stream" in a dialog. These streams, like the depicted in Figure 5.5, allow to implement quick analysis and comparison. Table 5.4 summarizes the set of parameters depicted in the charts.

Each chart (i.e., stream) describes orderly, and for each turn, the amount of information elements provided by the system (left bar, grey colour) and the user (right bar, dark grey colour). The left bar also indicates in a light grey color those elements corresponding to system feedback. The modality used to provide user input is indicated at the top of these bars. Mean values are denoted by dashed lines. User response and action times are also represented by using Bézier curves along the interaction stream. In the charts depicted in Figure 5.5 the difference between delay, feedback and action times is negligible because the interface was single-action (i.e., the user only performs one action per turn) and does not require exploration.

At this point we encourage the reader to take a look at the streams in Figure 5.5, from which we can draw some quick conclusions. For example, we can analyze the effect of speech recognition (ASR) errors in user interaction. The chart in Figure 5.5 (a) depicts a high-difficulty task (i.e., difficulty 5). The reader can know it because, when using GUI modality, the user needs five turns to select the required value for a specific restaurant search criteria.

The figure shows the participant using a combination of speech and touch modalities to do the task. When an ASR error occurred in turn 2 (see the error annotation and the system feedback informing about the error in turn 3) the user decided to use only touch modality to accomplish that step of the task, which is represented in the figure by the five low bars after the ASR error. Then, the user shifted back to speech modality. The scenario depicted in this

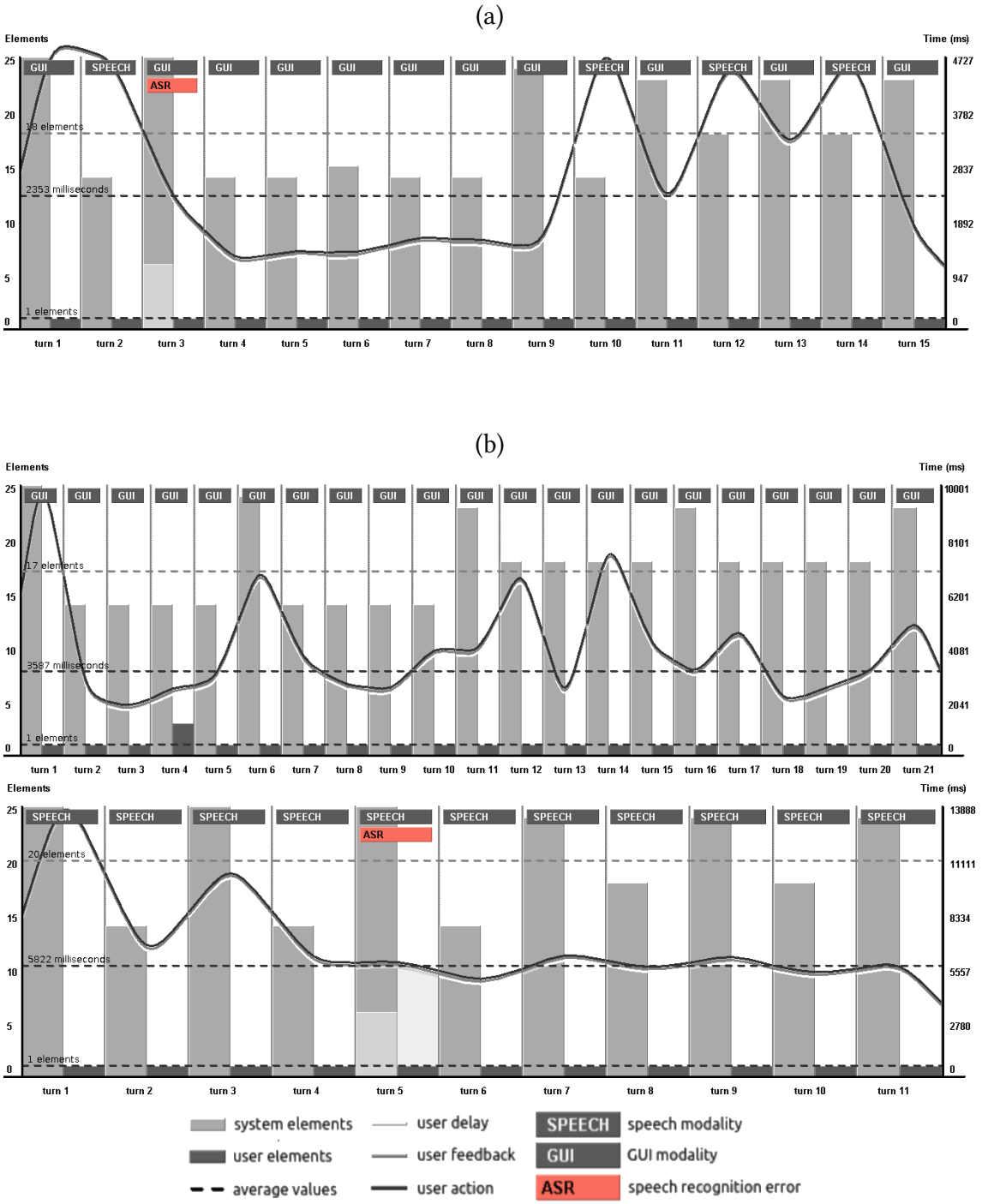


Figure 5.5 Graphical reports of several interaction records corresponding to the experiment using ReSA. Created with the multimodal interaction analysis tool.

Abbreviation	Parameters
#turns, #system turns, #user turns	Number of turns in a dialog.
EPST, $\overline{EPST}$	Elements per system turn; Average number of EPST in a dialog.
FPST	Feedback elements per system turn.
EPUT, $\overline{EPUT}$	Elements per user turn; Average number of EPUT in a dialog.
UFD, UED, UAD	User feedback delay, exploration duration and action duration.
UTD, $\overline{UTD}$	User turn duration; Average UTD.
#ASR rejections	Number of automatic speech recognition (ASR) errors in a dialog.
#UMC, #UMC:X-Y	Number and type of user input modality changes.

Table 5.4 Parameters visualized in the analysis tool captures.

chart was very common during the experiments, which shows an example of the impact of ASR errors in users behavior. In many cases the users shifted from speech modality to GUI modality to select a value when a speech recognition error occurred, instead of trying again by using speech.

As said above, the stream representations are useful to easily compare different interaction records to each other. Figure 5.5 (b) compares two different tests performed by the same user. Both streams depict two tasks of the same difficulty level (i.e., the user needs to select a value by navigating through the same number of screens). In the upper stream only GUI modality was used; meanwhile, in the lower one, only speech was used. Just taking a look to the charts we can make the following conclusions related to the efficiency of users when using ReSA.

First, the reader can see that, when using GUI input modality, the user needs at least as many turns as the task difficulty-level (4 in this case) to reach the requested value. However, using speech input the user needs only one turn to utter the value, always in case no ASR errors occur. Average user times per turn are higher using speech (3587 ms for GUI vs. 5822 ms for speech). However, we can confirm that in this application speech helps users to accomplish hard tasks more efficiently, because less turns are needed to select a value.

Second, after comparing both interaction streams we can conclude that the chosen layout for the user interface (which was designed conscientiously to test users efficiency) is inefficient for GUI interaction. This problem would be easily overcome by using a single-screen scroll-based interface, as in ReSA 2.0.

Besides data analysis and comparison, using a runtime model to describe multimodal interaction brings other possibilities for taking advantage of live data. One example is making runtime decisions based on the interaction data stored into a running instance of PALADIN (e.g., for interface adaptation). This feature was implemented in ReSA, which was augmented with a decider that, according to current state of user interaction, provided hints to improve users' efficiency in two different scenarios:

Decision scenario 1. The user is using touch, but using speech would be significantly more efficient (i.e., in high-difficulty task dialogs). In this case, the decider displays a message suggesting the usage of speech modality if the user is in a high-difficulty task (see Figure 5.6, a).

Decision scenario 2. Two or more ASR errors occur in the same dialog. In this case, the decider displays a message suggesting the user to adjust microphone settings to improve speech-input quality (see Figure 5.6, b).

5.5.2 Usage of PALADIN in a User Study

PALADIN was also used in a study [73] about user judgements on smartphone/tablet apps. The goal of the study was the analysis of the influence of specific system and user characteristics on the quality of prediction of user judgements from interaction parameters. Participants interact with three different apps on an Android based tablet, using the modalities touch and GUI. PALADIN was used to record the course of interaction for a subsequent analysis. The interaction parameters were used to analyze the relation between interaction characteristics and user judgements.

5.5.2.1 Participants and Material

The participants were divided into two groups by their age. The participants in the group of younger adults —7 women and 9 men, most are students of TU-Berlin— were from 17 to 27 years old (mean = 21.9, SD = 3.7). Furthermore, the participants in the group of elder adults

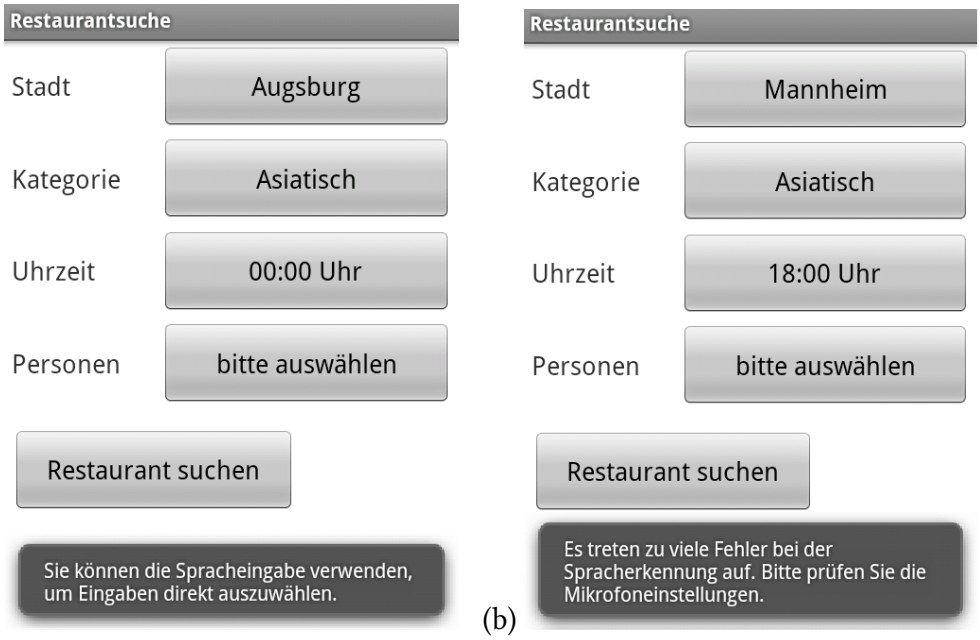


Figure 5.6 Runtime decider messages. (a) *Using speech, you can directly select an input.* (b) *The speech recognition does not work correctly. Please check the microphone settings.*

—9 women and 6 men— were in an age from 59 to 84 years (mean = 70.1, SD = 7.5). All 31 participants were native German speakers or had very good German language skills.

The used platform was a Google Nexus 7 tablet computer. Its display has a size of 7 inches (diagonal) and a resolution of 1280 x 800 pixel. The system's language as well as the layout of the virtual keyboard was German and the font size was left on standard setting. Solely the icons of the three used apps were shown on the home screen of the tablet.

A short description about the functionality of the three apps is given in the following. Moreover, Table 5.1 gives an overview about author, license, used version and availableness of each app.

The selection of the apps was carried out on the basis of 4 criteria. First of all, it was necessary to have access to the source code, in order to integrate PALADIN in an app. The other criteria were related to the experimental set-up. All three apps had to differ in the complexity of the interaction. Furthermore, each app had to allow task driven interaction and particularly gaming was not in the focus of the study. Finally, the app should reflect a functionality known by the participant from everyday life.

ReSA 2.0 bases upon ReSA with a pleasing design (compared to ReSA) and an improved

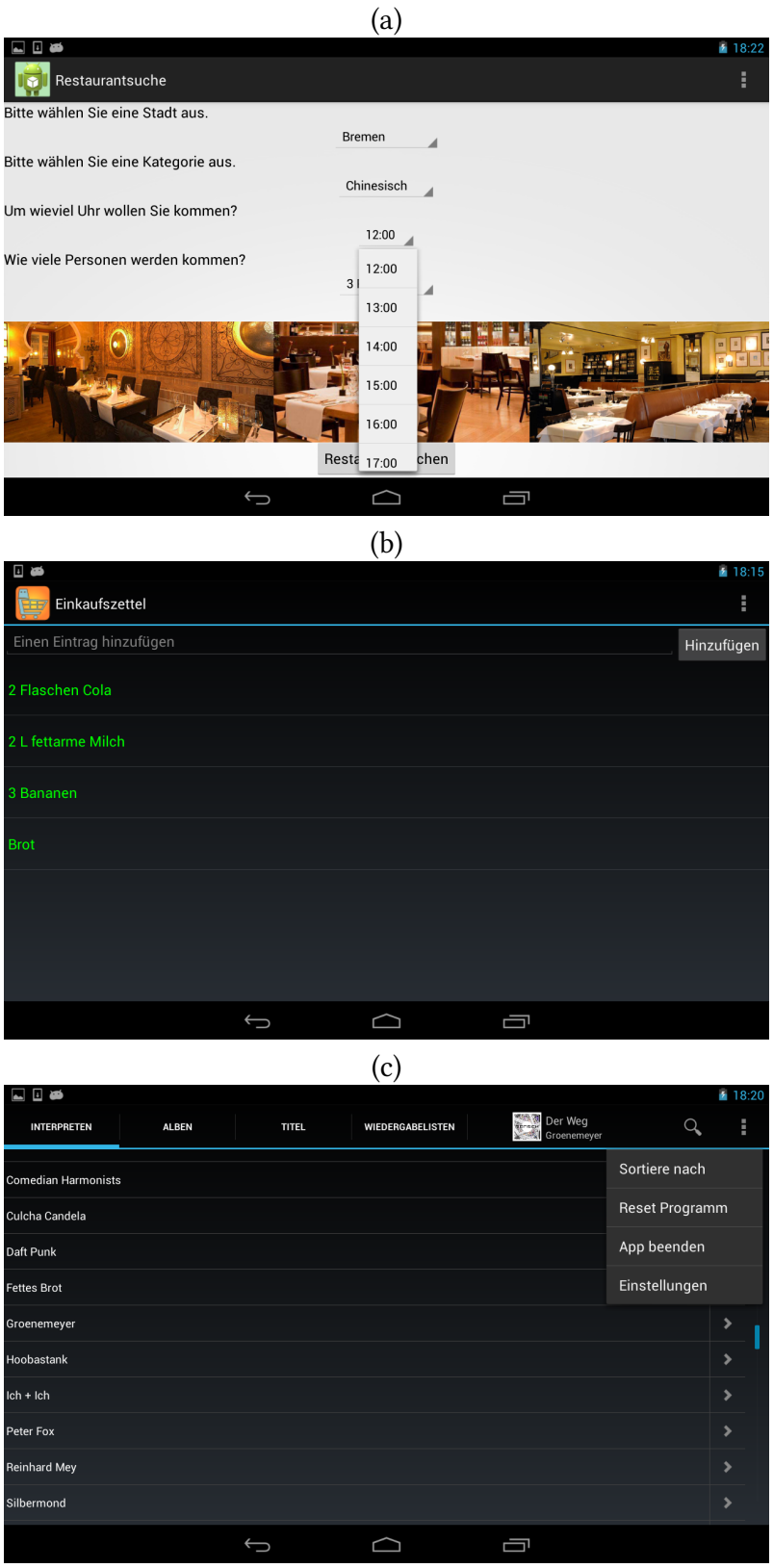


Figure 5.7 Screenshots of the apps used in the second experiment: ReSA 2.0 (a), Trolly (b) and Vanilla Music Player (c). See Table 5.5 for translations.

German	English translations
Bitte wählen Sie eine Stadt aus	*Please, select a city*
Bitte wählen Sie eine Kategorie aus	*Please, select a category*
Um wieviel Uhr wollen Sie kommen?	*What time do you want to come?*
Wie viele Personen werden kommen?	*How many persons will come?*
2 Flaschen Cola	*2 bottles of coke*
2 L fettarme Milch	*2 litre skimmed milk*
3 Bananen	*3 bananas*
Brot	*bread*
einen Eintrag hinzufügen	*add an item*
Einkaufszettel	*shopping list*
Hinzufügen	*add*
Alben	*albums*
App beenden	*close app*
Einstellungen	*settings*
Interpreten	*artists*
Reset Programm	*reset program*
Sortiere nach	*sort by*
Titel	*title*
Wiedergabelisten	*playlists*

Table 5.5 Translations and meanings of German sentences in Figure 5.7 corresponding to ReSA 2.0 (a), Trolly (b) and Vanilla Music Player (c) respectively.

criteria selection. Here we use drop-down lists for selection of search values, instead of additional screens, in order to have a very easily usable app. Figure 5.4 and Figure 5.7 (a) allow a a comparison of both interfaces. The functionalities of both app version are equal, they differ only in the way of selecting search criteria. ReSA 2.0 is the simplest of the three apps in this study, because the user just has to select values from four drop-down lists and to press a button finally (start the search).

Trolly provides a simple shopping list app. The user can add, edit or delete items on the list or mark it as done (e.g., after buying it). All items are contained in a list, which is displayed on the screen. An entry in the options menu allows deletion of all items in the

list. Figure 5.7 (b) shows a screenshot of Trolly with 4 items (green font) in the list. Trolly is also a simple to use app. However, it allows to enter and edit data, which makes this app more complex than ReSA 2.0.

Vanilla Music Player is an app for managing, searching and playing music. The user can search, group and order his music by title of the song, album, artist, or music genre. Furthermore, one can create, edit and delete playlists. Having multiple views and dialogs and a complex underlying data model (artists, albums and titles), Vanilla Music Player is the most complex app among the three described apps. A screenshot of the app is shown in Figure 5.7 (c).

5.5.2.2 Procedure

After describing the general experimental set-up and the groups of participants in the previous section, this section briefly describes the procedure for every participant.

At the beginning, the participant filled out questionnaires about demographic information, as well as technical experience and affinity. This was followed by an introduction into the usage of the tablet device, just in case the participant had no experience with such devices. The introduction mainly explained the usage of the touch screen and the virtual keyboard. Right after the introduction into the device, the participant fulfilled 4 tasks with ReSA 2.0 and Trolly and another 3 with Vanilla Music Player. Examples of the tasks are:

***ReSA 2.0**: Please search and make a reservation at a Chinese restaurant in "Bremen" for 3 people at "12:00".*

***Trolly**: Please change the "2 l Milch" item to "2 l fettarme Milch" , add "2 Flaschen Cola" and "2kg Äpfel" to the shopping list.*

***Vanilla Music Player**:*

1. *Please search the album "Mensch" from the interpret "Grönemeyer".*
2. *How many titles does this album contain?*
3. *Please start playing "Der Weg".*

The order of apps was randomized per user, but not the order of the tasks, since they partially build up on each other (e.g., items were added to the shopping list and then edited). After each task (altogether 11 times) the participant answered an "AttrakDiff mini" questionnaire [53], among others. Detailed information on the procedure, especially about the other questionnaires to be answered, are given in [73].

5.5.2.3 Results

In the following we show what interaction parameters could be automatically logged and computed through the usage of PALADIN. Additionally, the main finding of the study is reported without to got lost in the details that are not in the focus of our article.

Each participant fulfilled eleven tasks (the execution of one task is called an interaction in the remaining paragraphs of this section). The logged interactions (Table 5.2 shows the logged parameters) were represented in persisted PALADIN instances, from which it was easy to compute the following parameters ([73, Table 1, p. 201]) for each interaction:

- *average* values of user turn duration, system turn duration user feedback delay, system feedback delay, user action duration, system action duration, number of system concepts, user text elements, system elements, user elements
- *maximum* values of system feedback delay, user feedback delay, user elements
- *number* of transferring interactions, interaction duration, of interaction steps

The listed parameters were used to compute the following five interaction characteristics for the each task and participant: *interaction efficiency* (e.g., dialog duration), *cognitive efforts* (e.g., user feedback delay), *executing efforts*, *application complexity* (e.g., number of system elements) and *input efficiency* (e.g., action stage duration). Gong and Engelbrecht used the term “interaction parameters” instead of “interaction characteristics”, but we used the latter here, in order to distinguish between our model and theirs. They computed the correlation between the participant’s judgements (on AttrakDiff mini after each task) and the computed interaction characteristics. It could be shown that such correlations exist for $80.6\ \%$ of the participants.

5.6 Discussion

This section discusses whether PALADIN is able to answer the research questions posed in this chapter as well as its practical application in real evaluation scenarios. Furthermore, PALADIN is evaluated and compared to other approaches according to the guidelines proposed in [57]. The main limitations when instrumenting user-system interaction are analyzed in the last subsection.

5.6.1 Research Questions

This article posed three research questions according to some problems found in current state-of-the-art. These questions have been answered throughout the chapter. The results are summarized and discussed in the following.

Q1 asked for the possibility to evaluate different modalities at the same level. For this purpose, PALADIN uses two different levels of abstraction (i.e., generic and specific) to describe multimodal interaction, which helps to maximize the number of common interaction parameters.

Generic parameters are suitable for different modalities, so they are used to describe multimodal interaction regardless of the modality in use. Interaction is described seamless, i.e., communication content and meta-data are described as a whole without differentiating between different communication "threads", each one using a different modality. As a result, PALADIN puts all different modalities at the same level and describes communication as a single stream of information between the user and the system.

Additionally, more specific parameters were added to describe input and output peculiarities of each modality. As shown in Section 5.2.3, these particular parameters are used to provide additional information when more than a generic description of the communication content is needed for the analysis of interaction.

Q2 posed to find a way to compare different interaction records. PALADIN uses the same metrics to quantify interaction in different scenarios and structures them within a common representation. This provides experts with unified criteria to describe multimodal interaction. As a result, different interaction processes are recorded into similar model instances (i.e., same format and structure) that can be compared to each other regardless of the execution context in which they were recorded, the system features and the modalities in use.

Nevertheless, to take advantage of this feature it is necessary to use the same criteria when instrumenting the interaction parameters for different modalities. Finding equivalences between the information exchange process in speech, GUI and gesture modalities was an important part of our work. We used our findings to define a unified and balanced criteria to record interaction parameters (see Section 5.5).

Q3 asked for a way to analyze multimodal interaction from a dynamic perspective. For this, the PALADIN design was based on a turn-based nature, which come to be sufficient

to dynamically represent multimodal interaction. System and user turns are considered to be alternate and without overlapping each other (i.e., the model assumes that the user and the system are not providing information at the same time). Several modalities can be used alternatively or in parallel by the system or the user during their turn.

PALADIN instances describe, step by step, the amount of information exchanged between the system and the user as well as the meta-data about the exchange process. As a consequence of this, a relationship between the collected data and time is created. Based on this model, user and system interaction can be easily instrumented at runtime. It also provides the experts with new opportunities for the dynamic analysis of interaction, and enables the implementation of runtime processes like decision or monitoring. Off-line analysis can be implemented as well, since cumulative values can be easily computed from the model instances.

5.6.2 Practical Application of PALADIN

To provide a representation of multimodal interaction that enables the implementation of analysis, comparison and decision processes was another main goal of our research work in this chapter. Section 5.5 shows that the model instances created in the frame of our experiments are valid to implement such processes.

A graphical analysis tool was used to create abstract representations from data stored in PALADIN instances. Such representations (remember Figure 5.5) are charts representing the "interaction stream" of a dialog, which are used to implement agile analysis and comparison processes. We showed with our first user study that reliable conclusions can be easily made just by taking a look to the interaction charts. However, for a deeper analysis in which a higher number of parameters are involved, more than such a tool is needed.

Decisions based on model instances were implemented as well. ReSA 1.0 (see Subsection 5.5.1.1) was provided with a decision module, which was aimed at making suggestions to the user when low efficiency in the input method was detected. With this decision module it was shown that the data stored in PALADIN instances can be easily retrieved and then computed in runtime to make new decisions. This provides a common base for those systems that adapt themselves to improve their usability at runtime.

The experiments described in Section 5.5 showed that PALADIN (as well as the *Android HCI Extractor* and the Integration Framework) can be easily integrated and successfully

used in typical apps for daily use. This is mainly thanks to the open and generic approach used for its design and implementation. Furthermore, the work in this chapter aimed at "standardizing" the description of multimodal interaction, which enables PALADIN instances to be used for further analysis in different studies and/or by third-party tools. PALADIN is designed to be used separate or as a complement to other techniques such as questionnaires or structured interviews, in order to increase the productivity in user studies. However, PALADIN is not aimed to completely substitute them in the whole usability engineering life cycle.

5.6.3 Completeness of PALADIN According to Evaluation Guidelines

Dumas et al. introduced in [57] a set of nine guidelines for languages dedicated at multimodal interaction. This nine guidelines "are to be seen as a 'checklist' of potential features a given multimodal interaction description language can provide. By no means should every language follow all of them. Guidelines should be used as design tools, or as language analysis criterias [sic]." [57, p. 4].

The guidelines from G1 to G8 are used to evaluate PALADIN, as well as to compare it to a set of representative languages and data models describing multimodal interaction. The 9th guideline G* ("Find the right balance between usability and expressiveness") is not considered in our work because we were not able to analyze the usability of all named languages and models. In the following we explain whether and why PALADIN fulfils these eight guidelines.

G1 As discussed above for Q1, the proposed model uses two different levels of abstraction to separate specific details of each modality from the generic description of multimodal interaction, and thus ease analysis.

G2 Based on these two levels of abstraction, the human-machine dialog is modeled as a sequence of alternative system and user turns. Mention here that the proposed approach considers only one user interacting with the system at a time.

G3 The model contains modality related parameters to describe kind and reason of a modality change, i.e., SMCD, OMCR, UMCD and IMCR. Reasoning modality changes could be helpful in understanding context adaption.

G4 Modality fusion parameters are also included in the proposed design. Different combinations of the available modalities are supported by the model, as well as special parameters to describe their synchronicity. The parameter *modality type* (MT) (described in Table 5.10) is used to annotate how different modalities were used according to CARE. The MT parameter allows to annotate the usage of fusion and/or fission. Thus, in case of fusion the modalities are marked as used complementary (MT-CO). Otherwise, in case of fission the exclusively processed modality is marked as assigned (MT-AS).

G5 The CARE properties [50] are used to denote the availability of interaction techniques, and also to predict usability of multimodal input and output. Beside the *modality type* (see also G4) and *number of asynchronous events (# AE)*, the user and system interaction duration —divided into delay, feedback and action stages— is annotated by turn to link actions with the moment in which they are performed.

G6 Interaction and recognition errors have also a place in the model. Meta-communication data is used to describe communication errors, and they are annotated by turn to link such errors to the communication context in which they occurred. However, this is not the case for system events.

G7 An event description is not included into the model because the aim of the proposed design is not to describe the "How" of the interaction (e.g., by modeling use cases), but to describe the "What" (i.e., by quantifying system and user actions).

G8 For each system and user turn output and input content are relatively quantified. Such quantification process is performed by modality; then, a generic description of communication content is built from the values obtained at each modality. Moreover, such parameters are collected on a turn level, which creates a relationship between different data annotated in the same interaction step.

Table 5.6 shows these guidelines confronted with the approach proposed in this chapter. The table also compares PALADIN to some representative approaches previously analyzed in Section 2.2.1. They were selected due to the following reasons. ITU-T Suppl. 25 to P-Series Rec. [168] are part of the basis for the parameters used in PALADIN, thus we consider important to know the improvements achieved with the proposed model. EMMA [99, 118] represents a language in which data from different modalities are combined to make up the user input. Much of the work in PALADIN aims at providing a seamless representation of multimodal input as well. Since we see interesting to compare PALADIN to other approaches using models of a different nature, it is also compared to two model-based approaches for

Guideline	PALADIN	ITU-T	EMMA	ICO	SOMBDE
G1: Uses different abstraction levels	■	■	□	□	■
G2: Human-machine dialog is modeled	■	□	⊞	■	⊞
G3: Adaptability to context and user	⊞	□	⊞	□	□
G4: Modality fusion support	■	⊞	■	■	■
G5: Over time synchronicity support	■	⊞	⊞	■	□
G6: Error and recognition mistakes support	■	■	□	■	□
G7: Events description and management	□	□	□	■	⊞
G8: Input and output representation	■	■	■	⊞	⊞

■ *yes* ⊞ *partially* □ *no.*

Table 5.6 Comparison of different approaches for multimodal interaction by supported guidelines.

the development of multimodal interfaces. On the one hand the ICO notation [183], that uses formal models to describe multimodal interaction. On the other hand the solution proposed by [129] that we call SOMBDE, and that models components of an interface using two levels of abstraction.

At first sight, the reader can see in Table 5.6 that Paladin significantly complies more Dumas guidelines than approaches like ITU-T, EMMA or SOMBDE. The main reason is the time-based dialog approach for interaction quantification that is used in PALADIN. Modality meta-information (i.e., fusion, changes and description over time) is more comprehensive than in the ITU-T parameter-based approach. Like in SOMBDE, PALADIN provides two levels of abstraction to evaluate interaction, using the more specific one for the particularities of each modality. This is a lack in approaches like EMMA and ICO.

Plasticity (i.e., adaptability to context and user) is better in PALADIN than in other approaches, however it is still a problem. User and context information —which is essential to analyze interaction in mobile contexts— might be incorporated to enrich analysis processes. Not describing how the multimodal system works could be considered another shortcoming of the proposed design. It is easier to find in model-based development approaches (ICO and SOMBDE) than in evaluation approaches like PALADIN or ITU-T. Despite modeling the behavior of user and system was not a goal of this chapter, we consider that including additional data about this aspect might be helpful to detect design and operation errors.

5.6.4 Limitations in Automatic Logging of Interactions Parameters

The design of PALADIN was accompanied with instrumentation tools mainly aimed at providing automatic logging of interaction parameters. However, certainly not all of these parameters can be computed automatically. In the following we identify and analyze three of the main limitations when annotating interaction parameters automatically, which imply that some PALADIN parameters need the presence of an expert to be annotated.

First, some parameters are in principle not automatically determinable. Because of their nature, some parameters can not be determined by a computer, but they have to be annotated manually by a human (e.g., the number of concepts a real user wanted to utter in speech modality). There are also parameters whose determination depend on a subjective/qualitative judgement of the user (e.g., reason of changing the modality in use) or the knowledge of the expert conducting the evaluation (e.g., openness of a prompt). A method to identify some reasons for users' behavior during an interaction with a computer system (i.e., a smart home system) is presented in [197]. Schmidt et al. used in-depth interviews supported by video feedback.

Furthermore, when collecting and analyzing interaction data, experts should be aware that system and user turns are not discrete; even some of their stages might not be present in specific situations. For example, the delay stage in a system turn may be imperceptible to the user, or there may be no feedback stage at all or during user turn. It might be impossible to distinguish between the end of the delay stage and the start of the exploring stage, if the exploring action involves actions such as reading, which can only be assessed through observation or eye-tracking.

Finally, the interaction instrumentation and annotation process depends also on the concrete implementation of the application in which PALADIN is used. Not all runtime environments offer the possibility to collect all of the parameters programmatically, e.g., to precisely count the number of visible elements in a GUI.

5.7 Conclusions

Current approaches for multimodal interaction analysis do not facilitate the implementation of stepwise and seamless evaluation. Most common problems are the lack of a dynamic

nature and that different modalities are treated at different levels of abstraction. In this context, this chapter proposes PALADIN, a model aimed at describing multimodal interaction beyond such problems.

The proposed model design is based on a set of parameters to quantify the interaction between the user and the system. These parameters are annotated in a stepwise manner to preserve the dynamic nature of the dialog process. Generic parameters are used to describe the dialog content as a single interaction flow. Specific parameters are used to annotate peculiarities of GUI and speech modalities. The current version of PALADIN annotates gestures by using only generic parameters. In a short term we would like to analyze gesture inputs and outputs in depth, and thus provide specific parameters for this modality as well.

As a result, instances of PALADIN can be used as a uniform basis to describe and analyze interaction in different multimodal scenarios, as well as to evaluate the usability of such systems. To the best of our knowledge, PALADIN is the first approach structuring multimodal interaction parameters into a runtime model design with this purpose.

The implementation of PALADIN along with the integration framework are provided as a contribution to the open-source community. This makes up a framework to evaluate interaction in multimodal systems ready to be incorporated into developments. As proof of concept, this evaluation framework has been incorporated into real applications to conduct two different experiments with users. One experiment was aimed at showing its validity for the implementation of analysis, comparison, and real-time decision processes. The other experiment used PALADIN to conduct a users study aimed at determining several system and user characteristics, and their relation to user judgments.

5.8 Parameters Used in PALADIN

The tables in this section give an overview about all parameters which are modified or newly introduce in PALADIN compared to ITU-T Suppl. 25 to P-Series Rec. [168]. Table 5.7 provides an index containing each parameters (by its abbreviation) and the table or reference describing it. Table 5.8 explains the abbreviations which are used in the subsequent tables.

Parameter	Table	Parameter	Table	Parameter	Table	Parameter	Table
#AE	*	FPST	5.9	OMCR	5.10	#system turns	*
AN:...	*	FPUT	5.9	PA:...	*	TAC	5.12
%AN:...	*	#GR rejection	*	%PA:...	*	TAR	5.12
#ASR rejection	5.11	#help request	*	QD	*	#time-out	*
#barge in	5.11	IMA:...	*	#restart	5.11	TS	*
CA	*	%IMA:...	*	RME	*	#turns	*
CA:...	*	IMCR	5.10	SA	*	UA	*
%CA:...	*	IR	*	SAD	5.9	UAD	5.9
#cancel	5.11	κ	*	SCR	*	UCR	*
CE	*	KUP	5.12	#SCT	*	#UCT	*
CER	*	LT	*	SER	*	UEDu	5.9
CPST	5.9	MMF	5.12	SFD	*	UFD	5.9
CPUT	5.9	MML	5.12	SFDu	5.9	#UMC	*
$DARPA_{me}$	*	MS	*	#SMC	*	UMCD	5.10
$DARPA_{s}$	*	MT	5.10	SMCD	5.10	URD	5.9
DD	*	MUP	5.12	SRD	*	#user questions	*
#DIV rejection	5.11	NES	*	STD	*	#user turns	*
#EAC	5.12	NPST	5.9	SuBR	5.11	UTD	*
EAR	5.12	NPUT	5.9	#system error	*	WA	*
EPST	*	OMA:...	*	#system help	*	WER	*
EPUT	*	%OMA:...	*	#system questions	*	WES	*

Table 5.7 Index of parameters ordered alphabetically (leading % and # are ignored) and the tables containing those. The * refers to [168].

Abbreviation	Full name	Values
Abbr.	Abbreviation	Explained in the tables
Mod.	Modalities	*S* – Speech, *V* – Visual, *G* – Gesture
Int. lev.	Interaction level	*D* – Dialog, *SoD* – Set of dialogs, *T* – Turn, *U* – Utterance, *W* – Word
Meas. meth.	Measurement method	*E* – Expert, *I* – Instrumentally

Table 5.8 Glossary of abbreviations used in Table 5.9 up to Table 5.12.

Parameter				Int.	Meas.
Abbr.	**Name**	**Description**	**Mod.**	**lev.**	**meth.**
SFDu	system feedback duration	Duration of system feedback in, in [ms]. Examples: *Speech*: Time from the beginning of the feedback utterance until the beginning the action utterance. *GUI*: Time from the beginning of the feedback to the beginning of information presentation. *Gesture*: Time from the beginning of the feedback gesture performance to the beginning of the action gesture performance.	S,V,G	T	I
SAD	system action duration	Duration of system action, in [ms]. Examples: *Speech*: Time the system needs to utter the concrete system answer. *GUI*: Time the system needs to load/-draw the entire GUI. *Gesture*: Time the system needs to perform the gestures representing the concrete system answer.	S,V,G	T	I
URD	user response delay	Delay of user response, from the end of system output to the moment the user starts doing data transferring actions, in [ms]. Examples: *Speech*: The user starts the action when the user utterance starts. *GUI*: The user starts the action when he/she starts providing information to the system. *Gesture*: The user starts his/her action when the gesture starts being performed.	S,V,G	T	I,E

UFD	user feedback delay	Delay of user feedback, from the end of system input until the user starts providing feedback or doing exploring actions, in [ms]. Examples: *Speech*: User feedback starts just when the user starts saying a feedback utterance. *GUI*: User feedback starts just when the user starts exploring the GUI. *Gesture*: User feedback starts just when the user performs a feedback gesture.	S,V,G	T	I,E
UEDu	user exploring duration	Duration of user feedback/exploring stage, from the user starts doing feedback-/exploration actions until he/she starts providing the system with data, in [ms]. Examples: *Speech*: Time from the beginning of the feedback utterance until the beginning of the action utterance. *GUI*: Time during which the user scrolls the screen content to the moment he/she clicks an item in the screen. *Gesture*: Time from the beginning of the feedback gesture performance to the beginning of the action gesture performance.	S,V,G	T	I
UAD	user action duration	Duration of user action, from the user starts providing the system with new information until final data submission. Examples: *Speech*: Action duration corresponds to the user utterance duration. *GUI*: Action duration corresponds to the time the user is manipulating the graphical elements of a GUI to provide the system with new information. *Gesture*: Action duration corresponds to the time the user needs to perform the gesture.	S,V,G	T	I

CPST	concepts per system turn	Average number of semantic units (each represented by an Attribute Value Pair) per system turn, in a dialog.	S,V,G	T	I,E
FPST	feedback per system turn	Number of feedback elements per system turn in a dialog. Feedback refers to the information that one party taking part in a dialog sends to the other party to inform about the state of the process. It allows dialog partners to seek and provide evidence about the success of their interaction. Examples: *Speech*: An utterance saying that the process was done successfully. *GUI*: A confirmation window reporting an error. *Gesture*: A embodied agent performs an "OK" gesture to inform that the user input was properly understood.	S,V,G	T	I,E
NPST	noise per system turn	Number of "disturbing" elements per system turn in a dialog. Noise refers to those data which are irrelevant, meaningless, or disturbing, and are not needed to reach the goal of the dialog: advertisements, music played in background, etc. Pleasant information are not considered as noise, since they are part of the communication between two or more parties. Examples: *Speech*: Music played in background while the system is uttering. *GUI*: Advertisement banners inside the content of a web page. *Gestures*: A embodied agent points to an external advertisement at the beginning of its gesture.	S,V,G	T	I,E
CPUT	concepts per user turn	Number of semantic units (each represented by an Attribute Value Pair) in a user turn.	S,V,G	T	E

FPUT	feedback per user turn	Number of feedback elements provided by the user to the system. User feedback refers to the information that the user sends to the system to inform about the state of the interaction or to denote that he/she is analyzing the information provided by the system and elaborating a response. Examples: *Speech*: The user utters "Hmmm" while elaborating his/her response. *GUI*: The user is scrolling down and up the content of a web page to read the text. *Gestures*: The user is showing that he/she is thinking about the system question by using his/her face expression.	S,V,G	T	E
NPUT	noise per user turn	Number of "disturbing" elements provided by the user to the system. Noise represents data which may disturb the recognition process performed by the system. Examples: *Speech*: Off-talk, e.g., the user reads/repeats aloud information provided by the system. *Gestures*: Movements by the user which are interpreted as gestures, but just being spontaneous actions, e.g., scratching.	S,G	T	E

Table 5.9 Dialog and communication-related interaction parameters.

Parameter				Int.	Meas.
Abbr.	Name	Description	Mod.	lev.	meth.
MT	modality type	This parameter describes the type of the modality according to the CARE properties described in [50]. These properties represent a simple way of characterizing aspects of multimodal interaction considering the interaction techniques available in a multimodal user interface. In that way, the CARE properties characterize four types of relationships between the modalities used to achieve a goal or to reach a concrete state: *UM*: Unimodal (not part of CARE). *EQ*: Equivalent, it is sufficient to use any one of the available modalities. *AS*: Assigned, the user has no choice, because only one modality can be used. *RE*: Redundant, the available modalities are equivalent and all of them are used within a user or system turn. *CO*: Complementary, all available modalities must be used in a complementary within a user or system turn.	S,V,G	T	I,E
SMCD	system modality change direction	Label of system modality task direction, depending on the modalities the system has switched between. The label is generated from the pattern *SCMD:X-Y*. *X* and *Y* are substituted by S (Speech), V(Visual) or G (Gesture) according to the respective modality. *X* represent the used modality before the change, and *Y* after the change.	S,V,G	T	I

OMCR	output modality change reason	Label of the reason for the output modality change. *OMCR:ERR*: Due to a recognition error. *OMCR:ENV*: Due to an environment change. *OMCR:APP*: Due to low modality appropriateness. *OMCR:UDE*: Due to user's decision. *OMCR:SDE*: Due to system's decision. *OMCR:IAD*: Due to interface adaptation.	S,V,G	T	I,E
UMCD	user modality change direction	Label of user modality task direction, depending on the modalities the user has switched between. The label is generated from the pattern *UCMD:X-Y*. *X* and *Y* are substituted by S (Speech), V(Visual) or G (Gesture) according to the respective modality. *X* represent the used modality before the change, and *Y* after the change.	S,V,G	T	I
IMCR	input modality change reason	Label of the reason for the input modality change. *IMCR:ERR*: Due to a recognition error. *IMCR:ENV*: Due to an environment change. *IMCR:APP*: Due to low modality appropriateness. *IMCR:UDE*: Due to user's decision. *IMCR:SDE*: Due to system's decision. *IMCR:IAD*: Due to interface adaptation.	S,V,G	T	E

Table 5.10 Modality-related interaction parameters.

Parameter				Int.	Meas.
Abbr.	Name	Description	Mod.	lev.	meth.
#DIV rejection	number of data input validation rejections	Overall number of DIV rejections in a dialog. A DIV (Data Input Validation) rejection is defined as a system feedback indicating that the data provided by the user was not "understandable" for the system. Examples: *GUI*: The user enters characters into a numeric field.	V	T	I
SuBR	successful user barge-in rate	The percentage of these barge-in attempts in which the user obtained a positive result: The system stops its action and processes the user's input.	S,V,G	T	E
#cancel	number of users cancel attempts	The parameter has NOT the meaning of #cancel in [168] (please see also #restart in this table). Our usage of #cancel is the following: Overall number of user cancel attempts in a set of dialogs. A user turn is classified as a cancel attempt, if the user gives up the dialog and does not accomplish the task.	S,V,G	D	E
#restart	number of user restart attempts	The meaning of #restart is equal to #cancel in [168].	S,V,G	T	E

Table 5.11 Meta-communication-related interaction parameters.

Parameter				Int.	Meas.
Abbr.	Name	Description	Mod.	lev.	meth.
KUP	keyboard usage percentage	Average percentage/rate of keyboard usage during the user turn duration. Keyboard usage is measured in terms of elements introduced by the user, e.g., words, special keystrokes. See also *EPUT* in Table 5.9.	V	T	I
MUP	mouse usage percentage	Average percentage of mouse usage during the user turn duration. Mouse usage is measured in terms of elements introduced by the user, e.g., clicks, mouse wheel performance, mouse movements. See also *EPUT* in Table 5.9.	V	T	I
MML	mouse movement length	Average length of the mouse movements performed in a user turn in pixels[px], per dialog.	V	T	I
MMF	mouse move frequency	Average number of mouse movements performed in a turn, per dialog.	V	T	I
#EAC, EAR	number of exploring actions, exploring actions rate	Overall number (*#EAC*) or percentage (*EAR*) of exploring actions made by the user in a dialog. This parameter is measured in terms of "exploring" actions performed by the user, e.g., mouse wheel, down-key press, etc.	V	D	I
#TAC, TAR	number of transferring actions, transferring actions rate	Overall number (*#TAC*) or percentage (*TAR*) of transferring actions made by the user in a dialog. This parameter is measured in terms of "information transferring" actions performed by the user, e.g., mouse click, alphanumeric keystrokes, etc.	V	D	I

Table 5.12 Keyboard- and mouse-input-related interaction parameters.

6

Modeling and Evaluating Mobile Quality of Experience

Quality of Experience (QoE) is a subjective measure of users experiences with a service that encompasses users behavioral, cognitive, and psychological states. The analysis of the surrounding context is critical in the evaluation of QoE in mobile environments, in which users and their handheld devices are continuously moving in several simultaneous fuzzy contexts.

The approach in Chapter 5 is mostly targeting the quality of the multimodal dialog. In order to achieve the quality of the whole interaction experience, this chapter describes CARIM, a model arranging parameters to describe the interaction between the user and the system, the context in which it is performed, the usability perceived by the users, and their mood and attitude toward technology use.

CARIM supports the stepwise analysis of mobile interaction to determine and compare the QoE of users using a system or a software. Applications can also make use of its runtime nature to make context- and QoE-based decisions in real-time to adapt themselves, and thus provide a better experience to users. CARIM provides unified criteria for the inference and analysis of QoE in mobile scenarios.

6.1 Introduction and Motivation

Quality of Experience (QoE) is a subjective measure of users experiences with a service. It rivets on the true feelings of end users from their perspective when they do an activity [169, 227, 41, 165] and encompasses users behavioral, cognitive, and psychological states, along with the context in which the services are provided to them. The context is particularly relevant in mobile environments, where applications are used in different, more dynamic, and social scenarios [165].

Mobile interaction involves users and their handheld devices continuously moving in several simultaneous fuzzy contexts. This dynamic environment, which has become more complex in the last few years [88], sets special requirements for the quality assessment of mobile applications. A close relationship between interaction, its context, and QoE can be found in these scenarios. The lack of a uniform approach for modeling mobile interaction in a specific context is evident [29].

The context plays a critical role in the users experience (UX) with mobile products [112]. Moreover, context-awareness is a core function in modern ubiquitous and mobile systems [21] in which the surrounding information is analyzed at runtime to adapt the functionality of applications [31] and thus providing natural and intelligent interaction to users [125]. By context information we mean any data used to characterize the situation of an entity (i.e., person, place, or object) and that is considered relevant for user-system interaction analysis.

A challenge for designers and researchers studying mobile applications is that no robust methodologies combining qualitative methods for usability assessment and quantitative methods evaluating performance do exist [91]. This is why this work proposes incorporating context information and user ratings into a user-system interaction assessment method. However, integrating all these data into such a process is not straightforward.

There exists a low standardization of technologies used in context-aware systems [88]. A common and extensible representation of the mobile context is needed to ease such analysis processes. Another problem is to treat the variety and diversity of interaction and context data, as well as to decide what parameters are useful to measure QoE in mobile contexts [165]. A well balanced set of parameters not aimed at modeling “the entire world” has to be chosen to describe the whole mobile interaction process.

We consider essential that mobile interaction data and user ratings are collected by using current devices capabilities. How these data are integrated into an interaction analysis method is also a problem to solve [91]. This method should enable the assessment of QoE from the perspectives of the user-system interaction, its surrounding context, and the quality perception of the users. It should also support the implementation of analysis and decision processes, as well as to allow the cooperation between different analysis applications.

According to the aforementioned problems, the following research questions are posed:

> **Q1**: How can context data and user perceived quality information be properly incorporated into interaction analysis processes?
> **Q2**: How can QoE be analyzed from the perspectives of users interaction, users subjective data, and the interaction context?
> **Q3**: How can interaction experiences recorded for different users, or for different systems or contexts, be compared to each other?

In a first step towards answering these questions and to overcome some of the limitations present in previous works, a new approach to analyze mobile interaction is described in Section 6.2. This approach proposes a runtime model arranging interaction parameters, which is augmented with information about the surrounding context (Section 6.3) and with parameters describing the users perception of interaction quality (Section 6.4).

The result of the research in this chapter is the design of a new model named CARIM (Context-Aware and Ratings Interaction Model), presented in Section 6.5. Some considerations about its implementation are described next in Section 6.6. Instances of the CARIM model provide a basis to determine and compare QoE, allowing experts to identify those aspects contributing to the user having a good or bad impression of the system [222]. These instances provide also a basis to make runtime adaptation decisions into an application with the aim of providing a better interaction experience to users.

Section 6.7 describes an experiment with real users carried out as proof of concept for this approach. It compares two different interaction designs and validates some user behavior hypotheses. After that, the proposed solution is discussed and compared to other representative analysis approaches in Section 6.8. Section 6.9 includes some conclusions.

6.2 Context- and QoE-aware Interaction Analysis

This approach tries to give an answer to the three research questions posed above. For this purpose, the design of a model describing user-system interaction, context data and user ratings is proposed. All these data are structured into a common representation to be the basis for the implementation of QoE analysis and inference processes.

6.2.1 Incorporating Context Information and User Ratings into Interaction Analysis

The proposed design is based on PALADIN [137, 139], a model resulted from a joint effort between the Cátedra SAES [39] and the Telekom Innovation Laboratories [206]. This model was described in detail in Chapter 5.

PALADIN describes interaction dynamically, step by step, and was created to support usability assessment in multimodal systems. Multimodal systems combine several types of sensory modalities for interaction, allowing users to use the most adequate combination in their specific situation, mood, and capability [126]. PALADIN uses general parameters to describe multimodal interaction as a whole, regardless the modalities used. It includes also specific parameters to describe the peculiarities of interaction in speech, GUI, and gesture modalities. As a result, the same metrics are used to describe interaction in different contexts, allowing its analysis and comparison.

However, one problem with PALADIN is that it mainly includes quantitative data related to user-system interaction. Despite interaction parameters are a good indicator of the quality of the evaluated interaction, they do not necessarily provide reliable information about user satisfaction [126]. Therefore, subjective opinions of users are needed to implement a qualitative assessment of interaction [34]. Moreover, PALADIN does not include information about the context of interaction, that is necessary to evaluate mobile scenarios. Therefore, it was extended in two ways.

On the one hand, the base model was added with new parameters to describe the interaction context in mobile scenarios. These context parameters are further described in Section 6.3. On the other hand, new parameters were incorporated into the model to measure the quality perceived by users of the product under test. These subjective parameters are described in detail in Section 6.4.

Figure 6.1 depicts an overview of the metrics included in the design of CARIM compared to the base model PALADIN. As a result, an instance of the resulting model includes live context information related to the interaction, providing also a link to the user ratings. This tries to give an answer to research question **Q1**.

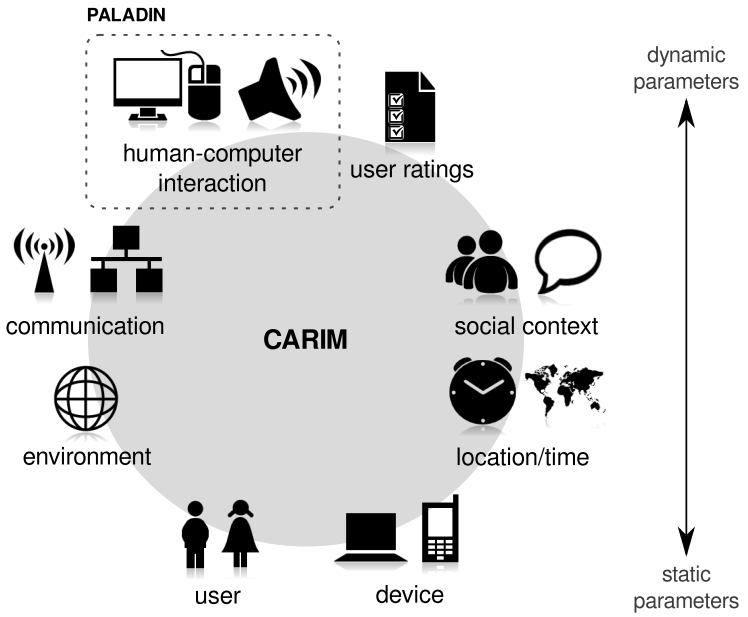


Figure 6.1 Overview of the parameters and metrics included in CARIM.

The resulting model is based on a set of parameters divided into three main categories:

- **Human-computer interaction** parameters, used to quantify the interaction between the user and the system (e.g., quantity of information provided by the system, average reaction time of the user).
- **Context** parameters, aimed at describing the changing context of interaction in mobile scenarios (e.g., screen size, user location, user current mobility).
- **User rating** parameters, used to measure the perceived quality and the experiences of the users with the system under test (e.g., simplicity of the system, motivating aspects of the product).

6.2.2 Arranging the Parameters for the Analysis of Mobile Experiences

All the parameters in CARIM are thoroughly arranged to ease further analysis, as well as to help applications and experts to understand why such a certain level of QoE was inferred from the perspectives of interaction, its context and the user state. The model not only provides a link between interaction data and time, but new links between interaction and, e.g., the current user location and social context, the device features or event the user impressions, are created.

Therefore, once QoE is determined for a specific interaction occurrence (i.e., a model instance) the results can be analyzed and interpreted in terms of what the user did and how he/she did it, in which contextual conditions, and his/her mood and attitudes in that specific moment. This tries to provide an answer for research question **Q2**.

The great majority of the aforementioned parameters are intended to be collected automatically, e.g., by using tools like the Android HCI Extractor (see Section 6.6). Even some of those parameters based on subjective judgments of the user might be extracted automatically by using, e.g., questionnaires. However, technology limitations will force experts to annotate some parameters manually in exceptional cases.

Some of these parameters are runtime and have to be collected many times during interaction, e.g., the quantity of user input at a specific time. Others are not, and are collected only once, e.g., screen resolution. This feature is specially relevant to decide the allocation of the parameters within the model design.

Automatically collected or not, either runtime or static, a uniform set of metrics are used to quantify the interaction between the user and the system in different execution contexts. Furthermore, these metrics are structured into a common representation in the CARIM design. As a result, all the interaction instances will have the same structure, providing experts and tools with unified criteria to describe the mobile interaction process.

Different interaction records can be analyzed and easily compared to each other regardless of the system/application under test, the current interaction context, even the modalities used to provide input and output data. E.g., to detect why QoE worsens when using an application in a different scenario. This tries to give an answer to research question **Q3**.

6.2.3 Using CARIM for QoE Assessment

Instances of CARIM describe user-system mobile interaction using three dimensions: interaction, context, and user ratings. Data are thoroughly arranged and connected to each other providing a robust and accessible basis to implement QoE analysis. QoE inference is the first application of the model that comes to mind. QoE of a user within a specific mobile context can be systematically determined by using the data included into a CARIM instance, e.g., by using Bayesian networks [165].

Moreover, the runtime nature of the model enables QoE inference in real-time. If the resulting QoE value is not the expected at a specific time, the interaction and context history can be analyzed to make a decision and improve the user experience during application execution, e.g., by adjusting microphone settings or changing screen brightness.

From a practical point of view, we consider essential the easy incorporation of CARIM into current mobile devices like tablets, smartphones, etc. The design proposed in this chapter advocates the use of data that can be collected by using current devices capabilities, e.g., the screen to collect touch interaction metrics, GPS to get position, etc. As a result, advanced and expensive sensors are not required to fill the model instances, easing the use of such a model in current applications. It also eases the implementation of context- and QoE-aware methods in real applications, and not only for laboratory environments [88].

6.3 Context Parameters

Understanding the context is essential for evaluating the experience of users using mobile products and services. By *context* we mean any information that can be used to characterize the situation of a user or a system. More specifically, in this work we refer to those external factors influencing users activity and quality perception while using a mobile application or service [210]. This section aims at describing what parameters will be used to describe the mobile context within CARIM.

Before defining the parameters to describe the surrounding context, we need to choose the "subject" of the model (Bolchini et al., [29]). The context may be described from the user point of view, i.e., as it is perceived by the person using the application or service. An alternative would be assuming the application point of view and consider the user itself as part of the context. Using a point of view or another affects the final set of parameters to be

included in the model design. Since in this work the key is analyzing what the user does and in which conditions he/she does it, then the subject of the model will be the user.

6.3.1 Quantifying the Surrounding Context

Section 2.2.3 summarizes related literature. It was analyzed in order to find out those contextual factors affecting the experience of users in mobile environments. Many different parameters were selected from related work. During the selection process we gave priority to those parameters specially aimed at describing mobile communication and dynamic interaction environments. Parameters out of the scope of this work were rejected.

Moreover, new parameters where incorporated in order to overcome some limitations found in the analyzed approaches, e.g., to provide an abstract definition of mobility level or to classify most common sizes of current devices screens.

The resulting set of parameters is organized into six categories (i.e., *Physical context*, *User*, *Social context*, *Location and time*, *Device* and *Connectivity*) based on the eight proposed by Korhonen et al. in [112]. *Task* and *Service* categories were out of the scope of this work. The main reason is that we want to provide a generic description of the mobile context regardless of the task the user is currently engaged in and of the specific online service he/she is using. These six context dimensions are further described below.

1. **Physical context**: describes surrounding attributes that a user can sense, e.g., lighting, temperature, noise level, weather conditions, etc.
2. **User**: describes peculiarities of the person using the device and the application or service, e.g., gender and age, or her previous experience with the device and/or application to test.
3. **Social context**: describes the social aspects of the user context that may affect user experience. Particularly, in this work we focus on whether the user is alone or accompanied by some people during the test, as well as in which social arena the user is (e.g., workplace, leisure).

   It is known that the presence and reactions of external persons —in direct contact with the user or not— affects both the usage and user experience of mobile applications [225].

It is known that the presence and reactions of external persons —in direct contact with the user or not— affects both the usage and user experience of mobile applications [225].

4. **Location and time**: describes the position of the user while interacting the system (e.g., the user is at home) as well as a time stamp for it. This category is also augmented with information about the mobility level of the user, i.e., if the user is sitting, standing, walking, driving, etc.

From Roto [193] and Korhonen [112] we learned that time and location does not directly affect the user experience. However, these parameters affect other attributes, and these attributes then affect user experience (e.g., the location itself does not affect the user experience, but the artifacts and people over there).

5. **Device**: describes the peculiarities of the device in use. A device is described in terms of type and shape, screen features, internal values like volume level and brightness, and CPU and memory performance.

Information related to the available modalities or the input and output methods is avoided, because it is implicit into the human-computer interaction section of the model inherited from PALADIN (see Subsection 6.5.1).

6. **Connectivity**: describes the features of the communication between the device and online services. This communication is described in terms of type of connection (i.e., cellular, wireless LAN or Bluetooth) as well as its performance (i.e., coverage, sent/received data throughput, etc.)

This category is mainly based on the aforementioned user perceived quality of service (QoS). QoS is critical to the users QoE, especially for highly interactive mobile applications [91].

Table 6.1 summarizes all the context parameters to be incorporated into the design of CARIM, as well as the values they can hold during the annotation process. The parameters are also tagged according to whether they have a dynamic (D), mainly dynamic (MD), mainly static (MS) or static (S) nature. After some experiments, we consider that this set of parameters is enough to describe the context surrounding the users when doing an activity or using a service from their mobile devices. Nevertheless, new parameters can be added or removed whenever required (e.g., to add a new category or to enhance an existing one).

To provide an adequate level of abstraction for the context parameters to find a balance between precision and amount of data was a major concern in this work. To give an example we can analyze the attribute *social company* (Table 6.1, *Social context*). The number of persons accompanying the user could have been considered as an integer value. However, an enumeration is used instead in order to characterize only three different company

Context Dimension	Attributes	Nature	Values
Physical context	temperature	MS	integer (°C)
	weather	MS	clear, cloudy, windy, rainy, snowy
	noise	MS	percentage (%)
	light	MS	percentage (%)
User	age	S	integer
	gender	S	male, female, other
	education level	S	high school, professional, college, not applicable
	previous experience	S	none, low, medium, high, expert
Social context	social company	MS	alone, with a person, with a group
	social arena	MS	domestic, work, educational, leisure
Location/time	location	MS	home, office/school, street, other indoor, other outdoor
	geographical location	MD	coordinate
	mobility level	MS	sitting, standing, walking, sporting, driving, other
	current time	D	time value
Device	device type	S	laptop, tablet, smartphone, mmplayer, other
	screen size	S	small $\leq$ A $<$ medium $<$ B $\leq$ large (A=4", B=10)"
	screen resolution	S	small $\leq$ A $<$ medium $<$ B $\leq$ large (A=480x800, B=1280x800)
	screen orientation	MS	landscape, portrait
	screen brightness level	MS	percentage (%)
	volume level	MS	percentage (%)
	memory usage	D	percentage (%)
	CPU usage	D	percentage (%)
Connectivity	wireless access type	MS	mobile, wifi, bluetooth, no access
	access point name	MS	string
	signal strength	MD	percentage (%)
	received data throughput	MS	integer (KB/s)
	sent data throughput	MS	integer (KB/s)
	Round Trip Time (RTT)	MS	integer (ms)
	Server Response Time (SRT)	MS	integer (ms)

Table 6.1 Parameters used to describe the mobile context in CARIM.

scenarios, i.e., alone, with a person or with a group. In such a way precision is lower —the specific number of persons is unknown— but the task of quantifying and analyzing this information is simpler due to more abstract values are used.

6.3.2 Arranging Context Parameters into CARIM

Context parameters have properties that affect how and when the data have to be collected, and that directly guide the final design of the model described in Section 6.5. Their dynamic or static nature is one of these properties. As the reader can see in Table 6.1, there are not more than seven from twenty-nine parameters we can claim they will not change during an experiment. Static context parameters like those related to hardware will not change, so they have to be annotated only once. However, dynamic parameters that change over time (e.g., screen orientation, noise level) have to be collected many times.

6.4 User Perceived Quality Parameters

User perceived quality has also to be incorporated into the proposed model design. Perceived quality can be measured by collecting user ratings when using the application or service under test. As defined in [91], user ratings are QoE purely subjective, episodic assessments provided on the basis of the given perception of the specific episode of application use.

Relying on this definition, a set of parameters collecting user impressions and feelings (i.e., QoE assessments) will be used after or at any moment during the interaction process. These parameters will denote the perceived attractiveness of the product and the experience of using it. Moreover, some other parameters are used to describe the current state of the user when the evaluation process is performed.

6.4.1 Measuring the Attractiveness of Interaction

There are many alternatives to measure the attractiveness of a product and the quality of interaction. Section 2.2.2 described an study conducted by Wechsung and Naumann [172, 220] in which some different questionnaires with this purpose (i.e., AttrakDiff, SUS, SUMI and SASSI) were compared to each other. One conclusion we can extract from their work is that AttrakDiff was the only questionnaire —among the analyzed ones— providing a proper basis to implement a reliable and valid evaluation method for interaction experience in

multimodal contexts. This was confirmed by further works like [219] and [221]. This is the main reason why AttrakDiff was selected to be included in CARIM.

AttrakDiff [83] allows us to collect user ratings about interaction, or as said by its authors, to measure the attractiveness of interactive products. It is also suitable for evaluating user perceived quality in multimodal interfaces, as said above. This questionnaire is based on the Hassenzahl model of user experience, which poses that the attributes of a product can be divided in pragmatic and hedonic attributes. While pragmatic attributes refer to those aspects related to usefulness and usability of the system, the hedonic quality aspect addresses human needs for novelty or change and social status induced, e.g., by visual design, novel interaction techniques, etc. [82]

AttrakDiff uses 28 pairs of opposite adjectives to evaluate the following product dimensions:

- **Pragmatic Quality (PQ)**: describes the usability of the product.
- **Hedonic quality in Stimulation (HQ-S)**: describes how stimulating the product is.
- **Hedonic quality in Identity (HQ-I)**: describes to what extent the product allows the user to identify with it.
- **Attractiveness (ATT)**: provides a global value of the product quality perception.

Users (or potential users) use the AttrakDiff questionnaire to indicate their perception of the product quality once interaction is finished or during the session. However, providing ratings for 28 items, each time it is needed, might result in a bore and tiresome task. This is why CARIM incorporates a short ten-item version of the AttrakDiff questionnaire, already used and proven in, e.g., [84, 221].

Table 6.2 shows this brief version of the AttrakDiff questionnaire, which is composed of four pragmatic and four hedonic items, as well as two attractiveness attributes to rate the goodness ("bad–good") and beauty ("ugly–beautiful") of the product. Each of these items is rated using a 7-point Likert-type scale. The composite score ranges from 0 (minimum) to 60 (maximum quality perceived by the user).

6.4.2 Measuring Users Emotional State and Attitude toward Technology Use

QoE is individual to a given user, and largely depends on aspects like his/her personality and current state (Raake in [66]). Related work showed that there exist several factors

Pragmatic Quality (PQ)	Hedonic Quality (HQ)	Attractiveness (ATT)
impractical – practical	tacky – stylish	ugly – beautiful
unpredictable – predictable	cheap – premium	bad – good
confusing – clearly structured	dull – captivating	
complicated – simple	unimaginative – creative	

Table 6.2 Items included in the AttrakDiff mini version.

influencing quality perception. Factors like cognitive skills, mood, attitudes, and personality traits may influence ratings of the user.

As the readers can see in [10, 102, 221], users mood and attitude toward technology use are considered the two most influencing factors for perceived quality. On the one hand, good mood is considered to make recall of positive experiences. On the other hand, people holding better attitudes are more positive in the evaluation of aesthetics, pleasure and usability. Consequently a positive mood and positive attitude should result in better ratings. Therefore, to understand user perceived quality in deep, a measure of these two influencing factors should be incorporated into the proposed model.

In order to measure users mood we decided to use a faces scale. A faces scale is a method that uses a set of face pictures representing from an unhappy and frowning face to a smiling happy face. During testing the users are asked to pick the face that best represents their feelings. This method have been largely used in the literature to measure customer satisfaction, mood or even pain. Its effectiveness measuring users emotions has been showed in many works, e.g., [5, 78, 221, 226].

In this work it is proposed the use of an adapted version of the six faces scale described in [226]. First, since positive mood results in better ratings, then we decided to reverse the original scale. Now the scale starts from bad mood (the lowest rate) and ends with good mood (the highest rate). Furthermore, in order to simplify the process, only five faces were used. In this manner the *neutral* face represents the "center" of a well-balanced scale, along with two negative faces at left and two positive faces at right. This results in the scale depicted in Figure 6.2, which provides a simple mean for users to represent their current mood during tests. The score ranges from 0 (very sad) to 4 (very happy).

Moreover, in order to measure users attitude toward technology it is proposed an adapted version of the ATTIS (Attitude Toward Technology Integration Scale) scale described by Hassad in [81]. The author proposed a six-item scale to measure attitude of instructors

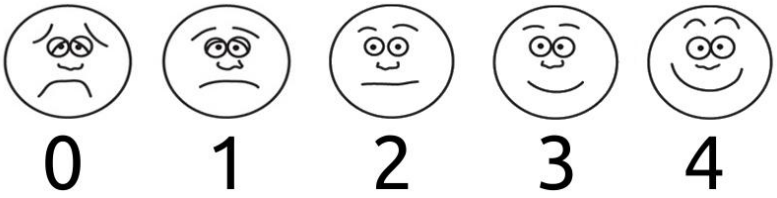


Figure 6.2 Resulting faces scale to measure users mood: (0) very sad, (1) sad, (2) normal, (3) happy and (4) very happy.

toward technology integration for teaching. However, in order to evaluate users attitude in a broader context, this scale was adapted as follows.

All items were rewritten using a more open description to get a more generic technology acceptance scale. Item 3 in the initial scale was removed because of its strong connection to the teaching context. Furthermore, now all the descriptions are "positive". This means that no reverse-coded items are used as in the original scale. As a result, higher values represent always more favorable levels of attitudinal predisposition toward technology use and integration.

This resulted in the scale described in Table 6.3. The new scale is composed of five items, each one representing a different facet of attitude toward technology: (1) perceived usefulness, (2) perceived pleasantness, (3) integration of technology, (4) self-efficacy and intentionality, and (5) perceived comfort. Each of these items is rated using a 5-point Likert-type scale. The composite score ranges from 0 (minimum) to 20 (maximum predisposition toward computer use and integration).

Statement	Rating
1. Using technology makes me more productive	◯
2. Using technology makes my daily life more pleasant	◯
3. Integrating technology into my daily life is not difficult for me	◯
4. I am not hesitant to use computers without assistance / help of an expert	◯
5. I am comfortable using computer applications for work and daily tasks	◯

Table 6.3 Items used to measure users attitudes toward technology. Rating: *Strongly Disagree*, *Disagree*, *Undecided*, *Agree* and *Strongly Agree*.

6.4.3 Arranging User Parameters into CARIM

As stated in Section 6.3 for context parameters, it is necessary to decide how and when user ratings should be collected before starting with the design of the model in Section 6.5. As mentioned above, by rating we mean an episodic assessment based on the given perception of the specific episode of application use [91]. Thus, user ratings should be collected when the expert considers that the "episode of application use" has concluded (e.g., at the end of a single test or after a whole experiment).

However, this decision may not be so trivial for users mood and attitude parameters. On the one hand, users mood might be measured only once during the experiment if it is merely necessary for analyzing mood influence on user ratings. However, user mood might be measured many times in case the expert wants to analyze its evolution during the experiment, as done by Korhonen et al. in [112].

On the other hand, it seems more than evident that users attitude toward technology has to be measured only once during an experiment. However, doing it before or after user interaction, might change the quality ratings provided by the users. As conclusion, when to collect users mood and attitude parameters directly depends on the aim of the analysis the expert is conducting.

6.5 CARIM Model Design

This section describes the proposed design for CARIM and its implementation. This section is structured into three blocks: (a) the main features of the base model, (b) the new proposed design and (c) some considerations about its implementation.

6.5.1 The Base Design: PALADIN

As said above, this work is based on PALADIN [137]. This multimodal interaction model was previously described in Chapter 5, and briefly summarized in this subsection.

PALADIN represents interaction data by turn, providing a dynamic description of multimodal interaction. It holds parameters related to the communication content, metacommunication, I/O information and modality description. Its design is centered around the concepts of *turn* and *dialog*, in which *"a dialog consists of a sequence of turns produced alternatively by each party"* (Möller, [167]) taking part in the interaction process.

The set of parameters in PALADIN is partially based on the work of Möller and Kühnel in [114, 168]. Some existing parameters were adapted and new ones defined in order to quantify multimodal interaction in a more abstract level than previous work. Moreover, new parameters were defined to describe the peculiarities of interaction in speech, visual-GUI and gesture modalities; e.g., number of user exploring actions, percentage of use of text and pointing devices. This finally resulted into a set of parameters arranged into seven categories:

1. *Dialog and Communication* (e.g., user turn duration)
2. *Modality* (e.g., modality used in a turn)
3. *Meta-communication* (e.g., speech recognition errors)
4. *Cooperativity* (e.g., appropriateness of system output)
5. *Task* (e.g., task success)
6. *Input* (e.g., number of words in a user utterance)
7. *Output* (e.g., number of elements shown at the screen)

PALADIN provides three main benefits to practitioners and researchers evaluating the usability of human-machine interfaces:

- Describing multimodal interaction using generic/abstract parameters. Different modalities are put at the same level, thus they are analyzed using the same metrics. More specific parameters are used to describe particular details of a specific modality.
- Having the same metrics to describe interaction in different contexts, which are structured within a common representation. It allows developers and designers to easily compare among different interaction records.
- A turn-based "step by step" description of multimodal interaction. It creates a relationship between collected data and time, providing evaluators with new opportunities for the dynamic analysis of interaction.

6.5.2 The New Proposed Design: CARIM

Before describing the design of the CARIM model, it is worth classifying the proposed model from a context point-of-view. For this purpose, it is used the classification proposed by Bolchini et al. in [29].

CARIM results in a model type *"C. Context as a matter of user activity"*. It is mainly focused on what the user is doing, thus the user is the subject of the model. The model design assumes a single user performing a main task by using a single mobile device. Not specially formal, the context definition describes in which conditions the user is doing the task. The context history is an important issue. Time and space are relevant because they provide information about the user current and past activity. Finally, automatic learning (when available) can be applied to guess user activity from sensor readings.

CARIM is aimed at providing a basis to evaluate QoE in mobile environments. As said above in Subsection 6.5.1, PALADIN quantifies only user-system interaction, which is not enough to assess QoE. Therefore, CARIM extends its design in two ways.

On the one hand, the parameters resulting from research in Section 6.3 were added to the initial model design to describe the interaction context in mobile scenarios.

The context is a dynamic and evolutionary entity in the model. As stated in Section 6.3 (and summarized in Table 6.1) only few of the context parameters do not change along an experiment with users (i.e., mainly *User* parameters and some ones related to *Device*). Therefore, these dynamic parameters have to be annotated by turn. Otherwise, static parameters have to be collected only once during an experiment. For this reason *User* parameters are separated from the dynamic context group to be annotated only once.

On the other hand, the parameters resulting from research in Section 6.4 were also incorporated into the model design in order to measure how attractive and user-friendly the product under test is. As stated above, CARIM considers the following subjective data of users: user ratings, user attitude toward technology and user mood (the latter are influencing factors of the former).

Attitude toward technology is considered as a non-variable attribute of users in a short term. Therefore, assuming that the attitude questionnaire will be answered only once during the whole experiment, this information need to be annotated only once.

Users *Mood* might be measured many times during an experiment to analyze its evolution during interaction. Capturing mood evolution was done in several of the works mentioned in Section 2.2.2, thus the model should be flexible enough to allow this.

We have a similar problem with *user ratings*. It was stated above that user ratings have to be collected when the expert considers that the episode of application use has concluded. As the idea of "episode of application use" may vary from one expert to another, the model design should be open to allow different interpretations of this concept.

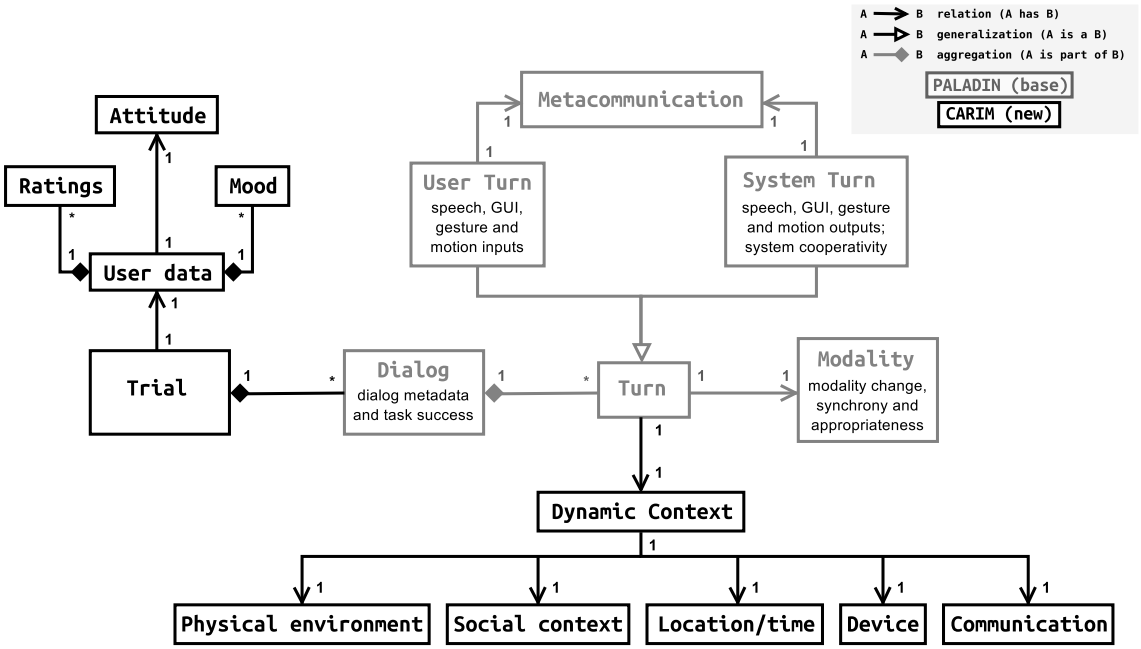


Figure 6.3 Design of the proposed model (CARIM).

Figure 6.3 depicts the proposed design for CARIM. As the reader can see, the design inherited from PALADIN is represented using a light grey color, while new parts of the CARIM model are represented using a black color. CARIM is structured around the concept of *Trial* instead of *Dialog*. Based on the definition of dialog provided by Möller in [167] (see above), a *Trial* can be defined as *"either one or a set of dialogs belonging to an experimental attempt, which is aimed at a specific experience measurement goal"*.

As said above, the subject of the model is the user, then a *Trial* is considered to be performed by a single person. As depicted in Figure 6.3, static context data related to the *User* is annotated at trial level. This means that it is collected only once for the set of dialogs composing the trial. Next to user data the reader can find *Attitude*, that is annotated only once as well.

User ratings and *Mood* are a different case. As these values can be annotated either one or several times depending on the experimental study, the model design provides an aggregation of them at trial level. This decision provides more flexibility, but lacks a chronological link between these data and a *Dialog*. This design issue is further discussed in Section 6.8.

Context information is collected at turn level, i.e., it is annotated stepwise in the model

during user interaction. This supports the concept of changing and evolving context proposed by Jaroucheh et al. in [96], as well as the need to consider the context information available in the different domains the user visits. The reader can see in Figure 6.3 that all the context parameters described in Section 6.3, except those related to the user, are annotated by turn. As a result, the model provides a link between time, the user interaction description, and other context information like the user location, the social company, the device CPU load in that moment, etc.

6.6 CARIM Model Implementation

For the implementation we transferred the data model of CARIM into a design in the Eclipse Modeling Framework [203] (EMF), which provides model transformation and automatic code generation functionality. The widely used EMF allows the definition of comprehensible, flexible, and extensible models, as well as the syntactically validation of concrete model instances. Tools like EMF help to make the modeling process more effective, and provide indispensable functionality to validate and extend the model [103]. The design of the model will be used to automatically generate the Java source code, which shall be integrated into the applications in which CARIM model instances have to be created.

In order to collect interaction and context parameters to fill the model instances we propose using the AHE11 [135]. This interaction extraction tool is based on the original Android HCI Extractor [134]. It is open-source and is aimed at collecting different interaction and context data automatically, as far as the device capabilities allow it.

The AHE11 can be easily used to create CARIM model instances at runtime in Android platforms. Such instances are valid to represent many different interaction scenarios, from the usage of an application during a couple of minutes to the usage of a device during hours. The AHE11 currently runs in Android devices with API 11 (Honeycomb 3.0.x) or higher. Android was chosen because of its open character and because it can run in many different mobile and non-mobile platforms, e.g., smartphones, tablets, netbooks, smart-tv, etc.

To implement questionnaires we also developed the Generic Questionnaire Library (GQL8) for Android. This open-source tool (available at [135]) automatically generates and displays questionnaires from a very simple JSON [51] specification. GQL8 was used to automatically collect user ratings at the end of each test trial. I was used also to collect some context parameters that can not be collected automatically (e.g., social company) as

well as to collect users mood and attitude values. The GQL8 currently runs in Android devices with API 8 (Froyo 2.2.x) or higher.

Data collected with AHE11 and GQL8 can be directly put into a new model instance. However, it might be complicated to accomplish this task manually. In order to ease the integration of CARIM into research and production systems, a helping framework called Instantiation Framework (IF) was developed. This element uses the information provided by the data sources to automatically create and fill the CARIM instances (see Figure 6.4).

The IF is aimed at managing the entire life-cycle of such instances. It provides an API to be notified with interaction data (e.g. the user touches the screen, new ratings are available). These data are then used to create and update the current CARIM instance. The IF provides access to such instance during interaction for runtime analysis, or once the interaction is finished for off-line analysis. The open-source implementation of this framework is available at [135].

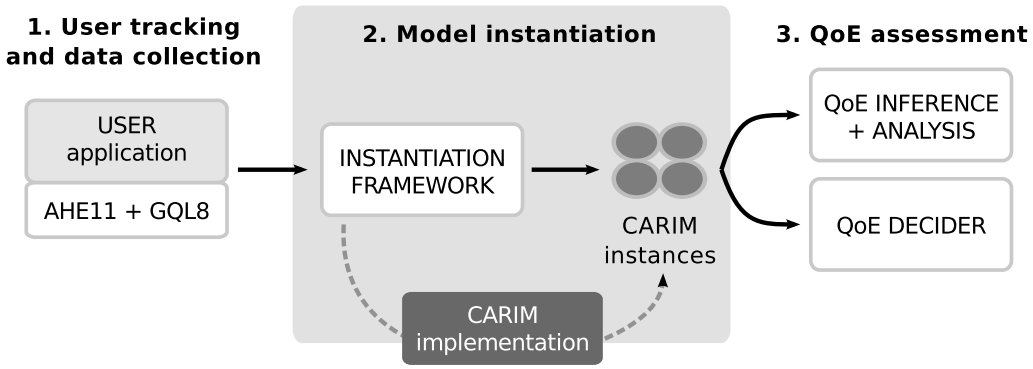


Figure 6.4 Typical execution scenario for creating CARIM instances.

Figure 6.4 depicts a typical instantiation scenario for CARIM, similar to the described later in Section 6.7. This scenario is divided into three stages:

1. **Data collection**. Necessary data about interaction, context and the user are collected. These data may be captured from real application usage in real-time (e.g., by using the AHE11 and GQL8), from interaction logs or anyhow artificially produced (e.g., user simulation).
2. **Creation of model instances**. The Instantiation Framework is notified with the interaction data, from which it creates live CARIM instances.

3. **Inference and analysis of QoE**. The data stored within CARIM instances is used to determine QoE of user. The instances can be checked at runtime to make live decisions for application adaptation.

6.7 Experiment

This section describes an experiment in which users interacted with two different interaction designs for a game named *UMU Lander*. CARIM, along with the tools described above in Section 6.6, were used to implement the evaluation process. The purpose of this experiment is not to present a formal users study, but to give an illustration of results obtained with the CARIM model, as well as to show its validity to calculate, analyze and compare QoE.

This experiment compares two interaction designs for the same application. The comparison is based on the QoE value achieved by the users with each version. The experiment is also aimed at investigating if some context factors can explain differences in users interaction and in their QoE. More concretely we studied the influence of social context and users mobility on user behavior and quality perception.

For this purpose, we selected three hypotheses that attempt to be validated using the data extracted during the experiment. To select them, we first encouraged our research group members to suggest scenarios in which the context factors may influence or change the way the users interact with their mobile devices. Then, all of the members voted for those three scenarios they considered it exists a higher influence of context in interaction. As a result of this analysis we posed the following three hypotheses:

> (H1) Users play longer in least time-restrictive environments (e.g., at home) than in most restrictive ones (e.g., at work).
> (H2) Users provide input with more arousal (i.e., using shorter and more frequent interactions) when there are people around them.
> (H3) The QoE level for users is affected by the social context and their mobility situation while they are playing the game.

6.7.1 Participants and Material

30 Spanish- and English-speaking individuals (16 male, 14 female) between the age of 18 and 58 (m=29.9, sd=8.96) took part in the study. The two versions of the game were

available in Google Play for installation. The participants were recruited using an open call for participation sent to our colleagues and contacts through email, Twitter and Linked-in. Other participants were directly encouraged by us to take part in the experiment using our own mobile devices.

The tested application was UMU Lander, a game based in the original Lunar Lander example provided by the Android SDK[1], which was extended to work with touch and motion modalities. The aim of this game is landing a lunar vehicle on the Moon surface. UMU Lander is presented in two versions. Version #1 (from now *lander_acc*) uses the accelerometer to change the lander heading, while speed is increased by touching the screen. In version #2 (from now *lander_butt*) both steering and acceleration are controlled by using three buttons at the bottom of the screen. Therefore, *touch* (GUI) and *motion* are used as input modalities, while the output is always given via *GUI* modality.

6.7.2 Procedure

The users were encouraged to play the two versions of the lander game. For each version, the experiment was composed of three stages:

1. The user is briefly asked for some information about the surrounding context. These questions are used to get those data that can not be automatically extracted using the AHE11 during interaction (e.g., social company, mobility level).
2. Once answered, the user interacts with the application. There is no restrictions in this stage, so the user is free to play the game as many times as he/she wants.
3. Finally, the user is asked about his/her current mood. Then the attitude and AttrakDiff Mini questionnaires are displayed to be answered.

The UMU Lander was augmented with the AHE11 to extract interaction and context data and with the GQL8 for the questionnaires. As the reader can see, the whole experiment is encapsulated within the application. All data necessary for the experiment are collected interactively, either through the use of GQL8 questionnaires or by using the AHE11 to automatically extract interaction. The resulting files (i.e., the CARIM instances) were sent to a server located in our dependencies in the University of Murcia (Spain). The anonymity and privacy of the participants was guaranteed throughout the whole experiment.

[1] http://developer.android.com/sdk

6.7.3 Results

The experiment resulted in a set of CARIM model instances. These instances describe user-system interaction, its surrounding context and user-related data (i.e., some demographic information, attitude toward technology, and mood and perceived quality after the experiment). These instances were used later to determine the QoE of each participant and implement its analysis under different context situations.

6.7.3.1 Comparing the Two Interaction Designs for UMU Lander

The first result comes from comparing the QoE of users for the two proposed interaction designs. The QoE value was calculated for each CARIM instance by adding up the values of the ten AttrakDiff-Mini questions previously asked to the participants (see Table 6.2). Then, the resulting value was converted to a final value between 0 and 1. As a result, each quality dimension measured by the questionnaire is represented in the final value of QoE according to the number of attributes of each dimension (i.e., PQ=40%, HQ=40% and ATT=20%). Then, the average values of QoE for each interaction design were calculated and included in Table 6.4. As we supposed before conducting the experiment, results showed that users have a better experience using the accelerometer in *lander_acc* instead using the buttons. However, the difference with *lander_butt* was pretty small.

Version	QoE mean	QoE+mood+attitude mean
lander_acc	(m=.469, sd=.206, N=30)	(m=.440, sd=.191, N=30)
lander_butt	(m=.428, sd=.239, N=30)	(m=.386, sd=.213, N=30)

Table 6.4 QoE mean of users for the two proposed interaction designs.

Then, the final QoE values for each user were adjusted by using the *attitude toward technology use* and *mood* values. First of all, the correlation of these influencing factors with the calculated QoE was analyzed to determine the relation between these values. We used Pearson correlation coefficient, as it is suitable for scale variables whose values represent ordered categories with meaningful metrics, thus distance comparisons between the values are appropriate [34].

We obtained (Pearson's r=.28, N=60, p=.02) for *attitude toward technology use* and (r=.49, N=60, p=.00) for *mood*. Then, based on these correlations, the QoE value was adjusted

by using a constant factor *f* (for this case we used f=-.3) to limit the significance of the influencing factors. We used a negative factor because the adjustment should be inverse, i.e., for a higher mood or attitude level the final QoE value should be reduced, and vice versa.

As the reader can see in Table 6.4 the adjusted QoE values are smaller than original ones (very often mood and attitude values were positive). The difference between the QoE of users with the two different versions is slightly increased in favor of *lander_acc*. Then, as showed by the original and the adjusted QoE values, it can be concluded that users prefer the interaction design used in *lander_acc*.

6.7.3.2 Validating the User Behavior Hypotheses

Once compared the two interaction designs for the UMU Lander game, we analyzed the data in the CARIM instances to attempt to validate the hypotheses posed above.

For (H1), in a first step, we provided numeric values to the *social arena* variable arranged in increasing order of how limited in terms of time they are. It resulted in *0-domestic*, *1-leisure*, *2-educational* and *3-work*. Then, we used the measures assessing *social arena* and correlated them with the *total time* spent by the user during the game. As the reader can see in Table 6.5, these two parameters showed a positive Pearson's correlation. Contrary to our expectations, users tended to spend more time in more time-restrictive environments.

For (H2) we also assigned numeric values to the *social company* variable arranged in increasing order of number of people around the user. It resulted in *0-alone*, *1-with a person* and *2-with a group*. The measures assessing *social company* were correlated with the *number of input elements* (i.e., motion and touch events) provided by the user during a game. In line with our assumptions, we observed a positive correlation showing that the higher is the number of people around the user in the interaction context, the higher is the arousal with which the user interacts with the application (see Table 6.5).

Finally, for (H3) we again provided numeric values to arrange the values of *mobility level* in increasing order of activity. It resulted in *0-sitting*, *1-standing*, *2-walking*, *3-sporting* and *4-driving*. Then we tried to find any correlation between the calculated *QoE* and *adjusted-QoE* values with the measures assessing the *mobility level* of the user. We also tried to find correlations between the two social context parameters used for H1 and H2 and the calculated QoE values. The results (see Table 6.6) showed that only the *mobility level* of the user presents a significant correlation with the quality of experience perceived by the

	Total time	Game time (average)	Input elements per game
Social company	(r=.23, p=.15)	(**r=.36**, p=.02)	(**r=.33**, p=.01)
Social arena	(**r=.43**, p=.01)	(**r=.31**, p=.05)	(r=.23, p=.07)
Mobility level	(r=-.05, p=.72)	(r=-.02, p=.93)	(r=-.04, p=.82)

Table 6.5 Correlation (Person's *r*) between social and mobility context parameters and interaction parameters, *N = 60, p-*$_{2tailed}$.

user. Since all the users who made the experiment were sitting or standing, in general they tended to have better experiences doing the second way.

	QoE	QoE + mood + attitude
Social company	(r=.11, p=.38)	(r=.10, p=.42)
Social arena	(r=-.04, p=.81)	(r=-.08, p=.62)
Mobility level	(**r=.46**, p=.02)	(**r=.41**, p=.03)

Table 6.6 Correlation (Person's *r*) between social and mobility context parameters and determined QoE values, *N = 60, p-*$_{2tailed}$.

6.8 Discussion

This chapter describes a runtime model intended for serving as a basis for the implementation of QoE inference, analysis and comparison processes. The main contribution of this work to the UX research field can be summarized as follows:

1. The analysis of related literature to select a valid set of context and quality parameters to analyze QoE in mobile applications.
2. The design of a model aimed at uniformly representing user-system interaction within real mobile contexts. For this purpose, it arranges parameters related to the interaction, the surrounding context and the perceived quality of users.
3. An open-source model implementation ready to be used with real applications and users in their natural daily environments, as posed in [216].

6.8.1 Modeling Mobile Interaction and QoE

Three research questions regarding the limitations found in current state-of-the-art were posed in Section 6.1. These questions —that have been answered throughout this chapter— as well as other open questions commented above are further discussed in the following.

To answer question **Q1**, a runtime model to analyze multimodal user-system interaction (i.e., PALADIN) was selected as the basis for this work. PALADIN describes interaction in a stepwise manner, thus it fitted perfectly the dynamic nature of the changing context of mobile scenarios. Therefore, the base model enabled the incorporation of surrounding context parameters by turn, which allows to provide a step-by-step description of the interaction context in the new proposed design. PALADIN provides a static-data section as well. This section was perfect for allocating user perceived quality parameters, as well as other parameters describing several influencing factors of interaction quality.

Instances of the proposed model (i.e., CARIM) describe therefore mobile interaction by using three dimensions: the interaction between the user and the system, its context and the user ratings. As a result, as posed in research question **Q2**, QoE determined for a specific interaction instance can be analyzed and interpreted using these three dimensions. Data are thoroughly arranged and connected to each other to help applications and experts to understand why such a certain level of QoE was inferred.

On the one hand, we have context data directly linked to interaction data. It allows experts to analyze the effect of context changes in user interaction (e.g., how user behavior changes when a new person appears in the social context). Henricksen et al. posed in [86] two main limitations related to context data.

First of all they talked about the gap between collected data and the level of information that is useful to applications. This problem is solved at runtime within the AHE11, where collected data are immediately processed to fit CARIM constraints. They also argued that there are many cases in which context information may be incorrect, inconsistent or incomplete. The proposed model does not provide any constraint between context attributes and other related data to avoid meaningless combinations. This problem is intended to be solved as future work.

On the other hand, a relationship between interaction and context data and time is created as well thanks to the runtime nature of the model. Related work includes many references to the link between data and time. For example, Forlizzi and Battarbee described

that user experiences are a constant stream of interaction sequences that change user behavior and emotions [69]. Jaroucheh in [96] and Ickin et al. in [91] argued that context information history has to be considered when modeling the situations, as the current state cannot be understood in isolation from the previous states. CARIM emphasizes the importance of time by describing the whole "course" of the dialog, thus enabling the analysis of interaction and context evolution. The mobile interaction process can be assessed either in real-time (i.e., by taking the collected data history until a specific time) or at the end of the process (i.e., by taking the whole dialog information).

Finally, the CARIM design provides enough flexibility for storing user data (i.e., ratings and influencing factors) and link them to interaction data. For example, a single user-data record can be stored related to the whole trial or, conversely, a different record can be stored for each dialog composing the experiment. More than one ratings and mood records can be stored for the same trial, even for the same dialog, if it is the goal of the study to analyze their evolution. However, providing this flexibility in the design causes a lack of a "chronological link" between user data and time. It is not clear in the model design when the user data was collected. It is also unclear how many times these data was collected. Therefore, this relationship between user data and time should be clearly showed in the experiment briefing.

Another limitation is related to the extraction of subjective and cognitive data. It is not straightforward to automatically extract information related to the cognitive domains of the user. Thus, these data have to be asked to the user [88]. Methods like the aforementioned ESM [48] or SAM [117] help experts to ask users and discover things like their mood, or whether they liked the location of the keys on the smartphone screen. However, for this work an automatic method was created to interactively ask subjective and cognitive data to the user during the experiment (see Section 6.7). This avoided the manual implementation of questionnaires and interviews with users. As a consequence, the presence of experts was not needed during the execution of the experiment.

Research question **Q3** concerns the lack in user experience research of a common vocabulary for describing and comparing user experiences. This problem was previously posed by Korhonen et al. in [112]. CARIM try to solve this by structuring data into a common representation that can be used to analyze interaction and QoE in systems of a very different nature. The same metrics and format are used to quantify the interaction between the user and the system regardless of the device and application under testing, the

current interaction context, and even the modalities used to provide input and output data. As a result, different interaction records can be analyzed and compared to each other. This provides experts with unified criteria to describe multimodal interaction and its context in different execution scenarios.

6.8.2 CARIM Implementation and Experimental Validation

Section 6.6 described the implementation work resulting from this research. Unlike many other approaches, we provide a real open-source implementation of CARIM ready to be incorporated into current Java-based developments. This implementation can be easily extended with new parameters in the same manner PALADIN was extended to become the model proposed in this chapter. Furthermore, the open implementation of an interaction instrumentation tool and a library to implement automatic questionnaires are also available and ready to be used in Android devices.

The implementation of CARIM was tested in a real experimental setup described in Section 6.7. The experiment was used as proof of concept for implementing the assessment processes for which CARIM was initially designed. Instances of the CARIM model were used to faithfully represent the multimodal interaction process between different users and two designs of the same application.

Subjective and cognitive data allocated in the model instances allowed us to determine the QoE of the users performing the experiment. The calculated QoE values were used to compare the two different interaction designs for the same application. Moreover, the thorough arrangement of data in the model allowed us to easily find correlations between these QoE values, the users behavior and some context features. As a result, it was showed that even a small set of users can be useful to draw important UX conclusions by using CARIM instances.

No special sensors were needed during the experiment. All parameters in CARIM were automatically extracted, either by using the AHE11 instrumentation tool or through the use of interactive questionnaires with GQL8. The users were not distracted by the use of unusual devices that they do not normally carry with them. As Thurnher et al. argued in [210], additional ports and interfaces as well as the intrusiveness of sensors might impair the usability study by disturbing and distracting the user during the test. Therefore, the

Feature	CARIM	PALADIN	CUE Model	Schulze and Krömker
1. Quantifies user-system interaction	■	■	⊞	□
2. Follows a runtime approach	■	■	□	□
3. Provides data correction mechanism	□	□	□	□
4. Is interaction-context-aware	■	□	□	□
5. Considers context evolution	■	□	□	□
6. Considers product quality dimensions	■	□	■	■
7. Considers users state and attitudes	⊞	□	■	■
8. Considers users state evolution	⊞	□	□	□
9. Measures user experiences	■	□	■	■
10. Allows user experiences comparison	■	□	⊞	■
11. Provides ready-to-use method	■	■	?	?

■ *yes* ⊞ *partially* □ *no* ? *unknown*

Table 6.7 Comparison of different approaches evaluating user-system interaction.

proposed method enhances QoE testing by avoiding additional distraction of subjects caused by observers or additional devices.

6.8.3 CARIM Compared with Other Representative Approaches

The features of CARIM were compared to other existing approaches aimed at evaluating user-system interaction. The aim of this comparison is to highlight the main innovations introduced by the proposed design, as well as its main deficiencies. The following three approaches were selected for the comparison for the reasons described next. PALADIN [137], because it proposes an interaction representation model, and because it is the model in which CARIM is based on. CUE-model [127], because it proposes a model of user experience concerning the relation between interaction characteristics, emotional reactions and the perceived usability and system aesthetics quality. And the framework presented by Schulze and Krömker [199], because as CARIM, it combines qualitative and quantitative data extracted from the users to analyze their experiences.

The reader can see the results of this comparison summarized in Table 6.7. First, as discussed above, CARIM enhances the PALADIN model by supporting the measurement

of mobile user experiences. For this purpose, the perspectives of the interaction context, the quality perception, and the mood and attitudes of the users are incorporated into the interaction analysis process.

CUE and the framework presented by Schulze and Krömker (from now S&K) do not quantify user-system interaction beyond some task completion measures included in CUE. Actually, this feature might be considered out of the scope of these methods. Moreover, it is surprising that such a fairly recent user-experience analysis methods do not take into account the influence of the context surrounding the user, specially because of the increasing importance of mobile interaction in the last few years.

It could be said that the main limitation of CARIM with respect to CUE and S&K is the lack of a deeper analysis of the cognitive state of the users. CUE studies the subjective feelings and emotional reactions of the users in detail. S&K thoroughly analyzes the users competencies and motivations, as well as their emotions. Meanwhile, CARIM only considers the current mood of users and their attitude toward technology use.

Finally, it is worth saying that none of the methods compared above (inc. CARIM) provides a data correction mechanism. This is an interesting feature that should be implemented in order to avoid inconsistencies in the description of interaction, which may lead to erroneous interpretation of user experiences.

6.9 Conclusions

This chapter presents CARIM, a runtime model aimed at describing the interaction between the user and the system, its context, and the perceived quality of users. The model includes a set of parameters to quantify multimodal mobile interaction and its surrounding context, as well as subjective data collected from the user. CARIM arranges all these data into a common and dynamic structure. Instances of this model keep the same format regardless of the system, the context, the users, and the modalities under test.

As a result, CARIM provides unified criteria for the inference and analysis of QoE in mobile scenarios. The model instances can be used to determine QoE of users. Different instances can also be easily compared to each other. Moreover, their runtime nature allows applications to make context-based and QoE-based decisions in real-time to adapt themselves, and thus provide a better experience to users.

An extensible implementation of the CARIM model and its instantiation framework has been provided as a contribution to the open-source community. This implementation is ready to be used with real application users in their natural daily environments, as suggested in [216]. This implementation was incorporated into a study with users and two different mobile applications as proof of concept. CARIM was successfully used to assess user-system interaction and to measure and compare the QoE of users in mobile environments.

7

Conclusions and Further Work

This chapter draws some conclusions about the research activities conducted in this PhD thesis, and describes future lines of research work.

Section 7.1 includes some conclusions about how the main problems found in current approaches of Software Testing and Human-Computer Interaction Analysis have been addressed in this PhD thesis. It summarizes also the four main research goals outlined in this thesis as well as the solutions designed in an attempt to achieve them.

Section 7.2 describes new lines of work that arise from our research in this PhD thesis, and that could be addressed in a near future.

7.1 Conclusions of this PhD Thesis

This section starts with an overview of the main concepts behind this PhD thesis. Then, some conclusions about the problems tackled in this research work are drawn. Finally, the four main goals outlined in this thesis are summarized, as well as the solutions proposed of achieving them.

7.1.1 Driving Forces of this PhD Thesis

Software Quality is a very **broad concept**. It encompasses many disciplines, from those assessing technical features of the software, to those analyzing psychological aspects of the human beings. As commented at the beginning of this PhD thesis, the quality of a software can be achieved in many different ways (e.g., checking the functional requirements, validating the software output, evaluating users perceived quality, etc.) This work tries to narrow this problem by focusing on the assessment of user interfaces through the analysis of the interaction between the user and the system.

User-system **interaction has changed dramatically** in the last few years. Not so many years ago, people's interaction with computers was restricted to desktop systems, mainly for leisure or to accomplish tasks in domestic, work, or academic environments. Systems in which a keyboard and a mouse enabled us to communicate with the software, and that showed us results through a conventional screen and speakers.

Nowadays, we continuously interact with the so-called "smart" devices. These devices provide full access to information, full connectivity to other people, and powerful processing capabilities. They are in our pockets, our hands, or even put on our heads in the form of wearable glasses. These devices usually offer us different ways of interaction by means of different sensory modalities, and allow us to communicate to them, e.g., touching the screen with our fingers, using our voice, or simply shaking or flipping the device.

In this changing and challenging environment, this PhD thesis addresses **two different approaches to achieve quality of a software** through the analysis of user-system interaction. On the one hand, the quality of the different components of interaction (i.e., input, output, and context) can be achieved separately. The main advantage of this approach is that tools and methods can focus their efforts on achieving the quality of a single component of interaction, and, as a consequence, the quality of the entire software is enhanced.

On the other hand, interaction can be assessed as a whole, as a single flow of actions from the user to the system and vice versa. In this approach the main advantage is that methods try to achieve the quality of the "whole thing" and not of the parts compounding it. This approach keeps the totality of the interaction process and enables the analysis of those cause-effect relationships between the actions of the user and the system.

7.1.2 Work and Research in User-System Interaction Assessment

Current approaches in Software Development and Testing, and HCI Analysis and Assessment have been analyzed during this PhD thesis to identify the main **problems when achieving software quality**. This analysis was specially focused on these approaches analyzing the actions of the user and the system, as well as the interaction process between these two parties. The identified problems, as well as the main contributions of this research work attempting to solve them, are summarized in this section.

Achieving the quality of a system or a software is still a **cumbersome and expensive process** in terms of time and human resources. Furthermore, developing testing tools and integrating them into real world, complex scenarios may result in a troublesome task. This causes testing to be hardcoded or implemented manually in many cases. Automation has been used by many of the analyzed approaches to reduce the effort and costs of achieving quality of an application. However, automating the actions of experts (specially those depending on subjective judgments) is sometimes not straightforward. This PhD thesis provides different frameworks to support the automation and encapsulation of testing and analysis processes, as well as to ease their integration into developments.

Nevertheless, **analyzing and testing user interfaces** could be even more complex than testing other components of a software. User interfaces have very **distinctive features** that hinder these processes, and that often prevent the tools used to test the rest of the application to be used to test the interface. These pitfalls and how they were tackled in this PhD thesis are discussed in the following.

When testing user interfaces the **execution environment becomes more complex**. Additional elements like the GUI platform, the voice recognition system, and other mechanisms used to support interaction by using different modalities have to be considered. This causes that tools and methods designed to be used into a specific testing environment can not be reused in others. Therefore, tool adaptation is an aspect that should be taken into account

when creating such methods. This PhD thesis has focused on providing open designs, which allow the proposed frameworks to be easily extended or adapted to be integrated into testing scenarios of a different nature.

There exists also **low standardization** of the methodologies used to describe users and system actions, specially in those approaches focused on HCI assessment. Some of the analyzed research works agree on this. Many of current approaches use their own representation of interaction intended for solving a specific problem (e.g., to analyze interaction with a particular device). However, these specific representations are often hard to reuse when facing problems of a different type (e.g., if we are analyzing interaction with a different device). Designing more generic and extensible solutions might help experts to apply these techniques into scenarios of a different nature (e.g., using the same method to analyze and compare an application running on different platforms or by using different modalities). This PhD thesis is specially concerned with providing generic designs and representations in order to widen the range of scenarios in which these can be applied.

Moreover, deciding which parts of a user interface will be evaluated, as well as what data will be included in our analysis, is not straightforward. Many authors call this stage **coverage criteria**, and it represents a main concern for developers and testers if they want to provide efficient testing or analysis processes. Using methods and tools to assist testers in this decision is helpful to simplify and narrow these processes, and thus ensure their effectiveness. Hence, the solutions proposed in this thesis try to lead experts towards those values, from all the available data, essential to solve the problem. These solutions have been designed to be extended when needed.

Another problem when assessing user interfaces is that it often involves the **analysis of a dynamic process** in which the users and the system are exchanging information within a changing context. Time represents an essential element of the interaction process. Therefore, it has to be properly "captured" by testing methods to enable the dynamic analysis of interaction. This research work is aware of the importance of time, thus all the proposed solutions are based on the dynamic and stepwise analysis of the actions of the users and the system.

User interfaces are often tested with real people. This means that testing and analysis processes run while potential users are using the software, which represents a critical feature. The implementation of efficient **real-time testing processes** is essential in order not to affect the users interaction and the system performance during this stage, and thus

guaranteeing the validity of the results and conclusions extracted from the experiments. The efficiency of such processes has been another important concern in this PhD thesis. The provided tools are designed either to run in parallel with the tested application exploiting the system architecture, or using the idle time between different actions of the user.

Interaction with a user interface may also **involve different sensory modalities** to allow disambiguation of input and to provide complex communications with a higher bandwidth. In the literature the reader can find many approaches describing multimodal interaction by separating the analysis of the different modalities. However, multimodal communication is a continuous process in reality. It can be seen as a single stream of actions and information exchanged between two or more parties, regardless of the type or number of modalities used during the process. Part of this PhD thesis focused on providing a uniform, generic, and dynamic description of multimodal interaction. To do that we have identified the existing equivalences between different modalities and harmonizing different types of data to define multimodal interaction as a seamless communication process. Putting different modalities at the same level of abstraction enables experts to analyze multimodal interaction as a whole, and not as an aggregation of inputs and outputs of a different nature.

Finally, The design of user interfaces is **prone to be changed** more frequently than the design of other components of a software. This is mainly because interfaces are being continuously adapted to changing users needs during their development, as well as due to the implementation of new functionalities that need to be supported. Testing processes are really sensitive to changes, specially if the tests or experiments conducted before these changes are reused in future testing or analysis processes, e.g., when implementing regression testing. This requires a high robustness of testing methods, as well as a high tolerance to changes in the design of the user interfaces. This PhD thesis was specially concerned with this problem. The provided tools that work directly on the user interfaces are often based on their internal functionality (e.g., data events or internal organization of interface components) instead of using values corresponding to their external appearance or behavior (e.g., the arrangement of elements in a GUI or the specific content of a system utterance).

7.1.3 Goals Achieved in this PhD Thesis

The **four main research goals** of this PhD thesis were posed in Section 1.3 and structured into two blocks depending on whether the interaction components are analyzed separately (Block 1) or the interaction process is assessed as a whole (Block 2). According to these goals, the backbone of this research work has been structured into four main chapters, i.e., Chapters 3, 4, 5, and 6. Each of these chapters describes the research activities aimed at achieving one of these goals, including also the design and implementation of the resulting methods and tools.

$\mathbb{G}$**1.1** suggested to find a framework to support the development of testing tools aimed at validating the software response; to support the automation of GUI testing processes that are often performed manually; as well as to allow the simulation of a human tester to implement testing in a real, reliable, and robust scenario. Ease the integration of these tools into applications of a different nature was a goal as well. This PhD thesis proposed the OHT framework to achieve this goal.

The OHT (**Open HMI Tester**) framework provides an open and adaptable architecture for the development of GUI testing tools. It uses a capture/replay approach that can support the automation of different testing processes based on GUI introspection, the capture of user interaction, and/or the execution of actions into the tested application. Moreover, some of the modules in OHT can be easily adapted to work in different testing environments (e.g., to test applications based on a different GUI platform).

After designing a solution to validate software output, $\mathbb{G}$**1.2** aimed at finding a lightweight and easy-to-integrate solution for implementing input data verification processes into GUI developments. The user input must be valid and conform to the data requirements, which should be written using a verification language chosen by the developers. This solution should be interactive to ease the work of developers, testers, and users during the whole life-cycle of a software. According to these requirements, this PhD thesis proposed S-DAVER.

S-DAVER (Script-based DAta VERification) is a lightweight verification framework that validates input data while the user is interacting with the software. It provides an interactive process in which developers can change the rules at runtime while testing the application, and users are dynamically notified about data errors while using it. All the verification processes are encapsulated in a separate layer that, once integrated into an application,

establishes a trust relationship between the GUI and the business logic. S-DAVER makes data verification an integral part of the development, testing, and execution processes.

Aiming at analyzing interaction as a whole, $\mathbb{G}$**2.1** suggested to find a generic description of multimodal interaction to support its instrumentation and assessment. This description had to capture the dynamic nature of the interaction between the user and the system. Furthermore, the comparison between different interaction records should be allowed, regardless of the execution context from which they were previously recorded. This PhD tried to fulfill these requirements with PALADIN.

PALADIN (Practice-oriented Analysis and Description of Multimodal Interaction) is a runtime model arranging a set of parameters which are used to quantify the interaction between the user and the system in multimodal systems. These parameters are annotated in each interaction step in order to preserve the dynamic nature of the dialog process. As a result, the model instances of PALADIN are used as an unified criteria to analyze, evaluate, and compare interaction extracted from different unimodal and multimodal systems. These instances are used to evaluate usability of such systems.

Based on the idea behind PALADIN, $\mathbb{G}$**2.2** aimed at providing a framework to support the assessment of user experiences in mobile scenarios. Such a framework had to include a generic and dynamic description of the surrounding context of interaction, as well as a set of metrics to capture users impressions about interaction. As a result, this framework should provide unified criteria for the assessment of systems usability and QoE in different mobile and non-mobile scenarios. With this purpose, this PhD thesis presented CARIM.

CARIM (Context-Aware and Ratings Interaction Model) is a runtime model describing the interaction between the user and the system, its context, and the perceived quality of users. It arranges a set of parameters into a common structure in order to provide a uniform method to describe, assess, and compare interaction experiences regardless of the system, the context, the users, and the modalities under test. Moreover, the runtime nature of CARIM allows applications to make context-based and QoE-based decisions in real-time to adapt themselves and thus provide a better experience to users.

As a result of this PhD thesis, and as proof of concept for our research work, four main software elements have been provided as a contribution to the open-source community. To show the **validity** of the proposed methods, these software have been either integrated into the internal processes of a software development company or tested with users in experiments conducted both in laboratory and real environments (see Section 1.5 for further

information). Furthermore, popular means of distribution like Sourceforge, Google Code, and GitHub were used to show our results and to allow the **open-source community** members use them and provide us with valuable feedback to enhance our work.

In order to meet the goals proposed in this PhD thesis, the work presented in this dissertation encompasses different ways of achieving software quality, focusing mainly on the analysis of users-system interaction. This work represents also a further step toward the definition of more generic, open, and adaptable methods for the analysis and assessment of human-computer interaction, its elements, and its surrounding context. Those methods presented as result of our research are ready to be used with real world applications, adapted to new execution environments, or extended in further research activities to be improved, as stated in the next section.

7.2 Future Lines of Work

This section describes new lines of work arising during the development of this PhD thesis. These open lines should be addressed in future research on Software Testing and Human-Computer Interaction.

One concern would be finding new alternatives to **automate the visual validation process** performed by testers when checking the validity of software output. This process is still wearisome, and increases the dependency on experts during software testing stages. A solution could be providing a way to visually specify what are the expected results, and then use this specification to automatically compare the results obtained during testing to the expected ones (e.g., by comparing the graphical output of the system, like in [40]). Such a method should be seamlessly integrated into current testing processes.

New methods to **ensure the correctness and consistency of specifications** should be also explored. This feature can be applied to two particular problems tackled in this thesis. On the one hand, ensuring the correctness and consistency of the rules used in S-DAVER. The rules are written using scripting languages. These languages provide high flexibility when defining and maintaining the rules, but often lack a method to check their internal consistency. On the other hand, check the correctness, consistency, and completeness of the models of interaction and its context (i.e., PALADIN and CARIM). Constraints between interaction and context data should be provided to avoid meaningless combinations.

Migrating the solutions proposed in Block 1 **from an only-GUI to a multimodal interaction context** would be another idea to be developed. Multimodal systems also need for support in validating their inputs and outputs, which have the peculiarity of being provided using different sensory modalities. The work and the knowledge acquired in Block 2 could be used to treat different inputs and outputs in a uniform way for its automatic validation, and thus achieve quality in such systems.

To **enhance the design of the proposed interaction models** is also among our goals. New parameters may be added to fix potential errors as well as to meet new emerging needs, specially those related to the **instrumentation of new sensory modalities**. A particular case is the analysis of gesture input and output in depth. In recent years, gestures are becoming even more popular as input modality, specially thanks to the advances in TV and video games fields (e.g., LG SmartTv Motion Recognition System, Nintendo Wii, Microsoft Xbox Kinect). Peculiarities of this modality should be integrated into the methods proposed in this work.

The models could also be enhanced with the incorporation of **additional aspects or dimensions of the interaction process**. One example is the description of the task the user is engaged in. This might help experts to identify interaction problems and link them to particular stages of the experiment being conducted. Another example is the incorporation of user emotions. Using affective computing to identify emotions in real-time would help us to associate the user state to concrete behaviors or responses.

New fields in which to exploit the full potential of the interaction models proposed in this thesis should be explored. One example is **model transformation and code generation**, that are being extensively used in current software engineering. This would allow us to easily create new perspectives of the data collected during the interaction process (e.g., user profiles based on QoE, statistical summaries, context/behavior evolution models, etc.) These processes can be implemented, e.g., by using the ATL Transformation Language [101] or the Acceleo Model-to-Text Language [60], which are part of EMF.

Another field to be explored are **decision systems based on live data**. The instances of the interaction models presented in this work can be accessed and analyzed at runtime while they are being created. This provides a basis for applications to easily make live decisions according to current and past interaction, context, or subjective information.

Thanks to the runtime nature of the proposed models, another field to be explored could be **continuous authentication of users**. Many works have shown examples of using

mouse and keyboards metrics to authenticate users during interaction (e.g., [190, 234]). Our intention is using interaction and context data included in the model instances to find behavior patterns with which attempting to identify users. Users security and privacy problem should be also posed and discussed.

Finally, it is worth highlighting a field in which there is still much work to be done: analysis and assessment of **interaction in multi-person**, **concurrent interfaces** and their peculiarities. These particular interfaces are used in scenarios in which multiple users are interacting with the system concurrently, e.g., in a smart home or in tabletop devices. The interaction models proposed in this work, as well as the turn-based approach in which they are based, are intended for the assessment of interaction between the system and a single user. These model designs should be reconsidered and checked whether they fit the concurrent interaction of users in multi-person scenarios.

Bibliography

[1] *HTML5 - Up and Running: Dive Into the Future of Web Development.* O'Reilly, 2010.

[2] Rasterbard Software. Luabind - Bindings between C++ and Lua. http://www.rasterbar.com/products/luabind.html, 2014. [Online; accessed May. 2014].

[3] ACM SIGCHI. Curricula for Human-Computer Interaction. http://www.acm.org/sigchi/cdg/index.html, http://www.acm.org/sigchi/cdg/cdg2.html, 1992.

[4] Gediminas Adomavicius, Bamshad Mobasher, Francesco Ricci, and Alexander Tuzhilin. Context-Aware Recommender Systems. *AI Magazine*, 32(3):67–80, 2011.

[5] Anshu Agarwal and Andrew Meyer. Beyond usability: evaluating emotional response as an integral part of the user experience. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '09, pages 2919–2930, New York, NY, USA, 2009. ACM.

[6] Pekka Aho, Nadja Menz, Tomi Räty, and Ina Schieferdecker. Automated Java GUI Modeling for Model-Based Testing Purposes. In *ITNG*, pages 268–273. IEEE Computer Society, 2011.

[7] Abdalha Ali, Abdelkader Ouda, and Luiz Fernando Capretz. A conceptual framework for measuring the quality aspects of mobile learning. *Bulletin of the IEEE Technical Committee on Learning Technology*, 14(4):31, 2012.

[8] Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. cKanren: miniKanren with Constraints. 2011.

[9] Frank M. Andrews and Stephen Bassett Withey. *Social indicators of well-being*. Plenum Press, New York, NY [u.a.], 1976.

[10] Antonella De Angeli, Jan Hartmann, and Alistair G. Sutcliffe. The Effect of Brand on the Evaluation of Websites. In Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe A. Palanque, Raquel Oliveira Prates, and Marco Winckler, editors, *INTERACT (2)*, volume 5727 of *Lecture Notes in Computer Science*, pages 638–651. Springer, 2009.

[11] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verification of Imperative Programs by Constraint Logic Program Transformation. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Festschrift for Dave Schmidt*, volume 129 of *EPTCS*, pages 186–210, 2013.

[12] Masahiro Araki, Akiko Kouzawa, and Kenji Tachibana. Proposal of a multimodal interaction description language for various interactive agents. *Trans. Inf. Syst.*, E88-D(11):2469–2476, 2005.

[13] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, ASM'03, pages 87–108, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] Sandrine Balbo, Joëlle Coutaz, and Daniel Salber. Towards automatic evaluation of multimodal user interfaces. In *Proceedings of the 1st international conference on intelligent user interfaces*, IUI '93, pages 201–208, New York, NY, USA, 1993. ACM.

[15] Lionel Balme, Alexandre Demeure, Nicolas Barralon, Joëlle Coutaz, and Gaëlle Calvary. Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In Panos Markopoulos, Berry Eggen, Emile H. L. Aarts, and James L. Crowley, editors, *EUSAI*, volume 3295 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2004.

[16] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *AIAA Journal of Aerospace Computing, Information and Communications*, 2010.

[17] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[18] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule Systems for Run-time Monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.

[19] Samuel Bayer, Laurie E. Damianos, Robyn Kozierok, and James Mokwa. The MITRE multi-modal logger: Its use in evaluation of collaborative systems. *ACM Comput. Surv.*, 31(2es), 1999.

[20] Aaron Beach, Mike Gartrell, Xinyu Xing, Richard Han, Qin Lv, Shivakant Mishra, and Karim Seada. Fusing mobile, sensor, and social data to fully enable context-aware computing. In Angela Dalton and Roy Want, editors, *HotMobile*, pages 60–65. ACM, 2010.

[21] Paolo Bellavista, Antonio Corradi, Mario Fanelli, and Luca Foschini. A survey of context data distribution for mobile ubiquitous systems. *ACM Comput. Surv.*, 44(4):24, 2012.

[22] Nicole Beringer, Ute Kartal, Katerina Louka, Florian Schiel, and U. Türk. PROMISE – A procedure for multimodal interactive system evaluation. In *Proc. Multimodal Resour. Multimodal Syst. Eval. Workshop (LREC 2002)*, pages 77–80, 2002.

[23] Isela Macia Bertran, Alessandro Garcia, and Arndt von Staa. An exploratory study of code smells in evolving aspect-oriented systems. In Paulo Borba and Shigeru Chiba, editors, *AOSD*, pages 203–214. ACM, 2011.

[24] N. Bevan. ISO 9241-11 Ergonomic Requirements for Office Work With VDTs. 1994.

[25] Eric Bodden. A lightweight LTL runtime verification tool for java. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 306–307. ACM, 2004.

[26] Eric Bodden. J-LO-A tool for runtime-checking temporal assertions. In *AOSD'08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 36–47. ACM, 2008.

[27] Eric Bodden. MOPBox: A Library Approach to Runtime Verification - (Tool Demonstration). In Sarfraz Khurshid and Koushik Sen, editors, *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 365–369. Springer, 2011.

[28] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. volume 20, pages 707–723, 2010.

[29] Cristiana Bolchini, Carlo A. Curino, Elisa Quintarelli, Fabio A. Schreiber, and Letizia Tanca. A data-oriented survey of context models. volume 36, pages 19–26, New York, NY, USA, December 2007. ACM.

[30] John Brooke. SUS: A quick and dirty usability scale. 1996.

[31] Hee Byun and Keith Cheverst. Utilizing Context History To Provide Dynamic Adaptations. *Applied Artificial Intelligence*, 18(6):533–548, 2004.

[32] Zoraida Callejas and Ramón López-Cózar. Implementing modular dialogue systems: A case of study. In *Final Workshop and ITRW on Applied Spoken Language Interaction in Distributed Environments (ASIDE 2005)*, Aalborg, Denmark, 2005.

[33] Zoraida Callejas and Ramón López-Cózar. Influence of contextual information in emotion annotation for spoken dialogue systems. *Speech Communication*, 50(5):416–433, 2008.

[34] Zoraida Callejas and Ramón López-Cózar. Relations between de-facto criteria in the evaluation of a spoken dialogue system. *Speech Communication*, 50(8-9):646–665, 2008.

[35] Zoraida Callejas and Ramón López-Cózar. Optimization of Dialog Strategies using Automatic Dialog Simulation and Statistical Dialog Management Techniques. In *INTERSPEECH*. ISCA, 2012.

[36] S.K. Card, T.P. Moran, and A. Newell. *The psychology of human-computer interaction*. CRC, 1983.

[37] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley Professional, 1997.

[38] Matt Caswell, Vijay Aravamudhan, and Kevin Wilson. Introduction to jfcUnit. *Retrieved August*, 4, 2004.

[39] Cátedra SAES Laboratories, University of Murcia, Spain. http://www.catedrasaes.org. [Online; accessed Jan. 2014].

[40] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. GUI Testing Using Computer Vision. In Elizabeth D. Mynatt, Don Schoner, Geraldine Fitzpatrick, Scott E. Hudson, W. Keith Edwards, and Tom Rodden, editors, *CHI*, pages 1535–1544. ACM, 2010.

[41] Kuan-Ta Chen, Chi-Jui Chang, Chen-Chi Wu, Yu-Chun Chang, and Chin-Laung Lei. Quadrant of euphoria: a crowdsourcing platform for qoe assessment. *IEEE Network*, 24(2):28–35, 2010.

[42] Kuan-Ta Chen, Cheng-Chun Tu, and Wei-Cheng Xiao. Oneclick: A framework for measuring network quality of experience. In *INFOCOM*, pages 702–710. IEEE, 2009.

[43] Vadim I. Chepegin and Stuart Campbell. NEXOF RA: A Reference Architecture for the NESSI Open Service Framework. *IBIS*, 8:53–56, 2009.

[44] Tanzeem Choudhury, Gaetano Borriello, Sunny Consolvo, Dirk Haehnel, Beverly Harrison, Bruce Hemingway, Jeffrey Hightower, Predrag Klasnja, Karl Koscher, Anthony LaMarca, James A. Landay, Louis LeGrand, Jonathan Lester, Ali Rahimi, Adam Rea, and Danny Wyatt. The Mobile Sensing Platform: An Embedded Activity Recognition System. *IEEE Pervasive Computing*, 7(2):32–41, 2008.

[45] Philip R. Cohen and David R. McGee. Tangible multimodal interfaces for safety-critical applications. *Commun. ACM*, 47(1):41–46, 2004.

[46] Christian Colombo. *Runtime Verification and Compensations*. PhD thesis, PhD thesis, Dept. of Computer Science, University of Malta, 2012.

[47] Adrian M. Colyer and Andy Clement. Aspect-oriented programming with AspectJ. *IBM Systems Journal*, 44(2):301–308, 2005.

[48] Sunny Consolvo and Miriam Walker. Using the experience sampling method to evaluate ubicomp applications. *IEEE Pervasive Computing*, 2(2):24–31, 2003.

[49] Constantinos K. Coursaris and Dan Kim. A Qualitative Review of Empirical Mobile Usability Studies. In Guillermo Rodríguez-Abitia and Ignacio Ania B., editors, *AMCIS*, page 352. Association for Information Systems, 2006.

[50] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction: The CARE properties. In S. A. Arnesen and D. Gilmore, editors, *Proc INTERACT'95 Conf.*, pages 115–120. Chapman & Hall Publ., 1995.

[51] Douglas Crockford. *RFC4627: JavaScript Object Notation*, 2006.

[52] Laurie E. Damianos, Jill Drury, Tari Fanderclai, Lynette Hirschman, Jeff Kurtz, and Beatrice Oshika. Evaluating multi-party multimodal systems. In *Proc. 2nd Int. Conf. Lang. Resour. Eval.*, volume 3, pages 1361–1368. MIT Media Laboratory, 2000.

[53] Sarah Diefenbach and Marc Hassenzahl. Handbuch zur Fun-ni Toolbox. manual, Folkwang Universität der Künste, 2011. Retrieved at 16.10.2013 from http://fun-ni.org/wp-content/uploads/Diefenbach+Hassenzahl_2010_HandbuchFun-niToolbox.pdf.

[54] Ergonomics of human-system interaction - Part 110: Dialogue principles (ISO 9241-110:2006), 2006.

[55] ISO DIS. 9241-210: 2010. ergonomics of human system interaction-part 210: Human-centred design for interactive systems. *International Standardization Organization (ISO). Switzerland*, 2009.

[56] Alan Dix, Janet Finlay, Gregory D. Abowd, and Russell Beale. *Human Computer Interaction*. Pearson, Harlow, England, 3 edition, 2003.

[57] Bruno Dumas, Denis Lalanne, and Rolf Ingold. Description languages for multimodal interaction: a set of guidelines and its illustration with SMUIML. *J. Multimodal User Interfaces*, 3:237–247, 2010.

[58] Laila Dybkjær, Niels Ole Bernsen, and Wolfgang Minker. Evaluation and usability of multimodal spoken language dialogue systems. *Speech Commun.*, 43:33 – 54, 2004.

[59] J.W. Eaton. *GNU Octave: a high-level interactive language for numerical computations*. Network Theory Limited, 1997.

[60] Eclipse. Acceleo, 2012.

[61] Omar el Ariss, Dianxiang Xu, Santosh Dandey, Bradley Vender, Philip E. McClean, and Brian M. Slator. A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications. In Shahram Latifi, editor, *ITNG*, pages 1038–1043. IEEE Computer Society, 2010.

[62] Klaus-Peter Engelbrecht, Michael Kruppa, Sebastian Möller, and Michael Quade. MeMo Workbench for semi-automated usability testing. In *Proc. .Interspeech 2008 incor. SST 2008*, pages 1662–1665, Brisbane, Australia, 2008. International Symposium on Computer Architecture.

[63] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety-Progress Properties. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.

[64] Michael Feathers and B Lepilleur. Cppunit cookbook, 2002.

[65] Barry Feigenbaum and Michael Squillace. Accessibility validation with RAVEN. In *WoSQ '06: Proceedings of the 2006 international workshop on Software quality*, pages 27–32, New York, NY, USA, 2006. ACM.

[66] Markus Fiedler, Sebastian Möller, and Peter Reichl. Quality of Experience: From User Perception to Instrumental Metrics (Dagstuhl Seminar 12181). *Dagstuhl Reports*, 2(5):1–25, 2012.

[67] Michael Fitzgerald. *Learning Ruby - the language that powers rails*. O'Reilly, 2007.

[68] James D. Foley and Piyawadee Noi Sukaviriya. History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation. In Fabio Paternò, editor, *DSV-IS*, pages 3–14. Springer, 1994.

[69] Jodi Forlizzi and Katja Battarbee. Understanding experience in interactive systems. In David Benyon, Paul Moody, Dan Gruen, and Irene McAra-McWilliam, editors, *Conference on Designing Interactive Systems*, pages 261–268. ACM, 2004.

[70] Norman M. Fraser. Spoken dialogue system evaluation: A first framework for reporting results. In *EUROSPEECH-1997*, pages 1907–1910, 1997.

[71] Norman M. Fraser and G.Nigel Gilbert. Simulating speech systems. *Comput. Speech Lang.*, 5(1):81–99, 1991.

[72] Steffen Göbel, Falk Hartmann, Kay Kadner, and Christoph Pohl. A Device-Independent Multimodal Mark-up Language. In Christian Hochberger and Rüdiger Liskowsky, editors, *INFORMATIK 2006. Informatik für Menschen*, volume 94 of *LNI*, pages 170–177. Gesellschaft für Informatik, 2006.

[73] Xin Guang Gong and Klaus-Peter Engelbrecht. The influence of user characteristics on the quality of judgment prediction models for tablet applications. In *10. Berliner Werkstatt*, pages 198–204, October 2013.

[74] Gnu general public license.

[75] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, pages 408–418. IEEE, 2009.

[76] H. Paul Grice. Logic and conversation. *Syntax Semantics*, 3:41–58, 1975.

[77] J. Gulliksen, B. Göransson, I. Boivie, J. Persson, S. Blomkvist, and Å. Cajander. *Key Principles for User-Centered Design.* Springer, Dordrecht, Niederlande, 2005.

[78] P. Gulur, S.W. Rodi, T.A. Washington, J.P. Cravero, G.J. Fanciullo, G.J. McHugo, and J.C. Baird. Computer Face Scale for measuring pediatric pain and mood. *The Journal of Pain*, 10(2):173–179, 2009.

[79] Sylvain Hallé, Tevfik Bultan, Graham Hughes, Muath Alkhalaf, and Roger Villemaire. Runtime Verification of Web Service Interface Contracts. volume 43, pages 59–66, 2010.

[80] H. Rex Hartson, José C. Castillo, John T. Kelso, and Wayne C. Neale. Remote Evaluation: The Network as an Extension of the Usability Laboratory. In Bonnie A. Nardi, Gerrit C. van der Veer, and Michael J. Tauber, editors, *CHI*, pages 228–235. ACM, 1996.

[81] R.A. Hassad. Faculty attitude toward technology-assisted instruction for introductory statistics in the context of educational reform. In *IASE 2012. Technology in Statistics Education: Virtualities and Realities*, Cebu City, The Philippines, 2012.

[82] Marc Hassenzahl. The Effect of Perceived Hedonic Quality on Product Appealingness. *International Journal of Human-Computer Interaction*, 13(4):481–499, 2001.

[83] Marc Hassenzahl, Michael Burmester, and Franz Koller. Attrakdiff: Ein fragebogen zur messung wahrgenommener hedonischer und pragmatischer qualität. (a questionnaire for measuring perceived hedonic and pragmatic quality). In Gerd Szwillus and Jürgen Ziegler, editors, *Mensch & Computer*, pages 187–196. Teubner, 2003.

[84] Marc Hassenzahl and Andrew Monk. The Inference of Perceived Usability From Beauty. *International Journal of Human-Computer Interaction*, 25(3):235–260, 2010.

[85] Jane Huffman Hayes and Jeff Offutt. Input validation analysis and testing. *Empirical Software Engineering*, 11(4):493–522, 2006.

[86] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In Friedemann Mattern and Mahmoud Naghshineh, editors, *Pervasive Computing*, volume 2414 of *Lecture Notes in Computer Science*, pages 79–117. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45866-2_14.

[87] Kate S. Hone and Robert Graham. Towards a tool for the Subjective Assessment of Speech System Interfaces (SASSI). *Natural Language Engineering*, 6(3&4):287–303, 2000.

[88] Jong-yi Hong, Eui-ho Suh, and Sung-Jin Kim. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509–8522, 2009.

[89] Shi-Jinn Horng, Ming-Yang Su, and Ja-Ga Tsai. A Dynamic Backdoor Detection System Based on Dynamic Link Libraries. *International Journal of Business and Systems Research*, 2(3):244–257, 2008.

[90] T. Husted and V. Massol. *JUnit in Action*. Manning Publications, 2004.

[91] Selim Ickin, Katarzyna Wac, Markus Fiedler, Lucjan Janowski, Jin-Hyuk Hong, and Anind K. Dey. Factors influencing quality of experience of commonly used mobile applications. *IEEE Communications Magazine*, 50(4):48–56, 2012.

[92] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, December 1990.

[93] IEEE. *IEEE-STD-610 ANSI/IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology*. IEEE, February 1991.

[94] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Lua.org, 2006.

[95] Institute of Electrical and Electronics Engineers. IEEE 1012-2004 - IEEE Standard for Software Verification and Validation. pages 0–110. IEEE, IEEE, June 2005. Revision of IEEE Std 1012-1998.

[96] Zakwan Jaroucheh, Xiaodong Liu, and Sally Smith. Recognize contextual situation in pervasive environments using process mining techniques. *J. Ambient Intelligence and Humanized Computing*, 2(1):53–69, 2011.

[97] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 1427–1430. IEEE, 2012.

[98] Peter Johnson, Stephanie Wilson, Panos Markopoulos, and James Pycock. ADEPT: Advanced Design Environment for Prototyping with Task Models. In Stacey Ashlund, Kevin Mullet, Austin Henderson, Erik Hollnagel, and Ted N. White, editors, *INTERCHI*, page 56. ACM, 1993.

[99] Michael Johnston. EMMA: Extensible MultiModal Annotation markup language. W3C recommendation, W3C, February 2009. http://www.w3.org/TR/2009/REC-emma-20090210/.

[100] Jonathan Turner and Jason Turner. ChaiScript: Easy to use scripting for C++. http://www.chaiscript.com/, 2012. [Online; accessed May. 2014].

[101] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[102] D. Kahneman, E. Diener, and N. Schwarz. *Well-being: The foundations of hedonic psychology*. Russell Sage Foundation Publications, 2003.

[103] J. Wolfgang Kaltz, Jürgen Ziegler, and Steffen Lohmann. Context-aware Web Engineering: Modeling and Applications. *Revue d'Intelligence Artificielle*, 19(3):439–458, 2005.

[104] Marcel R. Karam, Sergiu M. Dascalu, and Rami H. Hazimé. Challenges and Opportunities for Improving Code-Based Testing of Graphical User Interfaces. *Journal of Computational Methods in Sciences and Engineering*, 6(5-6):379–388, 2006.

[105] Orlando Karam and Richard Halstead-Nussloch. Introducton to Android development. *J. Comput. Sci. Coll.*, 28(2):224–224, December 2012.

[106] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ESEC / SIGSOFT FSE*, page 313, 2001.

[107] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. *Electr. Notes Theor. Comput. Sci.*, 70(4):95–111, 2002.

[108] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[109] J. Kirakowski and M. Corbett. SUMI: The software usability measurement inventory. *British journal of educational technology*, 24(3):210–212, 1993.

[110] Michael A. Kittel and Geoffrey T. LeBlond. *ASP.NET cookbook - the ultimate ASP.NET code sourcebook: covers ASP.NET 1.1.* O'Reilly, 2004.

[111] Dierk König. JavaFX (Second edition). *Java Magazin*, (2):32–38, 2012.

[112] Hannu Korhonen, Juha Arrasvuori, and Kaisa Väänänen-Vainio-Mattila. Analysing user experience of personal mobile products through contextual factors. In Marios C. Angelides, Lambros Lambrinos, Michael Rohs, and Enrico Rukzio, editors, *MUM*, page 11. ACM, 2010.

[113] Alfred Kranstedt, Stefan Kopp, and Ipke Wachsmuth. MURML: A multimodal utterance representation markup language for conversational agents. In *Proc. AAMAS02 Workshop Embodied Conversat. Agents - let's specify and evaluate them*, 2002.

[114] Christine Kühnel, Benjamin Weiss, and Sebastian Möller. Parameters Describing Multimodal Interaction - Definitions and Three Usage Scenarios. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *Proceedings of the 11th Annual Conference of the ISCA (Interspeech 2010)*, pages 2014–2017, Makuhari, Japan, 2010. ISCA.

[115] Viktor Kuncak. *Modular Data Structure Verification.* PhD thesis, 2007.

[116] Fenareti Lampathaki, Yannis Charalabidis, Spyros Passas, David Osimo, Melanie Bicking, Maria Wimmer, and Dimitris Askounis. Defining a Taxonomy for Research Areas on ICT for Governance and Policy Modelling. In Maria Wimmer, Jean-Loup Chappelet, Marijn Janssen, and Hans Jochen Scholl, editors, *EGOV*, volume 6228 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2010.

[117] P Lang. The cognitive Psychophysiology of emotion: anxiety and the anxiety disorders. 1985.

[118] James A. Larson, David Raggett, and T. V. Raman. W3C multimodal interaction framework. W3C note, W3C, May 2003. http://www.w3.org/TR/2003/NOTE-mmi-framework-20030506/.

[119] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[120] Geoffrey Leech and Andrew Wilson. EAGLES. recommendations for the morphosyntactic annotation of corpora. http://www.ilc.cnr.it/EAGLES96/annotate/annotate.html, 1996.

[121] Saija Lemmelä, Akos Vetek, Kaj Mäkelä, and Dari Trendafilov. Designing and evaluating multimodal interaction for mobile contexts. In Vassilios Digalakis, Alexandros Potamianos, Matthew Turk, Roberto Pieraccini, and Yuri Ivanov, editors, *Proc. 10th Int. Conf. Multimodal Interfaces*, pages 265–272, New York, NY, USA, 2008. ACM.

[122] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *ICSE*, pages 116–125. ACM, 2000.

[123] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing Non-safety Security Policies with Program Monitors. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 2005.

[124] James Lin and James A. Landay. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In Mary Czerwinski, Arnold M. Lund, and Desney S. Tan, editors, *CHI*, pages 1313–1322. ACM, 2008.

[125] Ramón López-Cózar and Zoraida Callejas. Multimodal dialogue for ambient intelligence and smart environments. In *Handbook of ambient intelligence and smart environments*, pages 559–579. Springer, 2010.

[126] Ramón López-Cózar Delgado and Masahiro Araki. *Spoken, Multilingual and Multimodal Dialogue Systems: Development and Assessment.* Wiley, Chichester, UK, 2005.

[127] Sascha Mahlke and Manfred Thüring. Studying antecedents of emotional experiences in interactive contexts. In *Computer Human Interaction*, pages 915–918, 2007.

[128] Ashok Malhotra and Paul V. Biron. XML Schema Part 2: Datatypes Second Edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[129] Marco Manca and Fabio Paternó. Supporting multimodality in service-oriented model-based development environments. In Regina Bernhaupt, Peter Forbrig, Jan Gulliksen, and Marta Lárusdóttir, editors, *HCSE*, volume 6409 of *LNCS*, pages 135–148. Springer, 2010.

[130] Tony Manninen. Rich interaction in the context of networked virtual environments—Experiences gained from the multi-player games Domain. In *People and Computers XV—Interaction without Frontiers*, pages 383–398. Springer, 2001.

[131] Evan Martin, Suranjana Basu, and Tao Xie. Automated Testing and Response Analysis of Web Services. In *ICWS*, pages 647–654. IEEE Computer Society, 2007.

[132] Jean-Claude Martin and Michael Kipp. Annotating and Measuring Multimodal Behaviour - Tycoon Metrics in the Anvil Tool. In *LREC*. European Language Resources Association, 2002.

[133] Pedro Luis Mateo Navarro. The Open HMI Tester. http://www.catedrasaes.org/wiki/OHT, 2011. [Online; accessed Jan. 2014].

[134] Pedro Luis Mateo Navarro. Android HCI Extractor. http://code.google.com/p/android-hci-extractor, 2012. [Online; accessed Jan. 2014].

[135] Pedro Luis Mateo Navarro. CARIM: A Context-aware Ratings Interaction Model for QoE analysis. http://www.catedrasaes.org/wiki/Carim, 2014. [Online; accessed Jan. 2014].

[136] Pedro Luis Mateo Navarro. CARIM metamodel implementation. https://github.com/pedromateo/carim, 2014. [Online; accessed Jan. 2014].

[137] Pedro Luis Mateo Navarro and Stefan Hillmann. Model-based Measurement of Human-Computer Interaction in Mobile Multimodal Environments. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction NordiCHI 2012*. ACM, 2012.

[138] Pedro Luis Mateo Navarro and Stefan Hillmann. PALADIN metamodel implementation. https://github.com/pedromateo/paladin, 2012. [Online; accessed Jan. 2014].

[139] Pedro Luis Mateo Navarro and Stefan Hillmann. PALADIN: a Run-time Model for Automatic Evaluation of Multimodal Interfaces. http://www.catedrasaes.org/wiki/MIM, 2013. [Online; accessed Jan. 2014].

[140] Pedro Luis Mateo Navarro and Francisco J. López. S-DAVER: Script-based Data Verification Framework. http://www.catedrasaes.org/wiki/GuiVerification, 2014. [Online; accessed Jan. 2014].

[141] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, and Diego Sevilla Ruiz. Towards Software Quality and User Satisfaction through User Interfaces. In *ICST*, pages 415–418. IEEE Computer Society, 2011.

[142] Pedro Luis Mateo Navarro, Gregorio Martínez Pérez, and Diego Sevilla Ruiz. A Context-aware Interaction Model for the Analysis of Users QoE in Mobile Environments. *International Journal of Human-Computer Interaction, Taylor & Francis, in press.*, 2014.

[143] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. Aplicación de Open HMI Tester como framework open-source para herramientas de pruebas de software. *REICIS: Revista Española de Innovación, Calidad e Ingeniería del Software*, 5(4), December 2009.

[144] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. Automated GUI Testing Validation Guided by Annotated Use Cases. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *GI Jahrestagung*, volume 154 of *LNI*, pages 2796–2804. GI, 2009.

[145] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. Open HMI Tester: un Framework Open-source para Herramientas de Pruebas de Software. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, 3(4), 2009.

[146] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. A Proposal for Automatic Testing of GUIs Based on Annotated Use Cases. *Adv. Software Engineering, Special Issue on Software Test Automation*, 2010.

[147] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. Open HMI Tester: An Open and Cross-Platform Architecture for GUI Testing and Certification. *International Journal of Computer Systems Science and Engineering (IJCSSE), Special Issue on Open Source Certification*, 25(4):283–296, July 2010.

[148] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. Verificación de Datos en la GUI como un Aspecto Separado de las Aplicaciones. *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, 2010.

[149] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. A Context-aware Model for the Analysis of User Interaction and QoE in Mobile Environments. In *CENTRIC 2012, The Fifth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services. Lisbon, Portugal*, pages 124–127, 2012.

[150] Pedro Luis Mateo Navarro, Diego Sevilla Ruiz, and Gregorio Martínez Pérez. A Context-aware Model for QoE Analysis in Mobile Environments. In *XIV International Congress on Human-Computer Interaction, Madrid (Spain)*, 2013.

[151] Mateo Navarro, Pedro Luis. OpenHMI-Tester Prototype. http://sourceforge.net/projects/openhmitester, 2009. [Online; accessed Jan. 2014].

[152] Mateo Navarro, Pedro Luis and López, Francisco J. S-DAVER: Script-based Data Verification for Qt GUIs. http://www.catedrasaes.org/wiki/GuiVerification, 2014. [Online; accessed Jan. 2014].

[153] Atif Memon. GUI Testing: Pitfalls and Process. *IEEE Computer*, 35(8):87–88, 2002.

[154] Atif Memon. An event-flow model of GUI-based applications for testing. *Software Testing Verification and Reliability*, 17(3):137–157, 2007.

[155] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *WCRE*, pages 260–269. IEEE Computer Society, 2003.

[156] Atif M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, Department of Computer Science, University of Maryland, 2001.

[157] Atif M. Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A Framework for Regression Testing "Nightly/daily Builds" of GUI Applications. In *ICSM*, pages 410–419. IEEE Computer Society, 2003.

[158] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *ESEC / SIGSOFT FSE*, pages 256–267, 2001.

[159] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Software Eng*, 31(10):884–896, 2005.

[160] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. volume 14, pages 249–289. Springer, 2012. http://dx.doi.org/10.1007/s10009-011-0198-6.

[161] Jan Meskens, Jo Vermeulen, Kris Luyten, and Karin Coninx. Gummy for multi-platform user interface designs: shape multiply fix use me. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '08, pages 233–240, New York, NY, USA, 2008. ACM.

[162] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25:40–51, 1992.

[163] M. Minge. Dynamics of user experience. In *Positionspapier im Workshop" Research Goals and Strategies for Studying User Experience and Emotion", NordiCHI, Lund, Schweden*, 2008.

[164] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams JAVA. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 2–8, New York, NY, USA, 2006. ACM.

[165] Karan Mitra, Arkady B. Zaslavsky, and Christer Ahlund. A probabilistic context-aware approach for quality of experience measurement in pervasive systems. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 419–424. ACM, 2011.

[166] Sebastian Möller. Parameters describing the interaction with spoken dialogue systems, October 2005. Based on ITU-T Contr. COM 12-17 (2009).

[167] Sebastian Möller. *Quality of Telephone-Based Spoken Dialogue Systems*. Springer, New York, United States, 2005.

[168] Sebastian Möller. Parameters Describing the Interaction with Multimodal Dialogue Systems. ITU-T Recommendation Supplement 25 to P-Series Rec., International Telecommunication Union, Geneva, Switzerland, January 2011.

[169] Sebastian Möller, Klaus-Peter Engelbrecht, Christine Kühnel, Ina Wechsung, and Benjamin Weiss. A Taxonomy of Quality of Service and Quality of Experience of Multimodal Human-Machine Interaction. In *First International Workshop on Quality of Multimedia Experience (QoMEX'09)*, pages 7–12, July 2009.

[170] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, Hoboken, N.J., 2012.

[171] Ravikanth Nasika and Partha Dasgupta. Transparent Migration of Distributed Communicating Processes. In *13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*, Las Vegas, Nevada, USA, November 2000.

[172] A.B. Naumann and I. Wechsung. Developing usability methods for multimodal systems: The use of subjective and objective measures. In *International Workshop on Meaningful Measures: Valid Useful User Experience Measurement (VUUM)*, page 8. Citeseer, 2008.

[173] Laurence Nigay and Joëlle Coutaz. A design space for multimodal systems: Concurrent processing and data fusion. In Stacey Ashlund, Kevin Mullet, Austin Henderson, Erik Hollnagel, and Ted N. White, editors, *Proc. INTERACT '93 and CHI '93 conf. Human factors in comput. syst.*, pages 172–178, New York, NY, USA, 1993. ACM.

[174] Nokia Corporation. Qt : cross-platform application and UI framework, 2012.

[175] M. Ogertschnig and H. van der Heijden. A short-form measure of attitude towards using a mobile information service. In *Proceedings of the 17th Bled Electronic Commerce Conference, Bled*, 2004.

[176] Héctor Olmedo-Rodríguez, David Escudero-Mancebo, and Valentín Cardeñoso Payo. Evaluation proposal of a framework for the integration of multimodal interaction in 3D worlds. In *Proc. 13th Int. Conf. Human-Comput. Interact. Part II: Novel Interact. Methods Techniques*, pages 84–92, Berlin, Heidelberg, 2009. Springer-Verlag.

[177] OMG. *Unified Modeling Language: Superstructure, version 2.1.1.* Object Modeling Group, February 2007.

[178] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[179] Matt Oshry, Paolo Baggia, Kenneth Rehor, Milan Young, Rahul Akolkar, Xu Yang, Jim Barnett, Rafah Hosn, RJ Auburn, Jerry Carter, Scott McGlashan, Michael Bodell, and Daniel C. Burnett. Voice extensible markup language (VoiceXML) 3.0. W3C working draft, W3C, December 2009. http://www.w3.org/TR/2009/WD-voicexml30-20091203/.

[180] Sharon Oviatt. Ten myths of multimodal interaction. *Commun. ACM*, 42:74–81, November 1999.

[181] Sharon Oviatt. Advances in robust multimodal interface design. *IEEE Comput. Graph. Appl.*, 23:62–68, September 2003.

[182] Ana C. R. Paiva, João C. P. Faria, and Raul F. A. M. Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. *Electr. Notes Theor. Comput. Sci.*, 190(2):99–111, 2007.

[183] Philippe A. Palanque and Amelie Schyn. A model-based approach for engineering multimodal interactive systems. In Matthias Rauterberg, Marino Menozzi, and Janet Wesson, editors, *INTERACT'03*, pages 543–550. IOS Press, 2003.

[184] Fabio Paternò, Carmen Santoro, and Lucio D. Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4):1–30, Nov 2009.

[185] Manolis Perakakis and Alexandros Potamianos. The effect of input mode on inactivity and interaction times of multimodal systems. In Dominic W. Massaro, Kazuya Takeda, Deb Roy, and Alexandros Potamianos, editors, *Proc. 9th Int. Conf. Multimodal Interfaces (ICMI 2007)*, pages 102–109. ACM, 2007.

[186] Manolis Perakakis and Alexandros Potamianos. Multimodal system evaluation using modality eficiency and synergy metrics. In *Proc. 10th Int. Conf. Multimodal Interfaces ((ICMI'08)*, pages 9–16. ACM, October 2008.

[187] Shelley Powers. *JavaScript Cookbook - Programming the Web.* O'Reilly, 2010.

[188] Python Software Foundation. The Jython Project. http://www.jython.org/, 2014. [Online; accessed May. 2014].

[189] C.M. Ray, C. Sormunen, and T.M. Harris. Men's and women's attitudes toward computer technology: A comparison. *Office Systems Research Journal*, 17:1–8, 1999.

[190] Revett Kenneth, Jahankhani Hamid, Magalhães SérgioTenreiro, and Santos HenriqueM.D. *A Survey of User Authentication Based on Mouse Dynamics*, volume 12 of *Communications in Computer and Information Science*, pages 210–219. Springer Berlin Heidelberg, 2008.

[191] Martin Rinard. From Runtime Verification to Runtime Intervention and Adaptation. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 276–276. Springer Berlin Heidelberg, 2013.

[192] Michiel Ronsse and Koenraad De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[193] V. Roto. Web browsing on mobile phones-characteristics of user experience. *Helsinki University of Technology*, 2006.

[194] C. Ryan and A. Gonsalves. The effect of context and application type on mobile usability: an empirical study. In *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*, page 115–124, 2005.

[195] Jost Schatzmann, Kallirroi Georgila, and Steve Young. Quantitative evaluation of user simulation techniques for spokendialogue systems. In Laila Dybkjær and Wolfgang Minker, editors, *Proc. 6th SIGdial Workshop Discourse Dialogue*, pages 45–54. Special Interest Group on Discourse and Dialogue (SIGdial), Associtation for Computational Linguistics (ACL), 2005.

[196] Jost Schatzmann and Steve Young. The hidden agenda user simulation model. *IEEE Trans. Audio Speech Lang. Process.*, 17(4):733–747, 2009.

[197] Stefan Schmidt, Klaus-Peter Engelbrecht, Matthias Schulz, Martin Meister, Julian Stubbe, Mandy Töppel, and Sebastian Möller. Identification of interactivity sequences in interactions with spoken dialog systems. In *Proc 3rd Int. Workshop Percept. Qual. Syst.*, pages 109–114. Chair of Communication Acoustics TU Dresden, 2010.

[198] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[199] Katrin Schulze and Heidi Krömker. A framework to measure user experience of interactive online products. In *Proceedings of the 7th International Conference on Methods and Techniques in Behavioral Research*, page 14. ACM, 2010.

[200] Marcos Serrano and Laurence Nigay. A Wizard of Oz Component-Based Approach for Rapidly Prototyping and Testing Input Multimodal Interfaces. *Journal on Multimodal User Interfaces*, 3(3):215–225, 2010.

[201] Marcos Serrano, Laurence Nigay, Rachel Demumieux, Jérôme Descos, and Patrick Losquin. Multimodal interaction on mobile phones: Development and evaluation using ACICARE. In Marko Nieminen and Mika Röykkee, editors, *MobileHCI '06: Proc. 8th Conf. Human-comput. interact. mob. devices serv.*, pages 129–136, New York, NY, USA, 2006. ACM.

[202] Sociedad Anónima de Electrónica Submarina. http://www.electronica-submarina.com. [Online; accessed Jun. 2014].

[203] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Merks Ed. *EMF. Eclipse Modeling Framwork Second Edition.* Addison-Wesley, Upper Saddle River, NJ, 2 edition, 2009.

[204] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *ISSTA*, pages 158–167, 2000.

[205] Janienke Sturm, Ilse Bakx, Bert Cranen, Jacques Terken, and Fusi Wang. Usability evaluation of a dutch multimodal system for train timetable information. In Manual Gonzales Rodriguez and Carmen Suarez Araujo, editors, *Proc. LREC 2002. 3rd Int. Conf. Lang. Resour. Eval.*, pages 255–261, 2002.

[206] Telekom Innovation Laboratories, Berlin, Germany. http://www.laboratories.telekom.com/. [Online; accessed May. 2014].

[207] The Eclipse Foundation. Eclipse modeling framework. http://www.eclipse.org/emf/, 2007.

[208] The GTK+ Team. The GIMP Toolkit (GTK), version 2.x. http://www.gtk.org, 2012. [Online; accessed Jan. 2014].

[209] Henry S. Thompson, Murray Maloney, David Beech, and Noah Mendelsohn. XML schema part 1: Structures second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[210] B. Thurnher, T. Grill, K. Hummel, and R. Weigl. Exploiting context-awareness for usability evaluation in mobile HCI. *Proceedings of Usability Day IV. Pabst Science Publisher*, 9:109–113, 2006.

[211] Trolltech Inc. Trolltech Qt 4.x Demo Applications. http://doc.trolltech.com/4.0/, 2012. [Online; accessed Jan. 2014].

[212] Gerrit C. Van Der Veer, Bert F. Lenting, and Bas A. J. Bergevoet. GTA: Groupware Task Analysis - Modeling Complexity. *Acta Psychologica*, 91:297–322, 1996.

[213] Tim van Kasteren, Gwenn Englebienne, and Ben J. A. Kröse. An activity monitoring system for elderly care using generative and discriminative models. *Personal and Ubiquitous Computing*, 14(6):489–498, 2010.

[214] Davy Vanacken, Joan De Boeck, Chris Raymaekers, and Karin Coninx. NIMMIT: A notation for modeling multimodal interaction techniques. In José Braz, Joaquim A. Jorge, Miguel Dias, and Adérito Marcos, editors, *GRAPP*, pages 224–231. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, 2006.

[215] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of GUI Testing Using a Model-driven Approach. In Hong Zhu, Joseph R. Horgan, Shing-Chi Cheung, and J. Jenny Li, editors, *AST*, pages 9–14. ACM, 2006.

[216] Katarzyna Wac, Selim Ickin, Jin H. Hong, Lucjan Janowski, Markus Fiedler, and Anind K. Dey. Studying the experience of mobile applications used in different contexts of daily life. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*, W-MUST '11, pages 7–12, New York, NY, USA, August 2011. ACM.

[217] Marilyn Walker, Diane Litman, Candace Kamm, and Alicia Abella. PARADISE: A framework for evaluating spoken dialogue agents. In *Proc. 35th Annu. Meet. Assoc. Comput. Linguist.*, pages 262–270. ACL 97, July 1997.

[218] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, Reading, MA, 1999.

[219] Ina Wechsung, Klaus-Peter Engelbrecht, Anja B. Naumann, Stefan Schaffer, Julia Seebode, Florian Metze, and Sebastian Möller. Predicting the quality of multimodal systems based on judgments of single modalities. In *INTERSPEECH*, pages 1827–1830. ISCA, 2009.

[220] Ina Wechsung and Anja Naumann. Evaluation methods for multimodal systems: A comparison of standardized usability questionnaires. In Elisabeth André, Laila Dybkjaer, Wolfgang Minker, Heiko Neumann, Roberto Pieraccini, and Michael Weber, editors, *PIT*, volume 5078 of *Lecture Notes in Computer Science*, pages 276–284. Springer, 2008.

[221] Ina Wechsung, Matthias Schulz, Klaus-Peter Engelbrecht, Julia Niemann, and Sebastian Möller. All users are (not) equal - the influence of user characteristics on perceived quality, modality choice and performance. In Ramón López-Cózar and Tetsunori Kobayashi, editors, *Proceedings of the Paralinguistic Information and its Integration in Spoken Dialogue Systems Workshop*, pages 175–186. Springer New York, 2011.

[222] Benjamin Weiss, Sebastian Möller, Ina Wechsung, and Christine Kühnel. Quality of experiencing multi-modal interaction. In *Spoken Dialogue Systems Technology and Design*, pages 213–230. Springer, 2011.

[223] Lee White and Husain Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. *IEEE Transactions on SMC Associate Editors*, pages 110–123, 2000.

[224] Lee White, Husain Almezen, and Nasser Alzeidi. User-Based Testing of GUI Sequences and Their Interactions. *IEEE | 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 54–65, 2001.

[225] Heli Wigelius and Heli Väätäjä. Dimensions of context affecting user experience in mobile work. In Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe A. Palanque, Raquel Oliveira Prates, and Marco Winckler, editors, *INTERACT (2)*, volume 5727 of *Lecture Notes in Computer Science*, pages 604–617. Springer, 2009.

[226] D. Wong and C. Baker. Pain in children: comparison of assessment scales. *IJI*, 50:7, 1988.

[227] Wanmin Wu, Md. Ahsan Arefin, Raoul Rivas, Klara Nahrstedt, Renata M. Sheppard, and Zhenyu Yang. Quality of experience in distributed interactive multimedia environments: toward a theoretical framework. In Wen Gao, Yong Rui, Alan Hanjalic, Changsheng Xu, Eckehard G. Steinbach, Abdulmotaleb El-Saddik, and Michelle X. Zhou, editors, *ACM Multimedia*, pages 481–490. ACM, 2009.

[228] Qing Xie and Atif M. Memon. Studing the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Envolving Software. *IEEE Computer Society*, 2005.

[229] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.

[230] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: using GUI screenshots for search and automation. In Andrew D. Wilson and François Guimbretière, editors, *UIST*, pages 183–192. ACM, 2009.

[231] Xun Yuan and Atif M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE*, pages 396–405. IEEE Computer Society, 2007.

[232] Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime Checking for Program Verification. In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2007.

[233] D. Zhang and B. Adipat. Challenges, methodologies, and issues in the usability testing of mobile applications. *International Journal of Human-Computer Interaction*, 18(3):293–308, 2005.

[234] Yu Zhong, Yunbin Deng, and Anil K Jain. Keystroke dynamics for user authentication. In *CVPR Workshops*, pages 117–123. IEEE, 2012.

A

List of Acronyms

AHE Android HCI Extractor

ANSI American National Standards Institute

AOP Aspect-oriented Programming

API Application Programming Interface

App Application

ASR Automatic Speech Recognition

ASW Anti-submarine Warfare

ATT Attractiveness

AVM Attribute Malue Matrix

BSD Berkeley Software Distribution

CARIM Context-Aware and Ratings Interaction Model

CLP Constraint Logic Programming

DEB Debian Software Package

DLL Dynamic Link Library

DOM Document Object Model

EMF Eclipse Modeling Framework

EVE Execution and Verification Environment

FOSS Free and Open-Source Software

GNU GNU's Not Unix

GPL GNU General Public License

GPS Global Positioning System

GQL Generic Questionnaire Library

GTK GNU Image Manipulation Program Toolkit

GUI Graphical User Interface

HCI Human-Computer Interaction

HMI Human-Machine Interface

HQ Hedonic quality

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

ITU International Telecommunication Union

ITU-T ITU Telecommunication Standardization Sector

LAN Local Area Network

LGPL GNU Lesser General Public License

LTL Linear-time Temporal Logic

MMDS Multimodal Dialog Systems

MMI Multimodal Human-Computer Interaction

MPA Maritime Patrol Aircraft

OCL Object Constraint Language

OHT Open HMI Tester

PALADIN Practice-oriented Analysis and Description of Multimodal Interaction

PhD Philosophiae Doctor (Doctor of Philosophy)

PQ Pragmatic Quality

QoE Quality of Experience

QoS Quality of Service

RPM Red Hat Package Manager

RV Runtime Verification

S-DAVER Script-based Data Verification

SAES Sociedad Anónima de Electrónica Submarina

SD Standard Deviation

SDK Software Development Kit

SDS Spoken Dialog Systems

SRS Software Requirements Specification

UML Unified Modeling Language

UMU University of Murcia

UX User Experience

VS Verification Step

VV Validation and Verification

WoZ Wizard-of-Oz

XML Extensible Markup Language

XSD XML Schema Definition