# A model-based architecture for the usability analysis and correction of user interfaces

José Manuel Cruz Zapata    Jesús García Molina
Diego Sevilla Ruiz

October 17, 2016

**Abstract**

Usability has become a key aspect in the design of graphical user interfaces (GUI) and its evaluation is essential to guarantee the quality of a software. Most usability evaluation techniques depend on the support of experts or at least require some training in usability from the members of the development team. Since this task entails significant effort, its automation would suppose notable resource savings. This paper presents a model driven architecture for the automatic usability analysis and correction. A guideline review evaluation process is performed by automatically applying a set of usability rules to the GUI. These rules do not only detect usability defects but also support the definition of correction methods, so that automatic usability correction is provided as well. We separate the evaluation logic from the engine allowing the user to define its own usability rules catalogues. At the same time, models enable technology independence by introducing an additional level of abstraction, making the solution suitable for different GUI platforms. For this paper, we present the integration with Android interfaces as an example of the applicability of the architecture.

## 1 Introduction

Consideration of human factors in the development of a Graphical User Interface (GUI) is a key factor for the success of a software application. Usability is a fundamental concept in the Human Computer Interaction (HCI) field, and its evaluation has become a topic of interest and concern over the last few years. A wide variety of usability evaluation methods has been proposed [16] and most of them require a manual application, frequently involving usability experts. Furthermore, studies have shown that the application of different techniques often produces different results, and even the same evaluation method can obtain different outcomes depending on the evaluator [13, 14, 10, 16]. Automation of usability evaluation arises as a potential solution to these problems [7], reducing the costs and effort and providing some systematicity in the results obtained.

Over the last decade Model Driven Software Engineering (MDSE or MDE) has established as one of the fundamental paradigms in automating software development [17, 19]. MDE proposes a systematic use of models in the construction of software through two main techniques: metamodeling and model transformation [3, 15]. MDE has proven to be useful not only in the development of new applications but also in software reengineering or modernization.

This paper presents a model driven architecture for the automatic usability analysis and correction. A guideline review evaluation process is performed by automatically applying a set of usability rules to the GUI. These rules do not only detect usability defects but we also support the definition of correction methods, so that automatic usability correction is provided as well. We separate the evaluation logic from the engine allowing the user to define its own usability rules catalogues. Models also enable technology independence by introducing an additional level of abstraction, making the solution applicable to different GUI platforms.

The rest of the paper is organized as follows. Section 2 presents a global view of the architecture. In Section 3 we describe the metamodels defined to represent the GUI concept during the evaluation process. Section 4 introduces the DSL defined to express usability rules. The analysis and correction processes are explained in Section 5. Section 6 describes the integration of the Android platform into the architecture. In Section 7 an example is defined to illustrate the usability evaluation and correction of Android GUIs. Finally, Section 8 describes the state of the art with regard of automatic usability evaluation and Section 9 ends with some conclusions and future work.

## 2 Overview of the solution

This section presents a high-level view of our approach, which consists of two main components: the one in charge of the usability analysis of the GUI and the one responsible for the automatic correction of the usability defects.

As our solution is designed following the principles of MDE, models represent both the input and the output of the process. Consequently, it becomes necessary to obtain a model from the GUI platform's export format. In fact, this step must be bi-directional, so that we must also be able to obtain the GUI's original format from the model. We use the term injection to refer to the former process and the term extraction to refer to the latter.

In order to achieve technology independence, we must work at a higher level of abstraction. We define a GUI metamodel that represents a generalization of the GUI concept. Both usability analysis and correction are performed on this generic GUI representation, achieving technology independence at the core of the solution. The introduced level of abstraction requires an additional step in the application of the solution: the transformation that allows us to obtain a generic GUI model from the platform-specific one. Once again, this step must be bi-directional, so that the corrections made on the generic GUI model reflect on the platform-specific model and, eventually, on the GUI in its original format. In this document, we present the implementation of these steps for the Android platform, but the architecture allows the injection of any GUI platform.

Once we reach technology independence, usability analysis and correction are performed on the GUI. The model-based structure of the two components is very similar and is shown in Figure 1. The analysis/correction is performed in two phases. First, we obtain a usability-oriented representation of the GUI by a model to model transformation. The purpose of this intermediate model is to facilitate the application of the usability rules. Then we do the usability analysis/correction, which takes the usability rules and the GUI as input and generates a usability defects model as output. In case of correction, the input GUI model is modified during the process. It is worth noting that the model to model transformation implementing the analysis/correction is in turn generated by a model to text transformation. This is one of the most interesting design choices and it will be motivated and explained in Section 5.2. Finally, a report is generated from the defects model by a model to text transformation, informing the user about the usability defects found and, in case of correction, the result of the correction process.

An important feature of the architecture is its flexibility in the definition of usability rules. The catalogue of rules to be applied to the GUI is an input to the analysis/correction system, thus allowing the user to create and apply its own usability rules. We have defined a reduced set of usability rules as a proof of concept of the architecture. These rules fundamentally refer to the key aspects of a GUI: important attributes such as the color, text or position of the elements.

Throughout this document we will focus on one of these usability rules, the label-input separation rule, stated as follows: "The separation between an input field and its corresponding label must not be greater than 30px". Labels and inputs compound the forms that populate most GUIs, and therefore several usability rules referring to these elements have been defined. A right separation between a label and its input is vital for the user to correctly perceive the relation between these two components and thus properly understand the GUI. We will make use of this usability rule to illustrate how the presented architecture supports the definition and application of such rules.

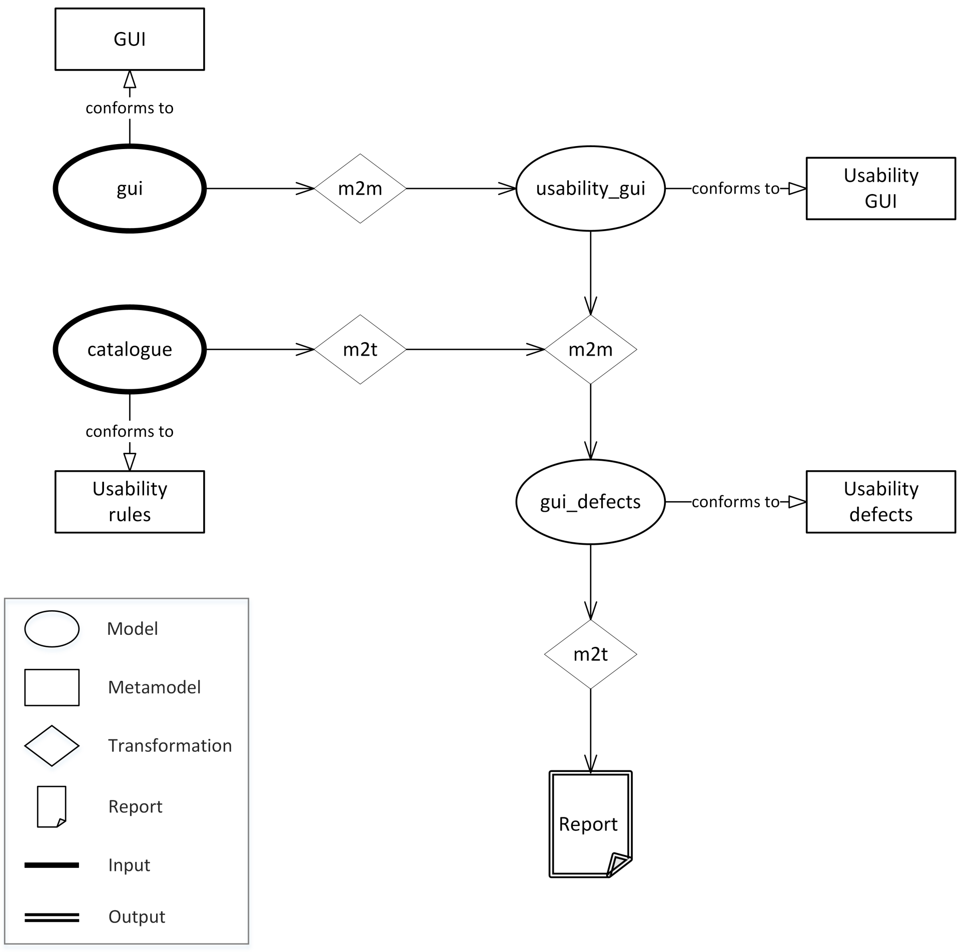

Figure 1: Basic model-based structure of the analysis and correction processes.

## 3 Interface metamodels

In this section we introduce the metamodels used for representing a GUI, presenting their motivation and purpose and the equivalences between them.

### 3.1 Generic GUI metamodel

Platform independence is achieved through an intermediate metamodel that represents the GUI abstraction used by the system. Simplification is an additional advantage of introducing this metamodel. Since it must be able to represent any kind of GUI, the metamodel considers only the most important and common concepts involved, making the definition of new usability rules a simpler task. Figure 2 shows an excerpt of the generic GUI metamodel defined.

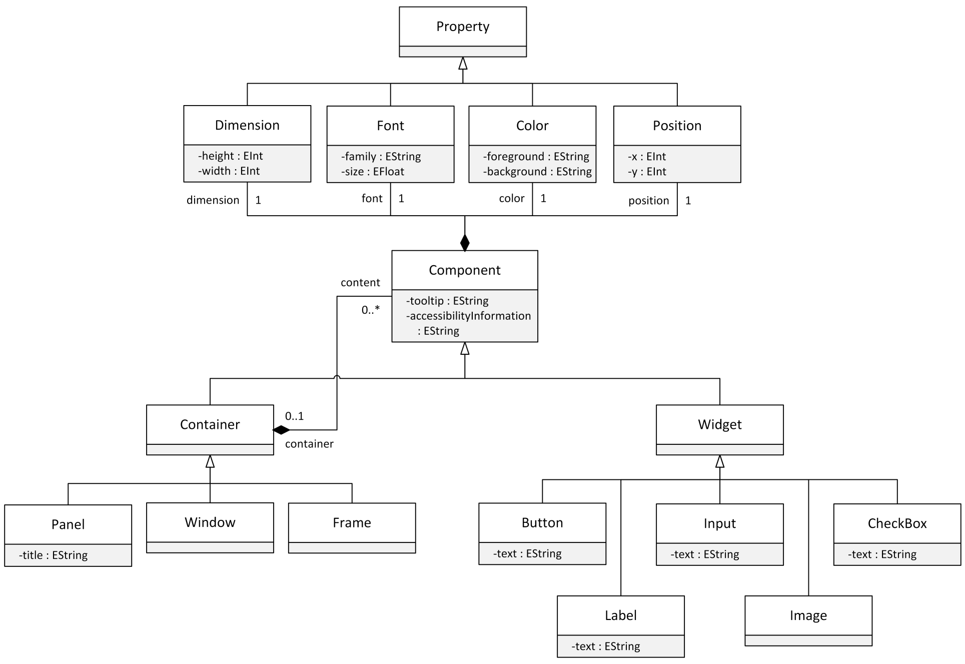

Figure 2: Excerpt of the generic GUI metamodel.

The metamodel is grounded on two main concepts: `Component` and `Property`. A `Component` can be a `Container` or a simple component that we call `Widget`. A `Container` can hold other components, thus allowing the usual hierarchical structure of a GUI. We consider the main types of container, namely `Panel`, `Window` and `Frame`; the basic widgets, including `Button`, `Label` and `Input`; and elemental properties such as `Dimension`, `Font`, `Color` or `Position`.

The usability rule that we take as an example, named *Label-input separation*, works with two components: the label and the input. It must analyze their positions to calculate the distance between them. In the generic GUI model presented, these components are represented as types of `Widget`. Both the `Label` and the `Input` have a set of properties associated, including a `Position` object storing the coordinates of the component in the GUI. However, there is no information whatsoever relating an `Input` with its `Label`. As we will see in the next section, this is something that must be inferred from the layout of the components.

## 3.2 Usability GUI metamodel

When evaluating the usability of a GUI, we apply rules that analyze specific properties of the GUI components, such as the color or the text attributes. Occasionally, we also evaluate only a certain type of component, since there are rules defined for specific components such as labels or menus. The hierarchical, component-centered structure of the metamodel presented in the previous section is not the most appropriate specially for the application of the first kind of rules, since it forces us to navigate through the components tree to evaluate a specific property.

In order to better support the actual behaviour of a usability analysis, we have defined an intermediate, property-oriented metamodel that we call usability GUI metamodel. An excerpt of this metamodel is shown in Figure 3. Its purpose is to ease the application of the usability rules, rearranging the usual component-centered structure of a GUI to give more importance to the concept of property, while at the same time maintaining the relevant information about the components' type. The usability GUI metamodel is thus grounded on two main concepts: `Property` and `Role`. We now have each property represented as an independent entity, and only the types of component relevant to the usability analysis remain as roles, dispensing with the rest of information in the GUI metamodel. We have, for instance, a role `Container` that allows us to maintain the hierarchical structure characteristic of a GUI, but no longer distinguish between simple widgets such as buttons or checkboxes, from which we only consider their properties. This way, we have a property-oriented representation of a GUI that still conserves the relevant information about the types of components, allowing us to easily apply usability rules that, for example, evaluate the characteristics of every text of the GUI, the position of the components in a particular container or the structure of a menu. Additional features have been defined for the usability GUI metamodel to further facilitate the analysis, such as the union of the properties `Dimension` and `Position` in a new `Box` property or the definition of a `Text` concept storing the content and characteristics of every piece of text in the GUI.

The *Label-input separation* usability rule previously introduced perfectly exemplifies how the introduction of this intermediate metamodel actually facilitates the usability analysis. As we mentioned in the preceding section,

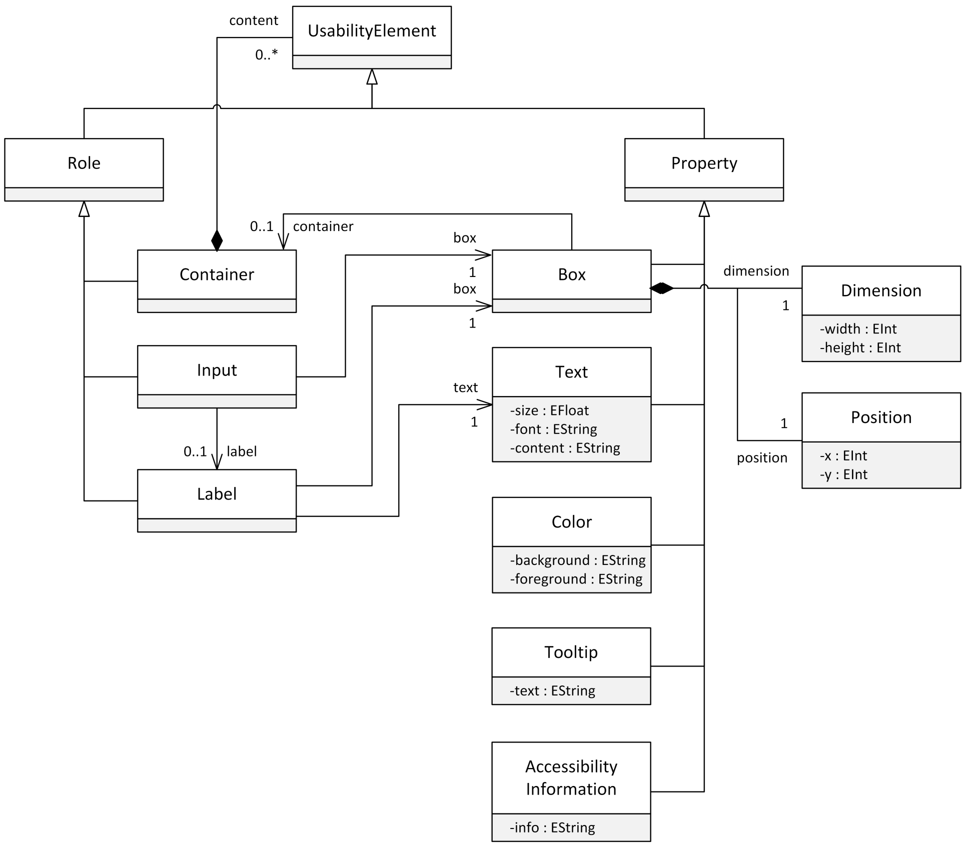

Figure 3: Excerpt of the usability GUI metamodel.

the relation between an input and its label is not something that we have available in the original GUI. Our generic GUI metamodel does not represent it either. This specific piece of information is important for a correct usability evaluation and so we consider it in the usability GUI metamodel. A position analysis process is required in order to infer the existing relations between inputs and labels in the GUI. Once these relations are inferred and stored in the usability GUI metamodel, the application of our usability rule becomes much simpler, as we have all the information needed on hand.

## 4 Usability rules DSL

We have defined a textual DSL to express usability rules, starting with the metamodel and then creating the textual notation from it using EMFText.

### 4.1 Usability rules metamodel

The catalogue of usability rules to be applied to the GUI is, along with the GUI itself, an input to the system, and as so it is a variable aspect of the architecture. Therefore, our usability rules metamodel must provide with enough flexibility to support the definition of a wide variety of usability rules. The abstraction of such a concept is a non-trivial endeavor, since usability rules can refer to many aspects of a GUI and produce very different types of results. We have defined a metamodel that considers the key features that a usability rule must have but leaves most of the behaviour of the rule in the hands of its implementator. The usability GUI metamodel is shown in Figure 4.

The root entity of the metamodel is `Catalogue`. A catalogue contains a collection of usability rules represented by the `Rule` entity. The metamodel considers basic properties for a rule such as its importance, description and the type of GUI element it is applied to. It also supports the definition of the behaviour of the rule through three blocks of code: i) *restriction*, to further restrict the elements that the rule is applied to, ii) *calculation*, to analyze the GUI element and obtain a representative value and iii) *correction*, to perform the correction of the usability defect. Finally, the metamodel introduces the concept of `Range`, of great importance in our conception of a usability rule. A range determines the set of values that are considered correct, that is, for a GUI element to be correct according to a usability rule, the value obtained by the calculation code must be in the range defined for that rule. There are different types of range to support sets of strings, boolean values or numerical ranges.

The definition of the different blocks of code, along with the concept of range, allow a very flexible, yet intuitive way to define usability rules. However, it also implies that, in order to define a rule, some knowledge of the EOL language is required to conform the different blocks of code. While

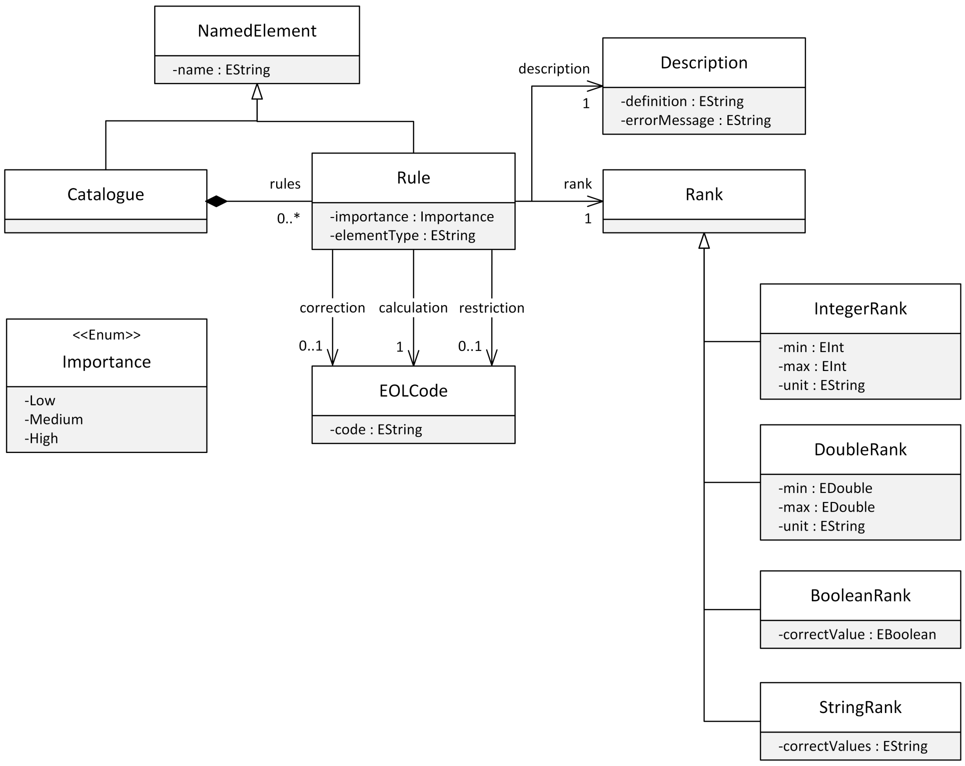

Figure 4: Usability rules metamodel.

some of the rules may not require an extensive knowledge of the language, this can not be the case for other more complex rules. Nevertheless, manual code writing is a necessary condition if we want to provide the user with enough flexibility to elicit his own custom-made catalogues of usability rules.

## 4.2 Usability rules textual DSL

We complete the usability rules DSL with a textual notation. This notation is constructed with EMFText from the metamodel previously shown in Figure 4, by defining the textual representation of each entity. EMFText generates both the editor and the injector, which allows us to easily create models containing catalogues of usability rules. The textual notation is very intuitive, as we use key words and common syntactic elements to represent the concepts, such as brackets for compounds entities or double quotation marks for strings.

We finish this section by showing the way we have defined the *Label-input separation* usability rule making use of our DSL. The complete specification of the rule is the following:

```
rule {
        name: "Label input separation"
        description {
                definition: "Checks the separation between an
                    input field and its corresponding label"
                errorMessage: "The label is too far from its
                    input field"
        }
        importance: Medium
        elementType: "Input"
        restriction: "return e.label.isDefined();"
        calculation: "return (e.box.position.x-(e.label.box.
            position.x+e.label.box.dimension.width)).max(e.box
            .position.y-(e.label.box.position.y+e.label.box.
            dimension.height));"
        range {
                type: integer
                min: 0
                max: 30
                unit: "px"
        }
        correction:     "
                var horizontal = e.box.position.x-(e.label.box
                    .position.x+e.label.box.dimension.width);
                var vertical = e.box.position.y-(e.label.box.
                    position.y+e.label.box.dimension.height);

                if (horizontal > vertical) {
                        e.label.box.position.x = e.label.box.
                            position.x + (horizontal - 30);
                        e.label.guiElement.position.x = e.
                            label.box.position.x;
                }
                else {
                        e.label.box.position.y = e.label.box.
                            position.y + (vertical -30);
```

```
                                        e.label.guiElement.position.y = e.
                                            label.box.position.y;
                              }
                      "
          }
```

The rule is applied to all the elements of type `Input`. We limit the application of the usability rule to those input elements that have a label defined, since this may not always be the case (orphan inputs constitute another usability error that we consider in our catalogue). The calculation code obtains the minimum distance between the input and its label, considering that a label can be on top or left of its input. The correction code modifies the position of the label so that the separation is smaller than a threshold value. This value is established in the range, which is an integer range that restricts the label-input distance to 30px. We can see how easily the concepts defined in the metamodel adapt to the structure and behaviour of the usability rule, and how clearly the defined textual notation can describe the whole specification.

## 5 Analysis and correction

In this section we introduce the usability analysis and correction processes, implemented as model to model transformations. We begin by explaining the target metamodel of these transformations: the defects metamodel, and continue with a complete study of how the analysis and correction of these defects are done.

### 5.1 Defects metamodel

We have created a metamodel to represent the usability defects found in a GUI. The result of the analysis and correction processes is a defects model conforming this metamodel, shown in Figure 5. The eventual purpose of the defects model is to be presented to the user in a readable format by a model to text transformation. Therefore, we define a very simple metamodel that considers the most important and representative attributes of a usability defect, represented mainly in a string format. Thus, each usability defect stores the name of the usability rule and the GUI component it is related to, for example, instead of the model elements themselves.

The root element of the metamodel is the `Evaluation` concept, that contains the set of usability defects found during the evaluation of a particular GUI. The list of attributes of a defect includes the aforementioned names of the corresponding GUI component and usability rule, information about the value obtained from the application of the rule, the correct range of values for such rule and other attributes such as the description of the defect or its importance. It also includes a `corrected` boolean attribute to be used dur-

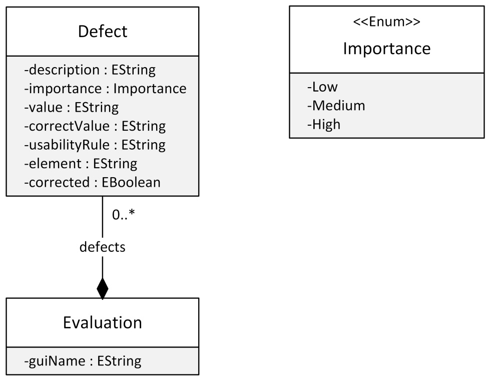

Figure 5: Usability defects metamodel.

ing the correction process to distinguish between corrected and not corrected defects.

This set of attributes is representative enough to perfectly understand the defects found in the GUI, allowing the user to easily locate and, if wished, manually correct the defects himself. In the next sections we will see how these defects are constructed and in Section 7 an example is shown of how they are presented to the user.

## 5.2 Analysis

In the analysis process we must take the GUI and the usability rules catalogue and generate a set of usability defects resulting from the application of the usability rules to the GUI elements. It is common to use a model to model transformation to implement a defect extraction process. This kind of transformation defines rules that are only applied to defective elements, that is, the detection of the defect occurs in the guard of the rule and the defect is created in its body, so that only for those elements who are considered defective the rule is applied and therefore the defect created. We adopt this approach in order to implement our usability analysis model to model transformation. Considering the concepts managed in the usability rules metamodel introduced in Section 4.1, our transformation rules have the following structure:

```
operation restriction()
operation calculation()

rule Example
    transform e : elementType
    to defect : Defect {
        guard : restriction() and not (calculation() inside range)

        << defect creation >>
}
```

The rule transforms a GUI element of some specific type into a usability defect. It is executed only if the restriction defined in the usability rule is fulfilled and the value obtained from the calculation code is not inside the range of correct values, i.e. the GUI element is defective. The usability GUI model is the input to the model to model transformation, but most of the information needed to compound the transformation rules comes from the usability rules model. This is a peculiar situation, since we can not define the analysis as a model to model transformation that takes as input both the GUI and the usability rules models. Instead, the very structure of the transformation depends on the usability rules metamodel. In consequence, it is necessary to implement an additional step: a model to text transformation that generates the model to model transformation from the usability rules model. This approach is shown in Figure 6.

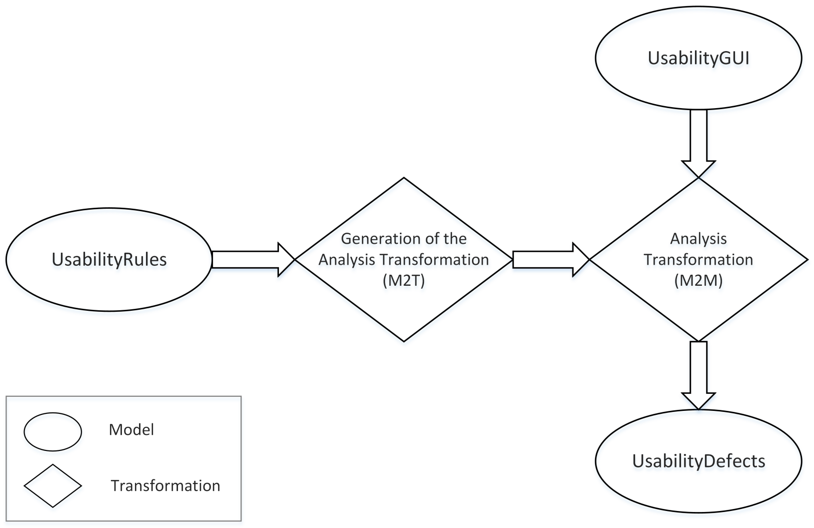
UsabilityGUI
UsabilityRules
Generation of the Analysis Transformation (M2T)
Analysis Transformation (M2M)
UsabilityDefects
Model
Transformation

Figure 6: Structure of the usability analysis process.

The model to model transformation in charge of the usability analysis is therefore a simple transformation that takes the GUI model as input and generates a defects model as output. This transformation is in turn generated by a model to text transformation that creates a transformation rule for each usability rule in the catalogue.

If we take a look again at the structure of the transformation rules to be generated, we can see how easily this can be done through a model to text transformation, since all the content can be dynamically generated from the information available in the usability rules metamodel (see Figure 7). The restriction and calculation code blocks are emitted directly, and we must only take into account the different types of ranges to implement the corresponding check.

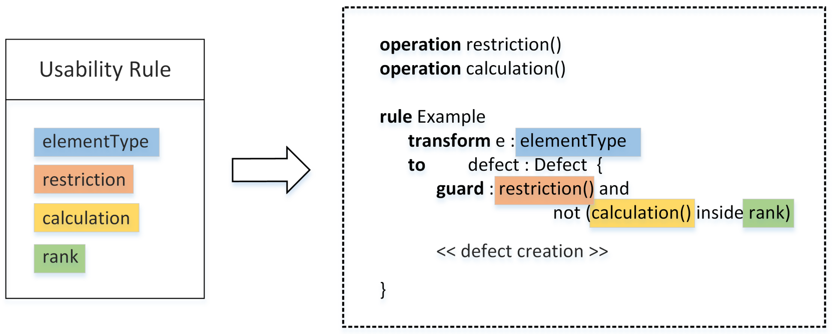

Figure 7: Creation of a m2m transformation rule from the corresponding usability rule.

In order to illustrate this process, we list below the transformation rule generated from our *Label-input separation* usability rule.

```
operation Label_input_separation_restriction(e : UsabilityGUI!Input) {
    return e.label.isDefined();
}

operation Label_input_separation_calculation(e : UsabilityGUI!Input) :
     Integer {
    return (e.box.position.x-(e.label.box.position.x+e.label.box.
        dimension.width))
       .max(e.box.position.y-(e.label.box.position.y+e.label.box.
           dimension.height));
}

@greedy
rule Label_input_separation
    transform e : UsabilityGUI!Input
        to defect : AnalysisDefects!Defect {

        guard : Label_input_separation_restriction(e) and
                  not(Label_input_separation_calculation(e) >= 0
                    and Label_input_separation_calculation(e) <= 30)

            defect.description = "The label is too far from its input
                field";
            defect.importance = AnalysisDefects!Importance#Medium;
            defect.usabilityRule = "Label input separation";
            defect.value = Label_input_separation_calculation(e).
                asString() + " px";
            defect.correctValue = "0-30 px";
            defect.element = e.guiElement.name;

            evaluation.defects.add(defect);
    }
```

The usability rule is translated into a transformation rule that generates a `Defect` from an `Input` element. The restriction and calculation operations are defined from the code available in the usability rules model. The guard of the rule makes use of these operations to assure that the transformation rule is only applied to those `Input` elements that satisfy the restriction condition and whose calculated value is out of the defined range. We can see how easily this is checked when we work with an integer range. The creation of the defect is also quite straightforward. Most of the information is extracted directly from the usability rules model. To set the `value` attribute, we invoke the calculation method once again and append the string describing the measure unit. The body of the transformation rule finishes by adding the created defect to the `Evalutaion` root element of the defects model. This process is repeated for every usability rule defined in the catalogue, conforming the model to model transformation that performs the usability analysis.

## 5.3 Correction

The correction process resembles the analysis, since it also performs a usability evaluation before the correction. Consequently, the same approach is taken and the generated transformation rules share a similar structure:

```
operation restriction()
operation calculation()
operation correction()

rule Example
    transform e : elementType
    to defect : Defect {
            guard : restriction() and not (calculation() inside range)

            << defect creation >>

            correction()
}
```

A new operation is defined: the one in charge of the correction, that is invoked at the end of the rule body. In this operation the input GUI element is properly modified to solve the usability defect. Once again, the code is obtained from the usability rules model and directly emitted during the model to text transformation. However, we must remember that the definition of a correction method is optional in a usability rule. Therefore, we do not necessarily have this operation in all the transformation rules. Finally, we also let the user take some control over the process by deciding whether the correction is automatically executed for all the defects found or, on the contrary, it is the user who decides which types of defects are corrected and which are not.

The model to model transformation generated for the correction process is quite similar to the one responsible of the analysis. Consequently, the transformation rule obtained from our *Label-input separation* usability rule barely differs from the one that we presented in the previous section. It is therefore unnecessary to show it here. It suffices to say that a new correction operation is defined on top of the transformation rule, and the invocation of this method is appended at the end of the rule body.

## 6 Android integration

The input of the analysis and correction processes, as depicted in Figure 1, is a model conforming to our GUI metamodel. Therefore, in order to integrate a specific interface technology into our architecture, we need to be able to provide such a model from the original interface. This is achieved through the definition of the following elements:

- A technology-specific metamodel.
- An injection process to obtain the technology-specific model from the GUI in its original format.
- A model to model transformation to obtain the GUI model from the technology-specific model.

Figure 8 shows the integration process for the particular case of Android interfaces.

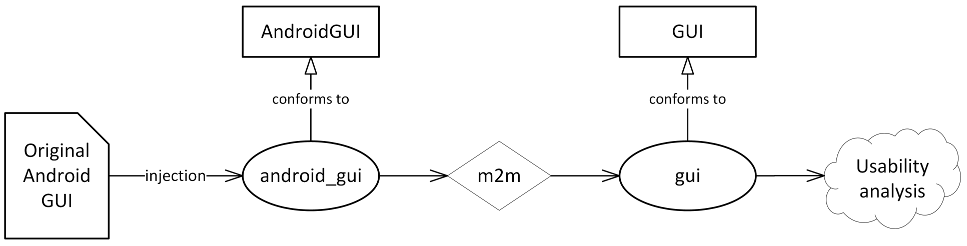

Figure 8: Integration process for Android interfaces.

We must first decide which artifact is the starting point of the process. This is not straightforward in our case. Android interfaces can be created instantiating `View` and `ViewGroup` objects from the Android API, but the easiest and most frequently used way is to define the layout with an XML file. When one of these layout resources is loaded in the app, Android initializes each node of the layout into a runtime object that the developer can use to define additional behaviour, query the object state, or modify the layout [1]. Accordingly, we can use this XML file as the input of the integration process, injecting the model from the XML tags defining the layout. The problem with this approach is that the definition of an Android interface in XML strongly relies on default and calculated values, that is, the elements and attributes defined by the developer in the XML are only a (sometimes small) part of the total attributes that each component has once it is instantiated as a Java object. Many attributes are ignored in the XML definition only to be later instantiated with default values, and other attributes such as the position are usually specified in a relative manner, and not calculated until the layout is instantiated in a particular Android device with a fixed screen size and resolution. Important information about the interface is therefore difficult or impossible to obtain from the XML definition, limiting the applicability of this approach.

The alternative that can be deduced from this is to take the API objects themselves as the starting point of the injection process. In this way, we make sure that we have all the information about the GUI elements available, since the instantiation has already been done and all the values have been assigned to the corresponding attributes. Hence, an Android GUI metamodel must be defined, and models conforming to this metamodel need to be injected from the Android Java API objects representing the GUI. Fortunately, Api2Mol [8] automates both of these processes.

Api2Mol is a tool that automates the implementation of API-MDE bridges for supporting both the creation of models from API objects and the generation of such API objects from models. A declarative rule-based language is provided to easily write mapping definitions to link API specifications and the metamodel that represents them. Furthermore, Api2Mol also allows both the metamodel and the mapping to be automatically obtained from the API specification. Both the discovery and injection processes are depicted in Figure 9.

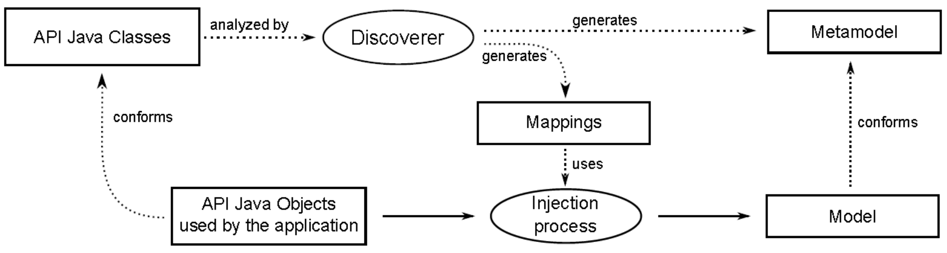

Figure 9: Discovery and injection processes in Api2Mol.

We launched the Api2Mol discovery process with Android 4.0 (API Level 14) to obtain a metamodel of the API and a first version of the mapping definition. However, the automatically discovered mapping definition rely on get and set methods named following Java conventions, and, unfortunately, many methods in the Android API do not follow such conventions. Consequently, it was necessary to manually specify the methods that allow us to get and set the attributes that we are interested in, so that during the injection process the values of the attributes in the Java object can be correctly recovered and set in the corresponding class of the model. We only set the mapping definitions of the classes in the Android API that are relevant for our purposes. Therefore, even if the metamodel discovered by Api2Mol considers all the classes in the API, for all practical purposes we only work with a subset of them. Figure 10 shows an excerpt of this simplified metamodel with some of the classes and attributes that are used during the injection process.

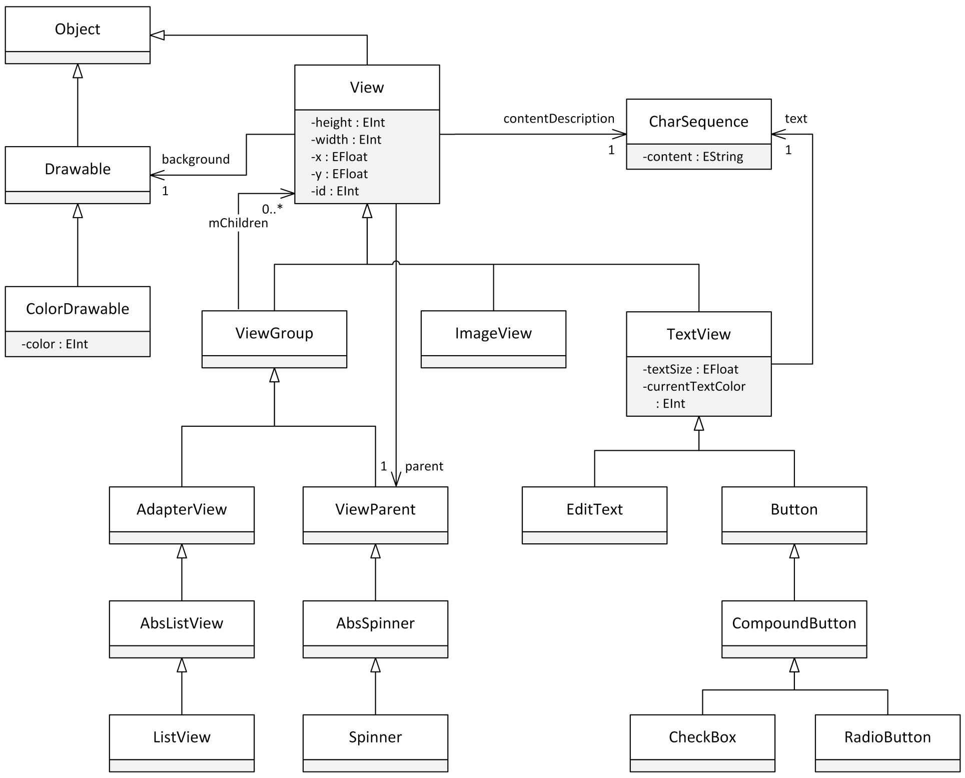

Figure 10: An excerpt of the simplified Android metamodel used during the injection process.

With the automatically discovered Android metamodel and the adapted mapping definition Api2Mol can run its injection process, taking a tree of Java objects as input and generating the corresponding model conforming the Android metamodel as output. Once we have the model representing the Android GUI, we can launch the usability evaluation tool with the model as input. A model to model transformation has been defined to obtain the generic GUI model from the Android model, and the rest of the process is executed as explained in previous sections, generating a HTML report with the usability analysis results.

The correction process is currently not supported for Android interfaces. The inaccuracies in the automatically discovered mapping definition make it extremely difficult to define the correct mapping between all the attributes of Java objects in the Android API and their corresponding classes in the Android metamodel. Consequently, a lot of information is lost when we inject the model. Although this is not critical for performing the usability analysis, in order to generate back the correct Java objects we need all the attributes with their original values, and the complete definition of such a mapping is a highly arduous task. Moreover, the changes made on the Java objects would not be reflected in the XML definition of the GUI, so even if the correction is possible, it could only be temporary.

## 7 Validation

In order to achieve a basic validation of the developed tool, we have designed a very simple case study in which we analyze the usability of an Android GUI. This validation process is only meant to be a proof of concept of the tool, i.e. an example of its potential and applicability. A set of usability rules has been defined that addresses the fundamental properties of GUI elements, such as their position or size, and checks for elemental, well-known usability issues. Table 1 summarizes the characteristics of these rules.

On the other hand, an ad hoc example of an Android GUI has been built, in which we deliberately insert different usability defects to test in a simple, agile manner the efficacy of the tool. Figure 11 depicts the Android GUI used as the input of our usability analysis process. It consists of a simple form, with different input fields and a submit button. The GUI contains different usability defects, most of which are visible to the naked eye. Figure 12 shows the GUI with all the usability defects already indicated. Additionally, some GUI elements have their accessibility information set and some do not, a feature that can not be perceived.

As we can see, although the GUI does not initially seem too defective, it actually contains a fair bit of usability errors that hinder the correct understanding and use of the GUI by the user. Since the defects have been intentionally injected in the GUI, it is easy to determine what the output of the usability analysis process should be. For the validation to be completed successfully, at least all the usability defects identified in Figure 12 (plus the accessibility related ones) must be reported, in a way that clearly specifies what the problem is and where it is located.

Following the steps detailed in Section 6, we first run the injection process from our Android application. As a result, a model representing our Android GUI is obtained. Then, we use this model and the defined usability rules catalogue as the input of our usability analysis tool. The tool handles automatically the rest of the process and generates a HTML report with the results. Table 2 shows the usability defects reported by the tool.

The report shows every usability defect detected in the GUI, listing for each the properties defined in our defects metamodel (see Figure 5). Access to the name of an Android GUI element was not trivial from a programmatic point of view. Instead, we use the ID of the element, which, unfortunately, may not be as representative. However, the set of properties shown is complete enough to allow the correct identification and understanding of each usability defect. This is particularly relevant for Android GUIs. Since our

| **Name** | **Definition** |
|---|---|
| Font size diversity | Check that the number of different font sizes used is not too high |
| Foreground color palette | Check if the foreground color of an element is contained in a defined color palette |
| Background color palette | Check if the background color of an element is contained in a defined color palette |
| Contrast | Check the contrast between an element's foreground and background color |
| Separation | Check the separation between an element and the rest of elements inside its container |
| Font types | Check the number of different font types being used |
| Label-Input separation | Check the separation between an input field and its corresponding label |
| Label text length | Check that the text in a label is not too long to be easily read |
| Font size | Check that the font size is big enough so that the text can be easily read |
| Vertical alignment | Check if an element is vertically aligned with the rest of elements in its container |
| Horizontal alignment | Check if an element is horizontally aligned with the rest of elements in its container |
| Number of elements | Check the total number of elements in a container |
| Input labelling | Check if an input field is associated to a label or not |
| Accessibility information | Check if the accessibility information has been set for an element |
| Overlapping | Check if an element overlaps other elements in its container |

Table 1: Set of usability rules defined for testing.

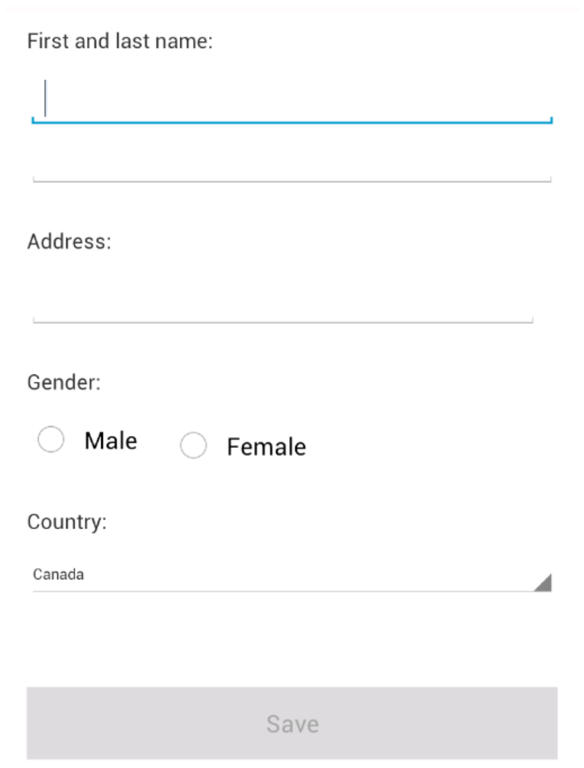

Figure 11: Android GUI designed for the validation process.

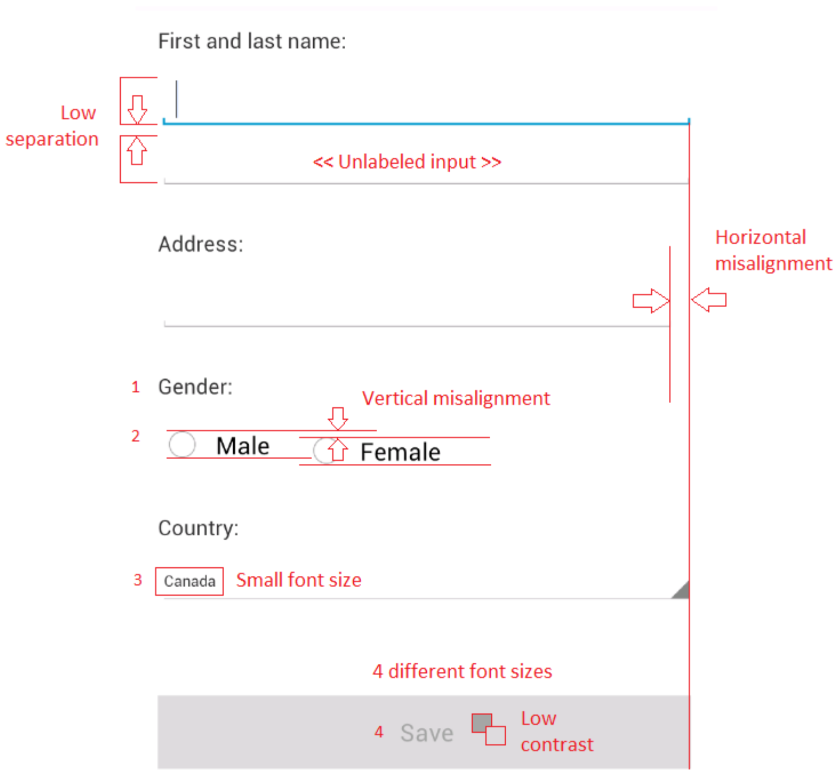

Figure 12: Android GUI designed for the validation process. The usability defects have been highlighted.

| # | Usability rule | Description | Importance | Element | Correct value | Current value |
|---|---|---|---|---|---|---|
| 1 | Contrast | Low contrast between foreground and background colors | High | Button-2131361803 | 4.5-* | 1.7 |
| 2 | Separation | The element is too close to other elements in its container | High | EditText-2131361794 | “” | EditText-2131361796 |
| 3 | Separation | The element is too close to other elements in its container | High | EditText-2131361796 | “” | EditText-2131361794 |
| 4 | Font size | Font size is too small | High | Spinner-2131361795 | 24.0-* | 20.0 |
| 5 | Vertical alignment | The element is not vertically aligned with other elements in its container | High | RadioButton-2131361800 | “” | RadioButton-2131361801 |
| 6 | Vertical alignment | The element is not vertically aligned with other elements in its container | High | RadioButton-2131361801 | “” | RadioButton-2131361800 |
| 7 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | EditText-2131361793 | “” | EditText-2131361798 |
| 8 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | EditText-2131361794 | “” | EditText-2131361798 |
| 9 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | EditText-2131361796 | “” | EditText-2131361798 |
| 10 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | TextView-2131361797 | “” | EditText-2131361798 |
| 11 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | TextView-2131361799 | “” | EditText-2131361798 |
| 12 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | TextView-2131361802 | “” | EditText-2131361798 |
| 13 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | Spinner-2131361795 | “” | EditText-2131361798 |
| 14 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | Button-2131361803 | “” | EditText-2131361798 |
| 15 | Horizontal alignment | The element is not horizontally aligned with other elements in its container | High | EditText-2131361798 | “” | TextView-2131361793,EditText-2131361794,EditText-2131361796,TextView-2131361797,TextView-2131361799,TextView-2131361802,Spinner-2131361795,Button-2131361803 |
| 16 | Labeled input | This input does not have any label associated | High | EditText-2131361796 | true | false |
| 17 | Font sizes | Too many distinct font sizes | Medium | Android GUI | 1-2 font sizes | 4 font sizes |
| 18 | Accessibility information | Accessibility information has not been set | Medium | EditText-2131361794 | true | false |
| 19 | Accessibility information | Accessibility information has not been set | Medium | EditText-2131361796 | true | false |

Table 2: Results of the usability analysis process.

current implementation does not support the automatic correction of the defects in these interfaces, it is important to provide a complete enough report so that the user can easily identify and manually correct the defects.

It is worth noting that some usability defective situations are reported through multiple defects. For example, there is an horizontal misalignment between one of the input fields and other elements in the GUI (see Figure 12). In Table 2, usability defect #15 correctly identifies this situation, by reporting such input field as misaligned regarding to 8 other elements. However, each of these 8 elements are also detected as being misaligned regarding to that input field, resulting in a total of 9 usability defects that indicate the same defective situation. It is important to take this behaviour into account to correctly interpret the report.

A total of 19 usability defects were reported, including all the defects indicated in Figure 12, as well as two additional defects related to the accessibility information which were not noticeable to the naked eye. This means that all the usability defects that we introduced in the GUI on purpose were correctly identified. This proof of concept demonstrates that the selected set of usability rules is able to identify basic usability defects in Android GUIs. We have performed a basic validation of the utility of our tool for the usability analysis of interfaces. However, a more complete validation process is necessary in order to better assess the efficacy and overall usefulness of our approach.

## 8 Related work

Usability has been a very prominent field in software engineering for the last few years, both at the theoretical and industrial level. A great number of usability evaluation techniques have been proposed, and many of them are regularly applied in plenty of software projects. Consequently, there has been an interest in the automation of this process, and there is a wide variety of available tools and proposals regarding this topic.

The most important reference when it comes to automatic usability evaluation is the study of the state of the art performed by Ivory et al. in 2001 [7]. In this paper, a taxonomy is proposed to classify usability evaluation tools, placing particular emphasis on the concept of automation. Ivory categorizes tools according to the following aspects:

- Method class. E.g. testing, inspection, analytical modeling or simulation.
- Method type. E.g. log file analysis, guideline review or GOMS analysis.
- Automation type. E.g. none, capture, analysis or critique.

- Effort level. E.g. minimal effort, model development, informal use or formal use.

This is the most extended taxonomy for the categorization of automated usability evaluation methods, and the majority of the techniques can be classified according to it.

In [11], for example, the authors propose the use of task models and logs generated from user tests to perform an automatic usability analysis. In this case, we have a method that combines analytical modeling with testing, and performs an evaluation that requires informal use of the interface. Au et al. propose HUIA [2], a usability testing framework for handheld device applications. It is a complete tool that performs both an inspection and a testing method on the interface, capturing the activity from the user and later doing an heuristic analysis of the results, as well as of the static GUI itself. It requires a formal use to know the desired behaviour, that is later compared to the results of the informal use of the application. In [12] another testing method is proposed for Android applications. The tool logs the activity of the users and later calculate some metrics from the obtained data. It is therefore a log analysis method that requires an informal use and in which both the capture and the analysis of the data are automated.

According to the introduced taxonomy by Ivory et al., our tool can be defined as an inspection method that applies an automated guideline review to an interface, requires no effort from the user and provides both a usability analysis and also critique in the form of correction.

However, Ivory's taxonomy does not take into account some important aspects for the evaluation of this kind of tools. For example, one of the characteristics of our tool is that the usability analysis is performed on existing interfaces through a reverse engineering process. This kind of procedure can also be observed in multiple tools such as the one proposed by Campos et al. [4], where behaviour models are derived from the source code of a given user interface. Nonetheless, usability evaluation tools can operate in other ways. It is common to find, for instance, tools that are not applicable to any given interface, but are integrated into a specific usability construction environment. These are methods to be used during the design process, instead of on the existing, finished interface. In [6], for example, a usability evaluation tool is proposed that combines MASP, a model-based runtime environment to offer multimodal user interfaces, and MeMo, a workbench supporting an automated usability analysis.

One important characteristic that is not considered in Ivory's taxonomy is technology independence. Our tool has been designed so that it can be adapted and applied to any kind of interface. Other tools, like the one proposed in [20], also achieve technology independence, in this case by directly analyzing captures of the user interface, focusing only on aesthetic aspects that can be measured regardless of the underlying technology. However, it is most common for tools to be defined to work with a specific GUI definition technology. Web interfaces, for instance, have got a lot of attention in the last few years. A study of the state of the art on web usability performed by Fernández et al. [5], for example, showed that as much as 31% of the analyzed tools proposed some kind of automation.

Another feature that is often left out is the capacity of correcting the usability defects that have been detected. Ivory considers in his taxonomy user assistance in the form of a critique destined to help correcting the defects, but some tools, like ours, go one step further allowing the automation of the correction.

Additionally, tools that assess usability with a guideline review can or can not allow the definition by the user of the usability rules that are applied to the interface. We believe that this flexibility is another important characteristic to keep in mind, since it favors reusability and extensibility.

DESTINE [18, 9] is a tool for the usability analysis of web interfaces that also brings together these two last characteristics. It supports the definition of usability rules by the user, plus allows the automated correction of the defects. In order to do this, it provides a language called GDL for the definition of guidelines, thus separating the evaluation logic from the engine. In this respect, DESTINE is probably the tool that resembles the most our architecture. However, it is specifically designed for web interfaces and therefore not applicable to other interface technologies.

In general, there is a wide variety of available tools for an automated usability analysis, spanning from methods such as testing or task analysis, from which we have mentioned some examples, to guideline reviews, the method used by our tool. Nevertheless, we find that the majority of these tools lack some features that, despite not being considered often enough, are in our opinion of great interest and importance. Among them, the three that characterize our architecture stand out: technology independence, automated correction and flexibility in the definition of usability rules. By virtue of them, we consider that the proposed tool is a solution of great interest in the field of usability evaluation.

## 9 Conclusions and future work

We have presented an approach for the automatic evaluation of the usability of a GUI. The approach is completely model-based, e.g. the architecture relies solely on models, metamodels and model transformations. Usability analysis and correction are defined as two distinct, yet similar processes. A GUI model, obtained from the original interface, and a catalogue of usability rules, defined using a DSL, are required as the input of the method. We use an intermediate GUI model and define the analysis and correction processes as model to model transformations. As a result, a model is obtained with the detected usability defects, which are reported to the user through a final model to text transformation. For the correction, the input model is also modified during the process.

The main characteristics of our proposal are: i) technology independence, ii) support for custom usability rules and iii) automatic correction. These features provide a great flexibility in the application of the method and differentiate it from most usability analysis techniques in the literature. At the same time, they offer multiple extension possibilities. In this article, we have described the integration of our method with the Android GUI technology. However, our approach can be used to analyze and correct the usability of any other type of GUI. In this sense, an interesting line of future work is to integrate additional GUI technologies, such as web-based interfaces. In the same way, a basic catalogue of usability rules was defined in the context of our validation case. In the future, this catalogue will be extended and/or alternative, more complete catalogues will be defined. The creation of these catalogues of usability rules will help us to: i) test the expressiveness and adequacy of our DSL for the definition of usability rules and ii) define a more extensive and formal case study to validate the approach. Following this last line of thought, it will also be important to apply our method for the GUI usability analysis and correction of real-world systems.

## References

[1] Android. Android ui overview, accessed December 29, 2015.

[2] Fiora T. W. Au, Simon Baker, Ian Warren, and Gillian Dobbie. Automated usability testing framework. In *Proceedings of the Ninth Conference on Australasian User Interface - Volume 76*, AUIC '08, pages 55–64, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[3] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 2012.

[4] José Creissac Campos, João Saraiva, Carlos Silva, and João Carlos Silva. GUIsurfer: A Reverse Engineering Framework for User Interface Software. *Reverse Engineering - Recent Advances and Applications*, 2012.

[5] Adrian Fernandeza, Emilio Insfrana, and Silvia Abrahãoa. Usability Evaluation Methods for the Web: A Systematic Mapping Study.

[6] Sebastian Feuerstack, Marco Blumendorf, Maximilian Kern, Michael Kruppa, Michael Quade, Mathias Runge, and Sahin Albayrak. Automated Usability Evaluation during Model-Based Interactive System Development.

[7] Melody Y. Ivory and Marti A. Hearst. The State of the Art in Automating Usability Evaluation of User Interfaces. *ACM Computing Surveys*, 33(4), 2001.

[8] Javier Luis Cánovas Izquierdo, Frédéric Jouault, Jordi Cabot, and Jesús García Molina. Api2mol: Automating the building of bridges between {APIs} and model-driven engineering. *Information and Software Technology*, 54(3):257 – 273, 2012.

[9] Arnaud Jasselette, Marc Keita, Monique Noirhomme-Fraiture, Frédéric Randolet, Jean Vanderdonckt, Christian Van Brussel, and Donatien Grolaux. Automated Repair Tool for Usability and Accessibility of Web Sites.

[10] R. Jeffries, J.R. Miller, C. Wharton, and K.M. Uyeda. User interface evaluation in the real world: A comparison of four techniques. *Proceedings of the Conference on Human Factors in Computing Systems (New Orleans, LA, April)*, pages 119–124, 1991.

[11] Andreas Lecerof and Fabio Paternò. Automatic Support for Usability Evaluation. *IEEE Transactions on Software Engineering*, 21(10):863–868, 1998.

[12] Florian Lettner and Clemens Holzmann. Automated and unsupervised user interaction logging as basis for usability evaluation of mobile applications. In *Proceedings of the 10th International Conference on Advances in Mobile Computing &#38; Multimedia*, MoMM '12, pages 118–127, New York, NY, USA, 2012. ACM.

[13] R. Molich, N. Bevan, S. Butler, I. Curson, E. Kindlund, J. Kirakowski, and D. Miller. Comparative evaluation of usability tests. *Proceedings of the UPA Conference (Washington, DC, June)*, pages 83–86, 1998.

[14] R. Molich, A.D. Thomsen, B. Karyukina, L. Schmidt, M. Ede, W. Van Oel, and M. Arcuri. Comparative evaluation of usability tests. *Proceedings of the Conference on Human Factors in Computing Systems (Pittsburgh, PA, May)*, pages 83–86, 1999.

[15] Jesús J.García Molina et al., editors. *Desarrollo de Software Dirigido por Modelos: Conceptos, Técnicas y Herramientas.* RA-MA, 2013.

[16] J. Nielsen. *Usability engineering.* Academic Press, 1993.

[17] Bran Selic. What Will it Take? A View on Adoption of Model-Based Methods in Practice. *Software and Systems Modeling*, 11(4):513–526, 2012.

[18] Jean Vanderdonckt, Abdo Beirekdar, and Monique Noirhomme-Fraiture. Automated Evaluation of Web Usability and Accessibility by Guideline Review.

[19] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, 2014.

[20] Mathieu Zen. Metric-Based Evaluation of Graphical User Interfaces: Model, Method, and Software Support.