Key words:
Semantic Networks, Graphical User Interface

Witold WYSOTA*

SEMANTIC MODEL OF APPLICATION USER INTERFACES

The paper introduces a concept of modeling graphical user interfaces (GUIs) in applications using semantic networks. It presents basic terms used to describe a user interface, shows their representation in a form of an ontology and relations between the terms. It also explains the benefits of having a semantic model of a user interface and gives a simple example how this model can be used in practice to generate implementation of the user interface for an arbitrary platform. The concept presented is a work in progress.

1. INTRODUCTION

Since the beginning of the last decade of the previous century we have witnessed a rapid growth of the number of applications with graphical user interfaces. The popularity of MacOS, AmigaOS and Windows operating systems that implemented the idea of graphical desktops and windows containing smaller graphical elements has catalyzed creation of new range of applications. They were more user friendly (hence easier to learn), allowed better interaction between different programs (i.e. using the drag and drop mechanism), looked better and were more fun to use than earlier more “scientific” command line or text-based applications.

The unix world slowly started to catch up by introducing new layers of complexity and toolkits to the X Window system created in 1980’s in MIT and making it open source in mid 90’s[3]. Still the platform was merely an addition to using command line utilities that had greater capabilities but also required more expertise and experience.

* w.wysota@ii.pw.edu.pl, Institute of Computer Science, Warsaw University of Technology

Then came advanced toolkits and desktops based on them such as Motif and CDE[1], GTK+ and Gnome or Qt and KDE[2][5,6]. Through the years they made significant progress to become what we see on our screens today.

Nowadays we have multiscreen 32bit deep desktops that look very pretty and colourful. If application developers want their applications to become popular they have to make them pretty as well to fit right into the candy-like environment or else the potential user will probably find a dozen other programs with similar functionality but a more pleasant look.

But a good looking frontend will not guarantee success. The user interface has to not only look good but also be easy to maintain, modify and configure. Finally, it has to do its job. And with more and more complex functionality of modern applications the user interface has to be complex as well. To be able to comprehend and implement the user interface, application developers need means to describe graphical objects, their relations and functionality.

There are different ways and notations to model graphical interfaces. This chapter presents a semantic approach to the problem by defining an ontology for graphical user interfaces. The following section of the chapter outlines existing ways of modelling user interfaces, their strengths and weaknesses. In the next part a solution being created for representing graphical user interfaces in applications based on semantic web is shown. It describes the potential advantages and use cases of an ontological description. The final section contains current conclusions and points research that still needs to be done in the area.

2. REPRESENTING USER INTERFACES

There are two kinds of domains in representing entities and user interfaces in particular. Firstly one can have a simple notation that serves only as a container for user interface data. This approach is widely used by the software development industry. It is simple and easily supported by many tools available.

The other way is to create a formal model of the environment described which gives additional abilities such as ability to simulate different scenarios and performing logical analysis of the data. This approach is used mostly in academic and scientific area due to complexity, overhead, and scalability problems that often arise when dealing with larger models.

---

[1] *Common Desktop Environment*

[2] *K Desktop Environment*

In this section of the chapter some of the existing methods from both domains shall be briefly presented.

2.1. FORMAL METHODS

There are many existing formal methods for representing computer systems but they are mostly general purpose approaches and almost none of them are tailored to specifically represent user interfaces.

As for the general methods – if we perceive a user interface as an autonomous system, we can model it as a finite state machine. We can assume that each of the elements of the interface is always in one of a set of defined states. A cartesian product of all elements yields all possible states of the system (although some or even most of them might not be reachable). This allows to use any of existing FSM techniques to model and simulate the system. This aspect is out of scope of this work and thus will not be described in detail.

As already mentioned there are not many formal methods dedicated to modelling user interfaces but it is worth mentioning those that are available.

At the beginning of the last decade of the 20th century there was research conducted by a group led by Stephen W.L. Yip that was related to developing a formal method to describe GUIs. They developed WinSTD and WinSpec that modelled both transitions of appearance and the behaviour of graphical elements through a combined method of state diagrams and a dedicated language similar to the Z-notation[8,9]. Unfortunately it seems the methodology has stopped being developed and never came into wider use.

An interesting concept is presented in [4]. The author introduces two languages – RML and TaskMODL that allow to create a formal model of the tasks performed by the application user interface. However, the concept is more related to the application logic than to the user interface itself. Its role regarding the user interfaces is mainly that it facilitates the design stage of the user interface but in the end GUI needs to be created manually.

2.2. MODERN WAYS OF DESCRIBING UI

Software development industry is thus far rarely focused on the user interface itself although the situation is starting to change, especially since the introduction of next generation touchscreen equipped mobile phones and PDA devices. Nevertheless the industry is interested mostly in a simple representation of the GUI that only allows to produce source code that implements the user interface.

Most solutions from this area simply stream properties of subsequent graphical objects into the target file in a form of binary or textual (most often XML) tags. Those serialized objects can form a hierarchy of objects in a form of a tree structure or the structure can be flat and relations of the objects are resolved after deserializing the description by the interpreter (i.e. based on the geometry or types of the objects).

It should be mentioned that such representations focus on single units (classes, windows, widgets) and do not give a perspective of the whole system. Therefore it is hard to treat those notations as anything more than a data payload for the UI generation mechanism. This of course has also its benefits, for instance it allows for loose coupling of objects thus increasing reuse of modules.

It is also the preferred choice for the industry. This is because of ease of handling and forced tendency to focus only on the part of the system that is of interest to the operator. Otherwise a situation might happen when he would have to wait until the perspective of the whole system loads and clutters the screen just to correct a spelling error in one of the windows of the application under development.

An example of such notation is an XML based representation used by a popular cross-platform toolkit Qt[2]. It uses a hierarchy of tags to describe objects of the graphical interface and their settings. This abstract description is then parsed by a tool called UIC[3] that generates C++ code which creates and sets up the GUI in the final application. A sample description of the interface as used by Qt is presented below (irrelevant parts have been cut out).

```
<widget class="QWidget" name="Window">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>314</width>
        <height>110</height>
      </rect>
    </property>
    <property name="windowTitle">
        <string>Form</string>
    </property>
    <widget class="QPushButton" name="Button">
        <property name="geometry">
            ...
        </property>
```

[3] *User Interface Compiler*

```
        <property name="text">
            <string>Button</string>
        </property>
    </widget>
    <widget class="QLineEdit" name="LineEdit">
        <property name="geometry">
            ...
        </property>
    </widget>
</widget>
```

3. SEMANTIC MODEL

The main purpose of having a semantic description of an environment is to be able to represent it in a way that is easily understood by humans and at the same time allows for manipulation by machines. An incomplete model can be updated with additional knowledge through inferring. This section identifies classes and properties that constitute an ontology to represent graphical user interfaces in applications. It also presents cases where it can be used directly and illustrates it with an example of a trivial graphical user interface.

3.1. GRAPHICAL USER INTERFACE ONTOLOGY

The basic class of the Graphical User Interface ontology is `Object` which is a direct descendant of `Thing`. The class represents all graphical elements of the interface (aka *widgets* or *controls*). All properties in the ontology are defined regarding this class or one of the classes derived from it. Thanks to that all definitions reference only terms contained within the GUI ontology itself without imposing a dependency on other ontologies. Of course it is possible to introduce such dependencies on demand by providing equivalents between terms defined in different ontologies. This provides other advantages, some of which will be mentioned later in this chapter.

The first relation between objects is that they can contain other objects – they can be treated as boxes that have child boxes where each child box occupies a region of the parent (in terms of geometry). This is a classic approach that lets one compose widgets from other widgets. For example a check box can be composed from two items – a little rectangle that can be marked as checked or not and a textual label describing the meaning of activating the box. This relation is defined by the `contains` and its inverse

`isContainedIn` object properties. For simplicity it is assumed that those properties are not transitive and with a natural constraint that each object can have at most one parent object we have a complete information about the hierarchy (which is a tree) of objects.

We can distinguish a situation where an object is not contained in any other object (cardinality of its `isContainedIn` property is exactly 0, the object is a root of a hierarchy). In that case the object represents the `Window` class and can be perceived as a top-level window on the screen that can be moved around, resized, hidden, etc. The whole system can have many disjoint object trees representing different windows and their respective content.

Another important class is `Property`. It can be used to describe additional features of objects. For example a control displaying a number might be defined as *an object that is described by the integer "value" property*. This is a generic way to define more detailed classes. Properties can be assigned to GUI objects through `describes` and `isDescribedBy` OWL object properties. This allows for classifying groups of objects based on features (functionality) they offer to the environment and the user they interface with in particular.

Aforementioned classes allow to describe arbitrary graphical objects and their hierarchies. This is more or less equivalent to what is offered by methods mentioned in section 2.2. of this chapter. To have a complete model of a graphical user interface it is also useful to have a description of logical connections between objects.

The architecture is based on a concept of objects communication changes of their state to their environment. Those changes can be intercepted by other objects and cause their state to change as well. Through this chain reaction the system reaches a new stable state, just like in the finite state machine approach.

This functionality is mirrored in the GUI ontology through an additional set of semantic terms. Informing about changes and intercepting them is done by classes `Notification` and `Action`. The former carries information ("signature") about a particular group of changes. `Action`, on the other hand, is functionality that can be invoked on an object as a result of intercepting a notification from another (or the same) object.

To be able to tell which object intercepts which notification from a particular sender and what action gets executed as a result, the concept of `Connection` is introduced. It is a quadruple containing the `source` and `target` objects as well as information which `Notification` the connection `isTriggeredBy` and what `Action` it `activates`. For example we can say that t*here exists a connection between the notification 'validated(boolean)' of object A and action 'setEnabled(boolean)' of object B.*

There is also a special class of objects that only serve as visualization of some other data. We can call this class a `VisualizationObject` class. Its instances don't interact with the user and they are not a source of data for other objects. An example of this

class could be a textual label but only such a label that doesn't contain elements the user can interact with (such as hyper links).

Based on what is written here we can deduct that instances of this class have to fulfil a prerequisite of not being a source of any `Notification`. The condition is not sufficient – there may exist objects that don't send any notifications but are not visualization objects. The ontology may be incomplete and extending it with new facts and rules might introduce notifications into the object or the presence of ability to send notifications can depend on the environment the object is placed in. For example if we treat the ontological model as an abstract definition of an interface, its concrete implementations in different toolkits might induce different properties on the final object.

To sum things up, we can say that a generic `Object` subclass or individual is represented by a set of `Properties` it is described by, `Notifications` it can send and `Actions` it can activate as well as a set of `Objects` it contains and the `Object` it is contained in. Furthermore `Objects` are linked together through `Connections` that allow to tie some of the logic, specifically changes in state of the user interface with the representation of the presentation layer itself.

3.2. EXAMPLE OF USAGE

In this part of the section an example is presented that illustrates how a natural language description transforms first into a semantic definition and then into final looks and logic of the user interface. It is shown how to exploit the semantic model to generate the source code for an application that implements it in various environments and conditions. The interface is described as follows:

> *Window contains two objects – Button and Line Edit. Line Edit is a single line of editable text. When Button is clicked, the text from the editable box is cleared. Line Edit is on the right of Button.*

From the interface description we can extract (or reason) the following facts:

- there are three distinct `Objects` in the scene – `Window`, `Button` and `Line Edit`,
- `Button` and `Line Edit` are siblings and not top-level windows – they are both part of `Window`,
- it is not possible to determine if `Window` is a top-level window – it can but doesn't have to be one,
- `Button` has a *clicked* `Notification`, `Line Edit` has a *clear* `Action`,

- there is a `Connection` between the two objects linking the *clicked* `Notification` to the *clear* `Action`,
- there is a *on the right of* relation between `Line Edit` and `Button`.

Figure 1 shows a mockup graphical representation of the described user interface. This definition can now be used for example to generate source code for an arbitrary language or toolkit, provided there exists a tool that understands the semantic description and is able to generate source code for a given target architecture.

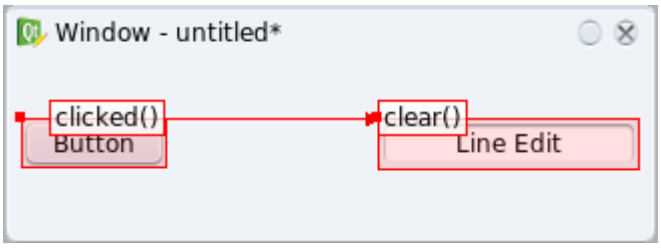

Fig. 1. Example user interface

The generator should create code suitable for the target platform based on the given semantic input as well as any other ontologies or other sources of rules. Please note that it's not explicitely mentioned anywhere that `Button` is actually what we (humans) understand as a button – the latter is used strictly as the name for the object. The object can be an instance of any type that provides the *clicked* notification (like a hyperlink). An additional ontology (for a particular toolkit) can contain definitions of classes providing the desired functionality and the generator shall pick any of those that don't cause inconsistencies in the model. Extra rules may reveal preferences for choosing one representation over the other based on some conditions (like platform guidelines such as those published by Apple in [1]) or external constraints. In addition to that, if the toolkit ontology contains the definition of the *on the right of* relation and the generator understands it, it can also be used to enrich the final output.

A conclusion is that the same semantic model of the GUI can result in different looking applications. However, the logic of the user interface should be retained.

Figure 2 presents the user interface of an application for Qt4 toolkit that implements the example. It uses the *signals and slots* paradigm of Qt4 to provide the logic of clearing the contents of the input object upon clicking the button. It can be compiled into a fully working application.

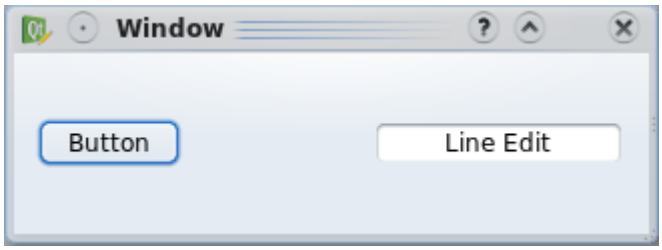

Fig. 2. Final application UI

4. CONCLUSIONS

Having an ontological model of the user interface has a benefit of being able to “plug in” concepts defined in other ontologies. With the expected inflation of popularity of semantic web we can expect to have ontologies from many new domains available. This will allow to express the user interface using terms from a specific, dedicated to the problem, domain, more high level expressions or equivalents to well-known concepts.

Another important field of use is creation and usage of semantics-enabled code generators. Having an abstract model of a user interface makes it possible to implement tools that would transform such definition into source code for different arbitrary development environments. There would be at least two advantages over existing notations. One is that we would have more defining symbols available.

The other allows non-technical business clients to express their expectations of the graphical user interface using a natural language. They could use the terms they are familiar with in their daily work in the area of life the future software will be dealing with. Such approach would require additional effort to transform the definition from a human-spoken language to a machine-understandable alphabet.

A good formal description of application user interface opens a possibility to use it for testing the graphical user interface itself and not the application logic controlled through the interface. This concept has been broader described in [7].

The work is not complete. The GUI ontology can be extended to cover new situations and define standard terms related to user interfaces. Research needs to be done whether it is possible and useful to generate UI representations for particular languages and toolkits.

REFERENCES

[1] APPLE: *Macintosh Human Interface Guidelines*, 2nd Edition, 2008

[2] ENG E., *Qt GUI Toolkit: Porting graphics to multiple platforms using a GUI toolkit*, Linux Journal, 1996 (31es):2, 1996

[3] SCHEIFLER R.W.: *X Window system protocol, version 11*, RFC 1013 (Memo), 1987

[4] TRÆTTEBERG H.: *Model-based User Interface Design*, PhD thesis, Norwegian University of Science and Technology, 2002

[5] WIKIPEDIA: *Comparison of X Window System desktop environments*, Jul 2009

[6] WIKIPEDIA: *Desktop environment*, Jul 2009

[7] WYSOTA W., *Testing User Interfaces in Applications*, Proc. of 1st International Conference on Information Technology, 2008, pp. 425–428

[8] YIP S.W.L., ROBSON D.J.: *Applying formal specification and functional testing to graphical user interfaces*, 5th Annual European Computer Conference, IEEE, 1991, pp. 557–561

[9] YIP S.W.L., ROBSON D.J.: *Graphical User Interfaces Validation: A problem analysis and a strategy to solution*, Proc. of the Twenty-Fourth Annual Hawaii International Conference on System Sciences, 1991, pp. 91–100