MDE Reading Report: Domain-Specific Modelling of complex User Interfaces

Corrado Ballabio

corrado.ballabio@student.uantwerpen.be - c.ballabio9@campus.unimib.it
University of Antwerp - University of Milano Bicocca

Abstract

This paper will present a couple of extended version of Statechart diagrams, hybrid languages that are capable to model complex user interfaces at a detailed level, with a brief explanation of the added elements and the capabilities of them. It will start explaining the syntax and the semantic of Interactive Object Graph, that includes special kind of nodes and relations for designing graphical widgets. Then it will introduce a mapping to SCCDXML, that merges Statechart and Object Oriented features for a rapid prototyping of models.

Keywords: Statechart, IOG, user interface, SCCDXML, model transformation

1. Introduction

Statechart formalism has the ability to model complex, timed, interactive discrete-event systems in an incredibly efficient way, but their expressiveness could be limited. For this reason they are not suitable to model applications with higher behavioural complexity. For overcoming this limit some hybrid version of it has been implemented, merging different formalisms in order to achieve a wider range of modelling capabilities. In this paper two extended version are presented: IOG and SCCDXML. The goal of this paper is to create a tool that starting from a user interface specification will generate code and allow a valid testing through a model transformation between IOG and SCCDXML.This will be implemented with AToMPM modelling environment.
Chapter 2 will present Interactive Object Graph method, introduced by

David Carr, and briefly explain its features. Chapter 3 will motivate the need of mapping IOG into another formalism, and SCCD will then be introduced. In chapter 4 future work will be exposed and then the last chapter, the 5th will conclude this paper.

2. IOG specification method

IOG (Interactive Object Graph) is a method for specifying user interfaces with the help of an extended version of the Statechart Diagram formalism introduced by David Carr in 1994. This extension was designed for widget implementation, and adds special nodes for increasing IOG readability and map user behaviour into the diagram. Main goals of the Interactive Object graph are to create a basis for rapid prototyping, to design complex user interfaces and to reduce the work required to design a user-computer dialog. Carr only defined the syntax and the semantic of the item he added, showing some example widget modelled with IOG. They will be now briefly explained:

2.1. New elements of the syntax

IOG method introduces some new states, meta-states and special arc types. Meta-states are particular objects that can contain multiple normal states or other meta-states. For avoiding the arc explosion problem, all the transitions that start from meta-states are inherited by the internal ones.
Here the new nodes are briefly explained:

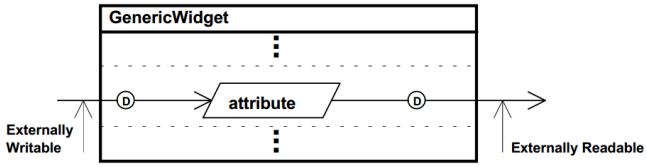
- **XOR meta-state:** contains a sequential transition network in which only one state is active.

- **AND meta-state:** can contain more then one transition network, and each one of them is executed in a parallel way.

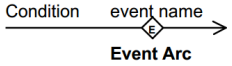
- **History states:** there are two different types, differing each other in the way that treat a return when last active state was a meta-state.The **H** state restarts meta-states at their start state and provides one level of history; the **H*** state restarts meta-states at their history state when they have one thereby allowing multilevel history. History states can only be contained in XOR meta-states, and they help to avoid states explosion.

- **Display state:** are represented in parallelograms and are control state that have a change in the display associated with them.

- **Data objects:** they never get the control signal, they just represent the storage of a data item. They are used in combination with Data Flow Arcs: if an arc of that kind enters the node the data object is updated, if it exits it represents a change in the value. An arc without destination symbolize an externally readable data while an arc without source represent an externally writeable data.



- **Event arc:** allows the designer to define messages, is represented with a special transition passing through a letter **E** in a diamond shape.



2.2. IOG abstract model

Now that all the nodes that compose the Interaction Object Graph are fixed, is necessary to understand how IOG abstracts interface and the dialogue between the user and the diagram.
A set of objects that can describe and discriminate the transitions between states is now shown:

- **BNS:** boolean, number(both $\mathbb{N}$ and $\mathbb{R}$ are contained) and string values are handled with the usual operations and correspond to the usual meaning associated with them.

- **Point:** corresponds to an ordered pair of numbers and denote a spatial coordinate. Values of a point variable p are assigned through the usual notation p = (*value_x*, *value_y*) and can be accessed via dot notation *p.value_x* and *p.value_y*.

- **Region:** is a set of points on the display identified with the upper left corner point, a coordinate pair named *location*. A couple of operators are also defined for this element: *region.size()* returns the values of the dimensions of the region, and *region.in(point)* is a boolean operator that states if the value *point* belongs to the region or not.

- **Icon:** are simple regions that can display pictures or graphical representations of other values. In addition to the region operators, icons also have *draw* and *erase* operations, that show it on the display. It is also possible to visualize a BNS value inside an icon: with the operator *icon(BNS_value, point_value, font, fontsize)* BNS variables are converted into a text representation with the specified font and size, and then shown in the display inside an icon at the desired position.

- **View port:** is a region that has a mapping function associated for some underlying application data. The mapping function works in two steps: it first applies a conversion for a graphic representation and then applies a projection that translates and scales the data onto the display.

- **Window:** they group all the objects illustrated so far into a stack, adding a *level* attribute that gives a hierarchic order to each object. Windows with lower level are at the top of the stack, so they cover all the other overlapping windows that have higher levels.

- **User input:** play a central role in IOG, it is the mean with which the user can interact with widgets. All the inputs are mapped into different transitions, depending on the kind of action that has been performed. Keyboard inputs have a string representation that indicates the key event that triggers or the text that has been typed. Each mouse input on the other hand has a special notation that characterizes the transitions. For getting the position, the change of the position and the movement of the mouse pointer $M@$, $M\Delta$ and ΔM notation are used. Mv and $M\hat{}$ respectively indicate the press and the release of the click button of the mouse. *in[region]* returns if the mouse pointer is inside a region, while $\sim$*[region]* and *[region]*$\sim$ are notations for the entering and leaving of a region event.

2.3. Graphical widgets

With the IOG method that has just been illustrated, David Carr modelled 5 widgets for testing its possibilities. These widgets are a range-selection slider, an alpha slider, a node-link tree viewer, a treemap viewer and a secure toggle switch.

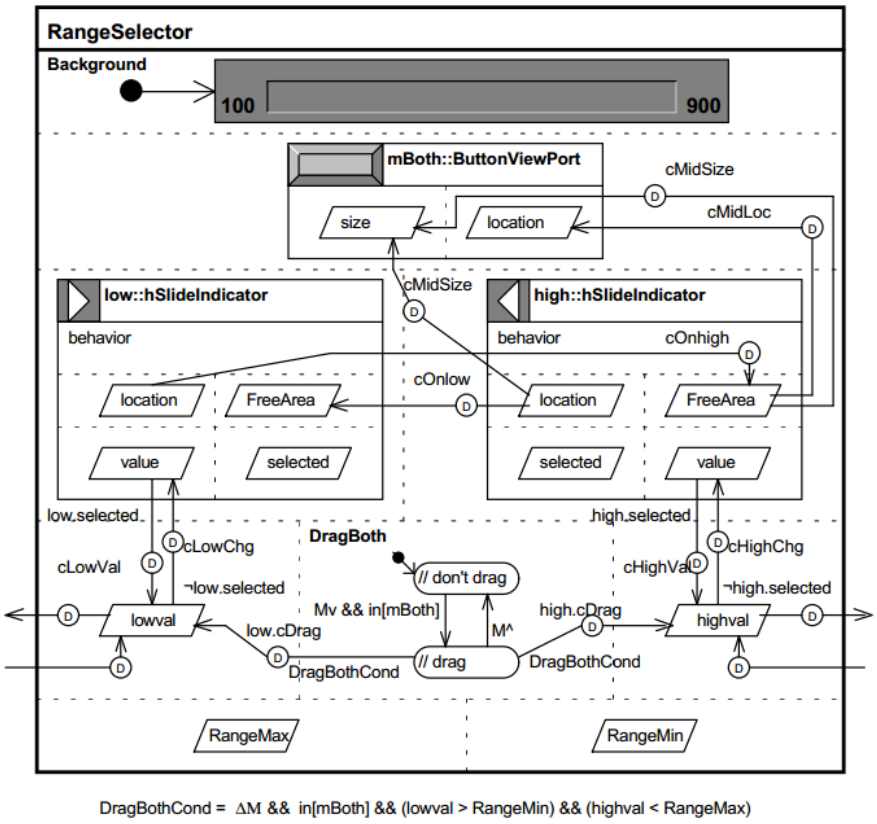


Figure 1: IOG implementation of a range-selection slider

3. IOG Transformation

Even if IOG formalism is not so recent (September 1994) it still lacks a fundamental part. What is missing is a transformation model that allows designer to draw user interfaces and then directly executing them for prototyping purposes. For making it possible it is necessary to transform it into another formalism and then generate code. A good candidate is SCCD language, an hybrid that combines Statechart and Class Diagram filling the software complexity gap and making possible to model complex graphical user interfaces at a detailed level. Class Diagram adds the structural object-oriented expressiveness that was missing without it.

3.1. SCCD

SCCD language models the structure of a system with the concept of an object oriented class, and then associates it the definition of its behaviour modelled as a Statechart diagram. Representation of such a language is done with its concrete syntax colled SCCDXML. This is an extension of SCXML, an XML-based markup language that provides a generic state-machine based execution environment based on Harel statecharts. With the addition of classes to the Statechart notation, is now possible to define attributes, relations with other classes and methods. Classes can be instantiated at runtime, making possible to create more objects at the same time. The main responsible for handling the instances is the *Object Manager*, it can create, delete, start the execution and associate classes at runtime. Once completed the SCCD model can be compiled in order to generate code for running application, choosing between the three programming languages that are supported: Javascript, Python and C# .

4. Future work

In a previous Model Driven Engineering project work, Pieter Aerts began implementing IOG formalism with AToMPM tool, starting from the definition of abstract and concrete syntax. Abstract syntax defines entities of the language, the relationships between them and also constraint on the values. Concrete Syntax creates a graphic visualization for each element of the AS. InteractiveObjectGraph formalism allows to reproduce David Carr IOG representation of widget in AToMPM environment.

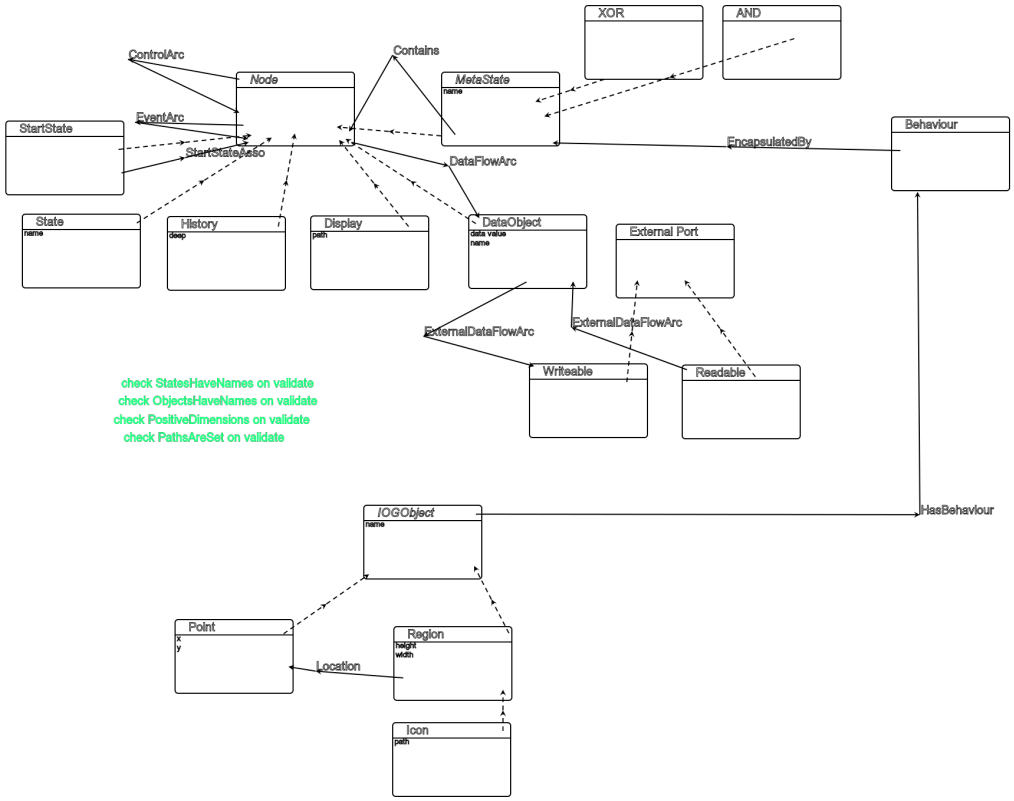


Figure 2: Interactive Object Graph abstract syntax specification

Future work will focus on the next step of AToMPM implementation: model transformation to SCCD. Each transformation rule is composed by a Left Hand Side (LHS) that represents the matching pattern to find, and a Right Hand Side (RHS), that contains the transformation to be made. Additional Negative Application Condition (NAC) is optional, it corresponds to the pattern condition not to be found to apply the transformation.
First step in the transformation is to map every element from the starting formalism (IOG) to a correspondent one in the final formalism (SCCD). For example every DataObject could be mapped into a class attribute, every user input could be mapped into an event, and so on. Then all the pattern matching condition need to be specified, together with the RHS. All the rules will be linked in a schedule that determines the order with wich the transformations will be applied in order to obtain the final transformed

model. Last step of the work will then be exporting the code and running the widget application. This will make possible to perform analysis and testing researches.

5. Conclusion

This paper explained the reason why an extension of Statechart formalism is necessary when it comes to model complex grafical user interfaces at detailed level. IOG and SCCD formalisms have been explained showing their syntax and their main purposes. The final goal is to create a transformation between these two languages and realise a working widget application.