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.

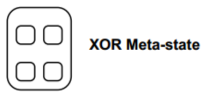
2. IOG specification method

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:

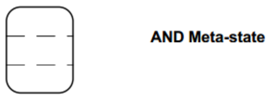
2.1. New elements of the syntax

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:

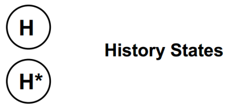
- **XOR meta-state:** contains a sequential transition network in which only one state is active.



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



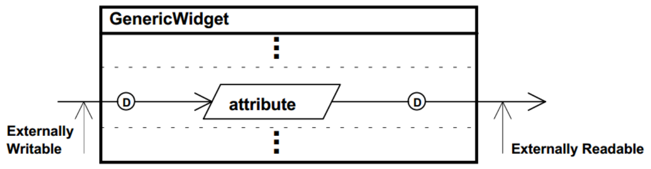
- **History states:** there are two different types, differing each other in the way that treat a return when last active state was a meta-state.The **H** state restarts meta-states at their start state and provides one level of history; the **H*** state restarts meta-states at their history state when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



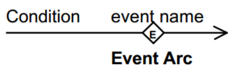
- **Display state:** are represented in parallelograms and are control state that have a change in the display associated with them.

ON
Display State

- **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.

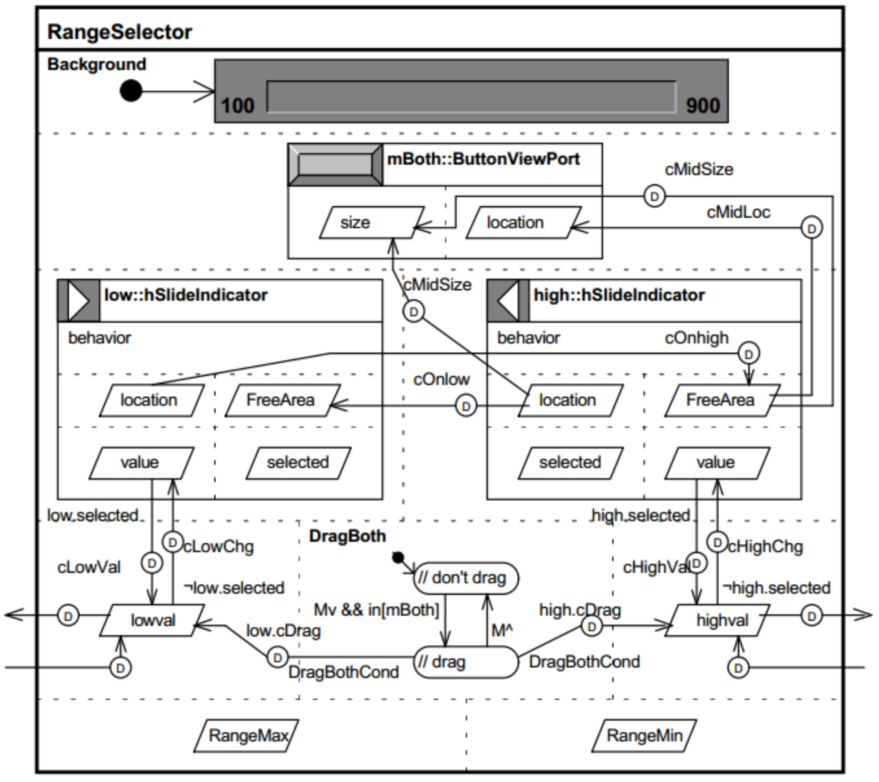
- **View port:** is a region that has a mapping function associated for some underlying application data. The mapping function works in two steps: it first applies a conversion for a graphic representation and then applies a projection that translates and scales the data onto the display.

- **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^* 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 ∼*[region]* and *[region]*∼ 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.



DragBothCond = ΔM && in[mBoth] && (lowval > RangeMin) && (highval < RangeMax)

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 called SCCDXML. This is an extension of SCXML, an XML-based mark-up 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. Implementation of the widget

In a previous Model Driven Engineering project work, Pieter Aerts begun the implementation of 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. This new InteractiveObjectGraph formalism allows then to reproduce David Carr IOG representation of widget in AToMPM environment without any loss of expressiveness.

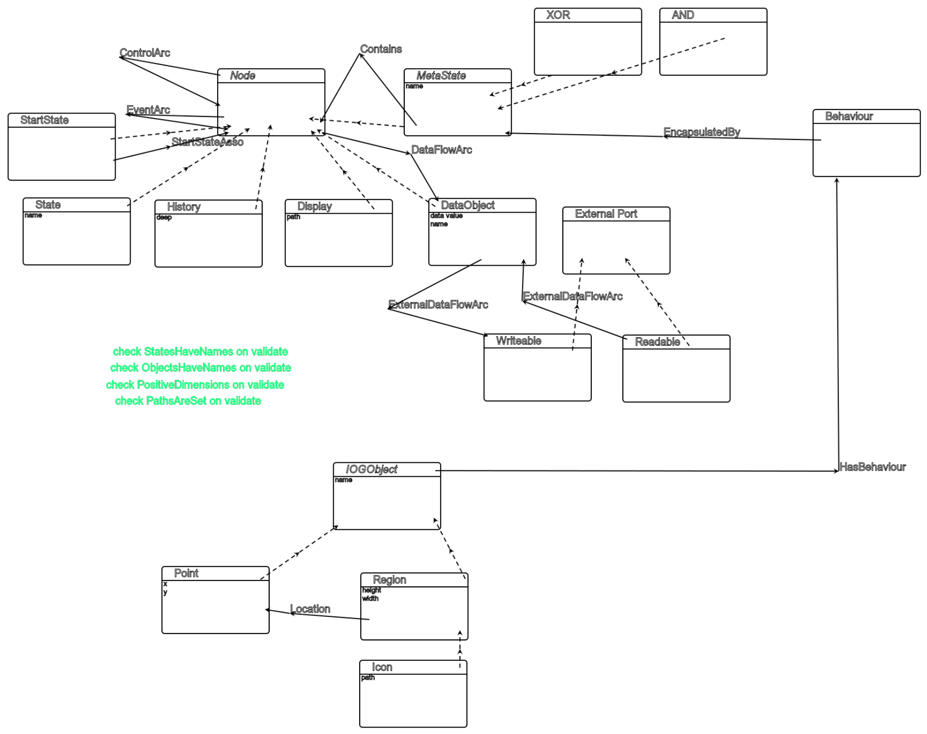


Figure 2: Interactive Object Graph abstract syntax specification

We decided to focus only on one particular widget specification, the *Drag Icon*, in order to create a strong and reliable basis for general widget implementation. *Drag Icon* widget shows an icon image inside a window. The icon is contained in two other regions: a smaller one called Free Area and a bigger one called Active Area. The icon can be dragged around inside the window with a click action and a continued pressure of the mouse, and it follows the pointer as long as it stays inside the borders of the Free Area; outside of these limits the icon stops following it. When the mouse is unclicked the icon position is anchored in with the new position. If the mouse is moved outside of the bigger region, the Active Area, the icon jumps back to its original position when the button is released.

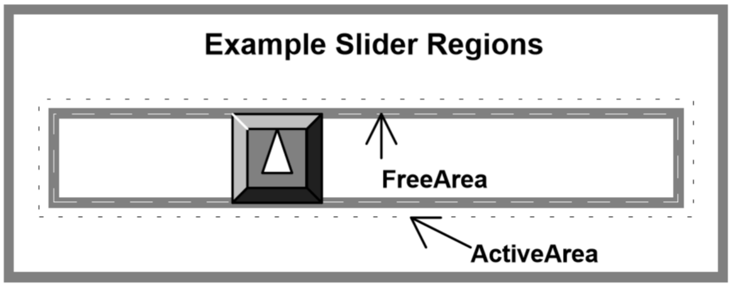


Figure 3: The areas of the drag icon widget

With the concrete syntax elements defined with AToMPM we created the model that specifies the behaviour of the widget. Starting from this point, we will focus on the process of transformation of this model to a SCCD one, and then on all the operations necessary in order to obtain a final functioning widget. This procedure is composed by several steps, and here they will be all explained starting from the fixing of the abstract syntax to the interface implementation in Python.

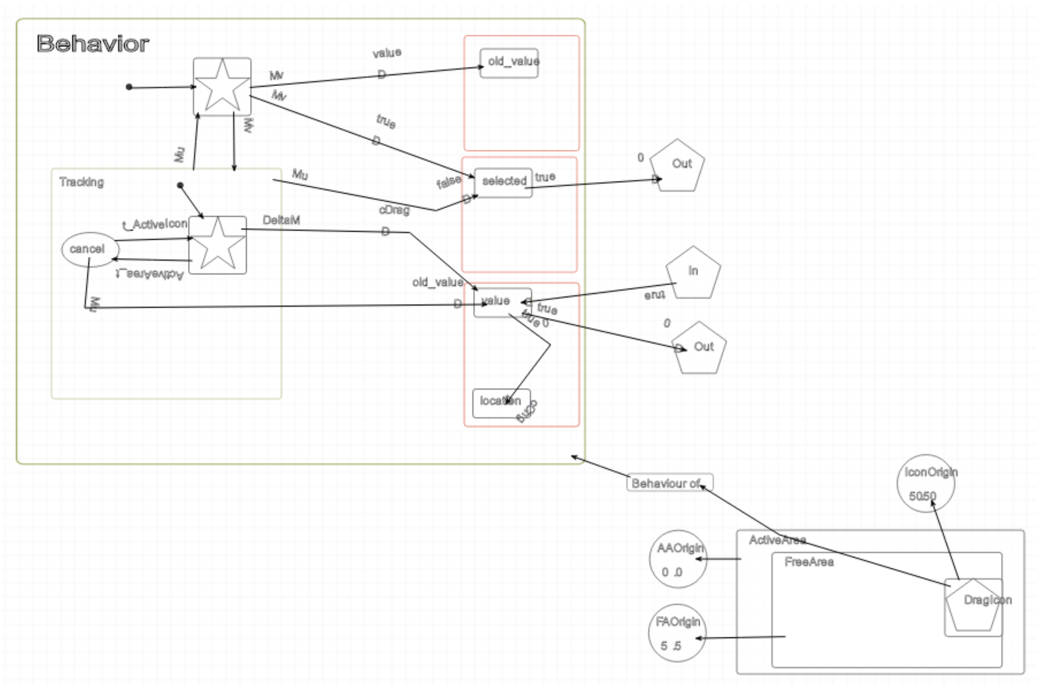


Figure 4: IOG diagram of a draggable icon widget

4.1. Changes on the previous work

Although the abstract syntax specification of the formalism was already efficiently implemented, a couple of modifications and adjustments were necessary in order to correctly create all the transformation rules. Here is a brief list of the main changes applied.

- The attribute **name** has been added to *Display* class attribute list. This is an indispensable change because it allows to uniquely identify icons element during transformation rule application.

- A new containment relation between IOG objects named *IOGContains* was created. It helps draw widget models in a better way since it is now possible to explicitly create a window/icon hierarchy starting from the AToMPM model specification.

- Some of the control and data flow arcs names has been modified in the model representation. This was necessary because during the SC-CDXML export phase some problem rose due to the presence of some reserved XML characters. The display names were hence modified with the "ordinary" characters preserving their meaning.

  – Mv && in[Idle] ⟶ Mv
  – Mˆ ⟶ Mu
  – DeltaM && in[FreeArea] ⟶ DeltaM
  – ∼[ActiveIcon] ⟶ t_ActiveIcon
  – [ActiveArea]∼ ⟶ ActiveArea_t

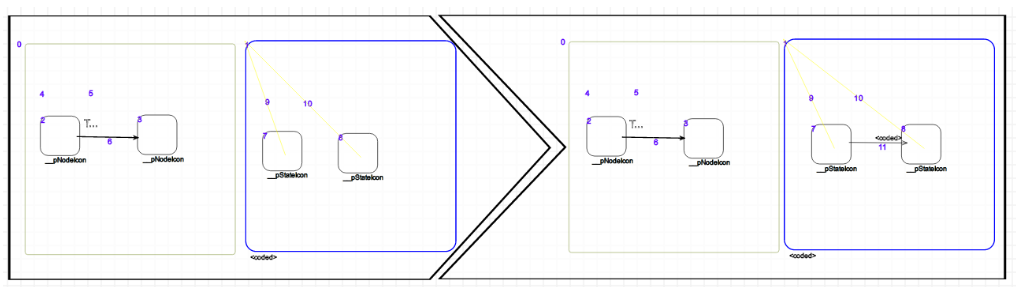


Figure 5: The rule for connecting nodes in the statechart

4.2. Transformation rules

In order to obtain a behaviourally equivalent SCCD from the IOG diagram, all the elements of it had to be coupled to elements of the other one. AToMPM model transformations are composed by two sides: a left hand one (LHS) that specifies a particular sub-pattern of the model that is intended to be transformed. If it is present inside the specification under transformation, this is replaced with the other pattern of the right hand side(RHS). All the transformation rules are now explained in a practical way, in the same order as they are applied in the transformation schedule. In that way the whole process is shown step to step for a better comprehension of its dynamic.

The starting point of the transformation process is the identification of the IOG object that encapsulate the behaviour of the widget. This element is transformed in a class linked to a Composite state. This state represent the principal element of the statechart that will contain all the states and transitions that model the widget behaviour. It is then named the same as the XOR state of the IOG diagram.
Transformation continues with the specification of the constructor method inside the class. In this method all the "visual attributes" of the widget such as dimension of areas, location of icon and source path of the image are instantiated.
Then a series of rules create all the nested metastates and nodes inside the statechart. XOR metastates are transformed into Composite states and Display node and state node into basic states. All the other different nodes are ignored because only these two have a direct graphical counterpart in SCCD formalism. Once again transformations assign the same name to the new states in order to be able to couple them in further rules. Since all the nodes are created, with an additional rule is possible to mark the right ones as initial.
Next step is to establish all the connections between nodes. This is made by finding nodes linked by a control arc and then connecting the corresponding states with a transition arc. One more rule sets the firing event of the transition the same as the condition attribute of the control arc.
Last transformation that need to be made is setting the actions that statechart transitions trigger, they are the correspondent element to a dataflow arc that updates a data object. For this reason the action methods are *setters* that change the value stored in a node. For making this call possible, the method signature has to be saved inside the class that encapsulates the

behaviour.

After the rules schedule finishes the transformation, a final SCCD model is obtained. This model has considerably less visual elements than the original one, but there is not loss of information because most of IOG elements are mapped to single SCCD elements, like for example nodes, icon, start states, dataflow arcs, control arcs and data objects are all mapped to nodes and transitions.

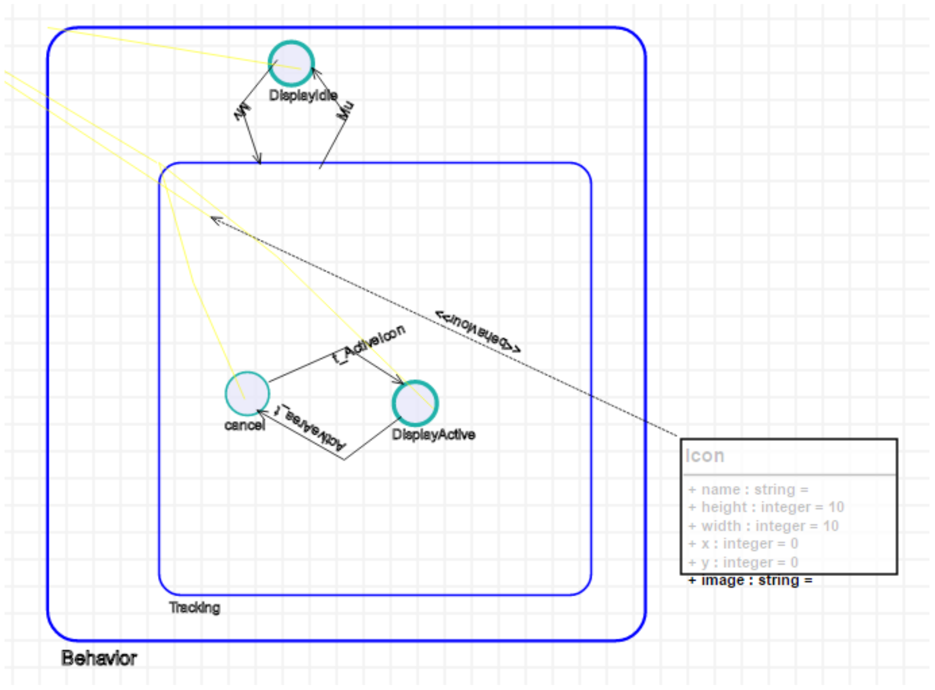


Figure 6: The model after the transformation

4.3. GUI implementation

With the compile buttons of the SCCD formalism was then possible to export the model in a XML file named *model.xml*. This was then compiled with other two .xml files in order to obtain the final python file, *Icon.py*. This file represents the SCCD model structure and behaviour, exactly as specified in the visual model with AToMPM. The final step before directly testing the widget is to build the graphical user interface of it. The python file *IOGinterface.py* does that, with the help of *Tkinter* GUI interface.
In this class canvas, icon object and all the variables are instantiated, is

then *Icon.py* that sets them to the right values and dimensions. When the GUI is displayed, image object is bound with listener methods that handle the click, drag and unclick events. These methods send input events to the SCCD diagram when called. In that way actions are generated, and the values that used to be represented by object datas are set to their new value. The widget is generated and is now possible to use it and test its behavioural correctness.

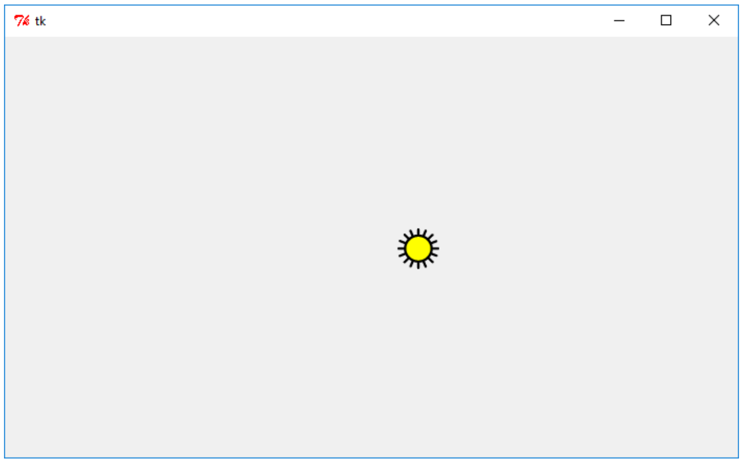


Figure 7: The final working drag icon widget

5. Conclusion

The goal of this project was to create a graphical widget interface starting from an Interaction Object Graph model, and has now shown all the aspects of it. Nevertheless is possible to continue the work on some aspects that still need to be implemented and refined.

Since this work focused on the development of a single widget, the *DragIcon*, some element of the IOG specification, such as history states and event arcs, are not handled by the transformations because they did not appear in the model under study. In other cases some element are present but not in all their usage. For example data objects are present together with data flow arcs, but only with incoming transition, representing the update of an attribute. Other usage like the reading of an attribute are not present in this model, so the specific rule is still missing. For all these reasons, for a generic and complete transformation rule set, some transformation pattern are still to be implemented. A good starting point would be to begin drawing all the other widgets models such as the *range-selection slider* or the *secure switch*, and then check if all the elements included in the model are translated. If the transformation is incomplete new rules have to be modelled.

Once complete, the IOG formalism represent also a perfect tool for the creation and the testing of a whole new possible set of widgets, since the creation of new objects is nearly immediate after the specification of the structure.