Domain-Specific Modelling of complex User Interfaces

Pieter Aerts

pieter.aerts@student.uantwerpen.be

Abstract

This paper explores the implementation of the Interaction Object Graph (IOG) in AtomPM. The IOG is an extension of the Statechart formalism and was first introduced by Carr et al. (1994). It's primary use case is the specification of graphical widgets. AtomPM is a general purpose modelling environment in which we define an abstract and concrete visual syntax for the IOG-formalism. Through this method we create valid IOG-models, which can then be mapped onto other formalisms, such as SCCDXML , or generate code using a graphical user interface library. This could greatly speed up the development and implementation of new and existing graphical widgets.

Keywords:
IOG, AtomPM, SCCDXML, user interface widget

1. Introduction

Carr et al. (1994) introduced the concept of the Interaction Object Graph, a formalism designed for the specification of graphical widgets. It allows the design of widgets at a level higher than that of programming languages. This allows non-programmers to prototype and design complex user interfaces. We will implement this formalism in AtomPM, by going through the usual process of language design. Once we have a language through which can formulate valid IOG-models, we can map these widget specifications onto other formalisms or code. The SCCDXML-formalism seems like an excellent candidate. Alternatively, we could generate code which uses a graphical user interface library in python or javascript. This allows rapid prototyping and testing of new and existing widgets.

This paper is structured as follows: Section 2 gives an overview of the IOG-formalism. Section 3 describes the process of language design in AtomPM.

Section 4 explores the various options for model to model transformations and code generation from our IOG-models. Section 5 discusses future work and section 6 concludes.

2. The Interaction Object Graph

Interaction Object Graphs (IOGs) Carr et al. (1994) are designed to add widget specification to Interface Representation Graphs. They combine the data flow and constraint specifications of IRGs with the statechart execution model. Statecharts added four new states to the traditional state diagram: the XOR meta-state, the AND meta-state and two types of history state. These states are used in IOGs.

These meta-states can contain both normal states and other meta-states. Transitions from meta-states are inherited by all contained states. This helps reduce the problem of arc explosion. The XOR meta-state contains a sequential transition network where exactly one state inside of an XOR meta-state is active. The AND meta-state on the other hand contains more than one transition network and allows for parallel execution.

The history states work the same way as they do for Statecharts. When a transition passes control from a meta-state, it remembers its last active state in the history state. When a transition returns control to the history state, the meta-state is returned to the remembered state. There are two types of history states. The **H** state restarts meta-states at their start state and provides one level of history. The **H*** works the same way but provides multilevel history.

IOGs add two additional node types to the regular statechart: data objects and display states (Figure 1). A data object represents the storage of a data item and control is never passed to them. They can only be destinations for data arcs discussed below. Display states are control states that have a change in the display associated with them. In IOG diagrams we use a picture of the display change to signal an update to the graphical representation of that item.

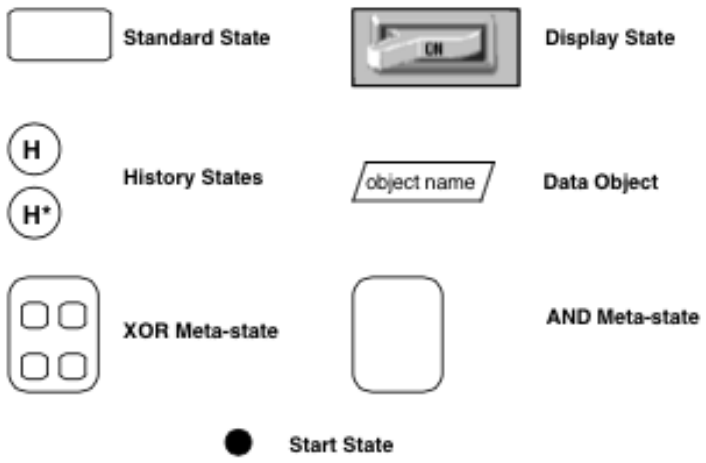

Figure 1: Node Symbols

IOGs also add two special arc types: the event arc and data arc. Events allow the designer to define "messages" which may be lacking in the underlying specification model. An event is represented by a special transition passing through and **E** in a diamond. (Figure 2)

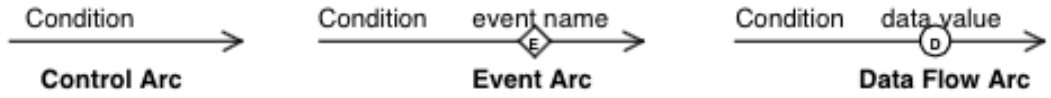

Figure 2: Node Symbols

Data flow is represented in a manner similar to events - an arc passing through a **D** in a circle (Figure 2). They can have any state as a source and can only terminate at a data object or have an unspecified termination. At least one end must be attached to a data object. Data flow arc with data object as a source, whose destination arrow is unspecified, and whose destination is outside of the containing interaction object, indicate externally-readable data (Figure 3). Data flow arcs with data objects as destinations represent updating the data object. If the arc's source is a control state, it represents a change in value when the arc conditional is satisfied. The label of the data flow arc holds the new object value. An arc without a source represents externally-writeable data (Figure 3).

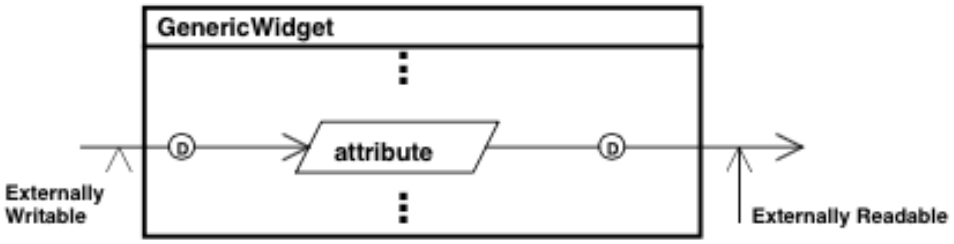

Figure 3: Node Symbols

IOGs also allow the use of constraints. These are useful in specifying one attribute of the user interface in terms of others. Data arcs support one-way constraints by expressing the data value as an equation in terms of other attributes. Together with a condition on data arcs, they allow the use of Boolean guards.

2.1. Transition descriptions

To describe the transitions between states we need an abstract model of the user interface and a description language for that model. IOG abstracts the interface into the following objects: Booleans, numbers, strings, points, regions, icons, view ports, windows, and user inputs. A brief description follows.

Booleans, numbers and strings are the usual abstractions with the usual operations. Numbers contain both the real and integer data types. Any of these can be converted into an icon representation by the icon() operator. This operation converts the Boolean, number or string to a text representation and then converts it to a picture.

Points are an ordered pair of numbers (x,y). They have the algebraic operator which are normally associated with them. It may be assigned value by writing **p=(x,y)**. Also, **p.x** and **p.y** represent the x and y coordinates from the point **p**.

A region is a set of display points defined relative to an origin called the **location**. The location of the region is always the point (minx, miny) where minx and miny are the smallest x and y coordinates in the region. Regions have a **size** operator which returns the height and width of the smallest rectangle which covers the region. Regions also have an **in()** operator which tests if a point is located in the region. This operator returns a Boolean value. Rectangles are commonly used for regions, but they are not restricted

to this shape. Note that a region cannot be visible on the display. It has no drawing operation associated with it.

Icons are regions with pictures. Some points in the region have a color number attached to them and are shown on the display. Icons add the operations **draw** and **erase** . If the origin of the icon is changed, there is an implicit **erase-draw** operation sequence. A view port is a region with an associated mapping for some underlying application data. This mapping consists of two parts: conversion to a world-coordinate-system graphics representation and projection onto the display.

Windows group the above objects together. They add a level attribute which determines window stacking relative to other windows. A window with a lower level obscures an overlapping window with a higher level.

All objects are addressed in the specification using a dot notation. For example, "win.icon1.location.x" would be the x coordinate of the location of the icon, "icon1", in window, "win".

User inputs are mapped to IOG events, numbers, points and Boolean values. Keyboard input is represented by quoted strings when the text is important or by key events when the event is important. The mouse is mapped to a point for location (**M@**), a point for relative change (**M** Δ), a boolean indicating it moved (Δ **M**), button change events (**Mv, M, M2v, ...**) and button states variables (**Mdn, Mup, M2dn, ...**). Since the value of the mouse location is tested frequently, **in**[**Region**] is written as shorthand for **Region.In(M@)**. Special notations ˜[**Region**] **and** [**Region**]˜ mean the event of the mouse entering and leaving **Region**. These symbols can be used in expressions alongside the logical operators. We included an example specification in Figure 4. It shows a simple draggable icon. An explanation and other extensive examples can be found in Carr et al. (1994) .

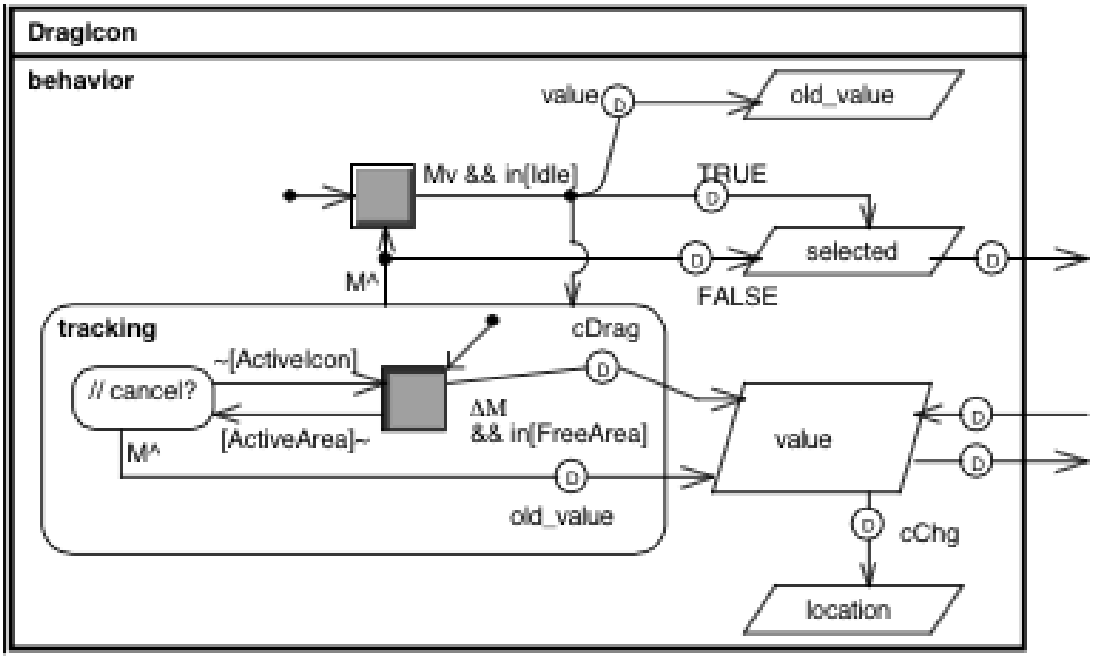

Figure 4: Specification draggable icon

3. Language design in AtomPM

In the previous section we discussed the IOG Formalism. We defined the abstract syntax by listing the different states and arcs and their semantic meaning. The above pictures which use these concepts already have a notion of visual concrete syntax. Both these aspects of the formalisms will have to be implemented in the modelling environment AtomPM.

When designing languages in AtomPM we first construct a meta-model for the language. This meta-model implements the abstract syntax for a given formalism, in our case, the IOG. We use the Class Diagram formalism to do this. The next step is specifying a visual concrete syntax using the ConcreteSyntax formalism. This makes it possible to have a truly visual syntax by defining the look of every class and association defined in the abstract syntax. With our meta-model and concrete syntax defined, we can create IOG instances that conform to our meta-model and are drawn by using symbols defined in the concrete syntax. The point where we can draw

models and check if they are valid in respect to some meta-model is a major milestone in designing a language.

4. Mapping IOG to other formalisms

In the previous section we explained how we can implement a given formalism in AtomPM by defining essential parts of the language. Now we can create IOG models which specify the behaviour and appearance of a certain graphical widget. The next step in the evolution of our language would be to map it to other formalisms, providing extra functionality already present in this other formalism.

For the IOG formalism, we see two paths we could take in this respect. Since IOGs are related to statecharts and have different object classes, mapping them to SCCDXML can be an interesting avenue. 'SCCDXML" stands for StateCharts Class Diagrams and XML. It combines the notion of classes and associations between them with the expressive power of state charts for the behaviour of these classes. The ability to execute valid SCCDXML models through the use of Javascript or python is a great thing for us, since we do not have to interact with these languages directly. The generating of code is already supported by the SCCDXML formalism. See Jonghe (2014) for a complete overview.

Another option would be to export our model to a different modelling framework such as Metadepth. Here we can use EGL (Epsilon Generation Language) to generate code in Javascript or python, ideally using a graphical user interface library which can draw the specified widget.

5. Future work

This report is written after the research-part of the project, which means all the implementation as outlined above still needs to happen. There are a lot of options when deciding how to exactly do this. We can start by designing a language for IOGs in AtomPM, later deciding where to map to or wether to generate code. This decision still needs to be made and will be greatly based on feedback received.

6. Conclusions

This paper gives an overview of the Interaction Object Graph formalism and how we can use it to specify graphical widgets. We explained how such a

formalism might be turned into a language by constructing a meta-model and concrete syntax for it in AtomPM. Through this language we could create valid IOG models. These specifications could then mapped onto different formalisms, such as SCCDXML, or be used to generate code. A graphical user interface library could be used to directly display the widgets and allow them to be executed and tested. Regardless of the path we choose to take, the goal is to evaluate the viablity of IOGs in user interface design by making it easier to execute and test our specifications.

References

Carr, D., Jog, N., Kumar, H., Teittinen, M., Ahlberg, C., 1994. Using interaction object graphs to specify and develop graphical widgets URL: `ftp://ftp.cs.umd.edu/pub/papers/TRs/3344.ps.Z`.

Jonghe, G.D., 2014. Statecharts and Class Diagram XML: A general-purpose textual modelling formalism. Master's thesis. Faculty of Science University of Antwerp Antwerp, Belgium.

de Lara, J., Guerra, E., Cuadrado, J.S., . meta-depth: A framework for deep meta-modelling. URL: `http://astreo.ii.uam.es/ jlara/metaDepth/`.

Syriani, E., . Atompm: A tool for multi-paradigm modeling. URL: `http://syriani.cs.ua.edu/atompm/atompm.htm`.