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 StateCharts and Class Diagrams (SCCD). We discuss how a model to model transformation from IOG to SCCD could be implemented. As an example, we have modelled the DraggableIcon widget from Carr et al. (1994).

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 . The SCCDXML-formalism Jonghe (2014) was the original candidate, but a SCCD implementation without XML existed already in AToMPM. We use the DraggableIcon widget from Carr et al. (1994) as our running example.

This paper is structured as follows: Section 2 gives an overview of the IOG-formalism. Section 3 describes our implementation of the IOG formalism in AToMPM. Section 4 discusses the mapping onto the SCCD-formalism. 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.

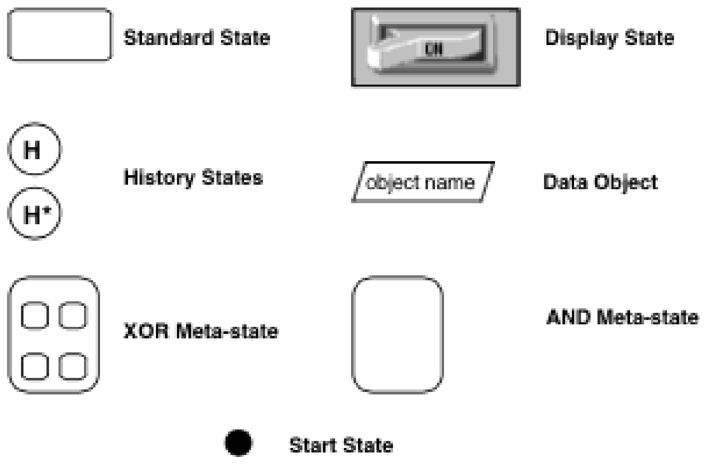

Figure 1: Node Symbols

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)

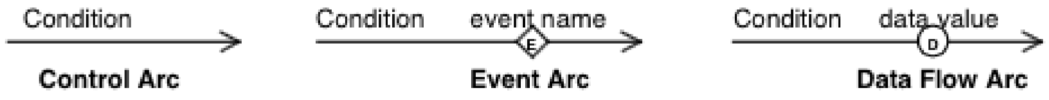

Figure 2: Node Symbols

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).

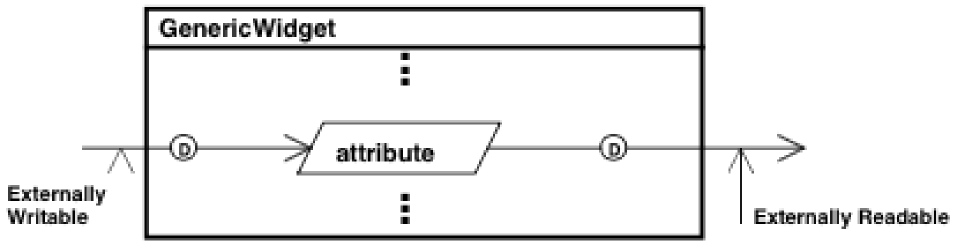

Figure 3: Node Symbols

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) .

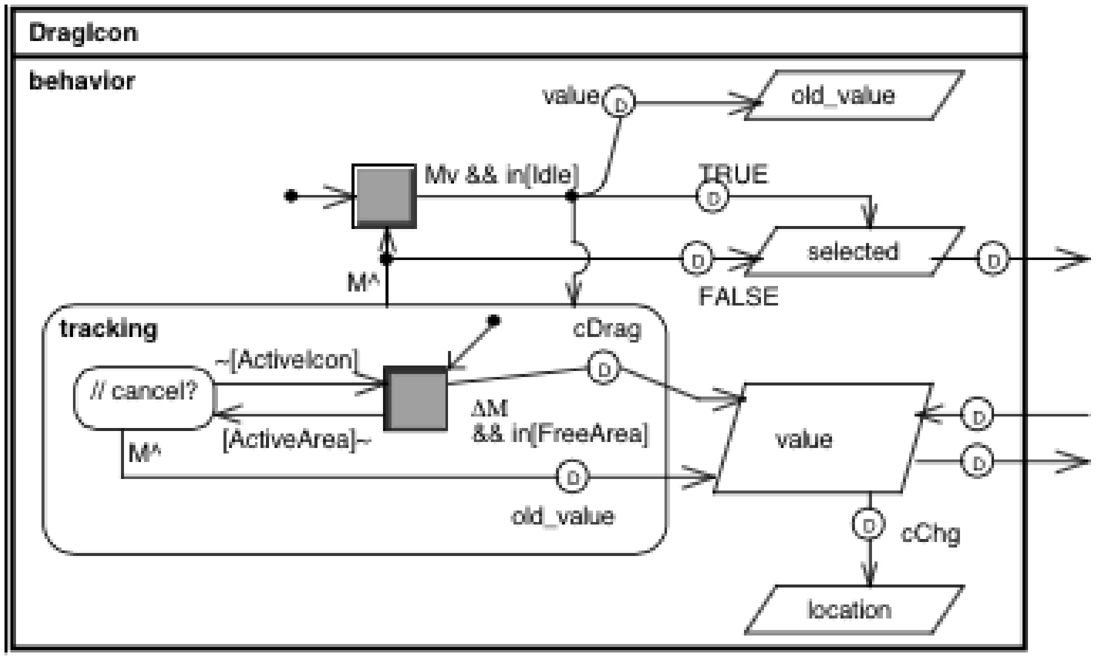

Figure 4: Specification draggable icon

3. Language design in AToMPM

his section describes our implementation of the IOG-Formalism into 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. In our implementation we have used a visual concrete syntax which is loosely based on the one above, but uses only the basic geometric figures supported by the concrete syntax toolbar in AToMPM.

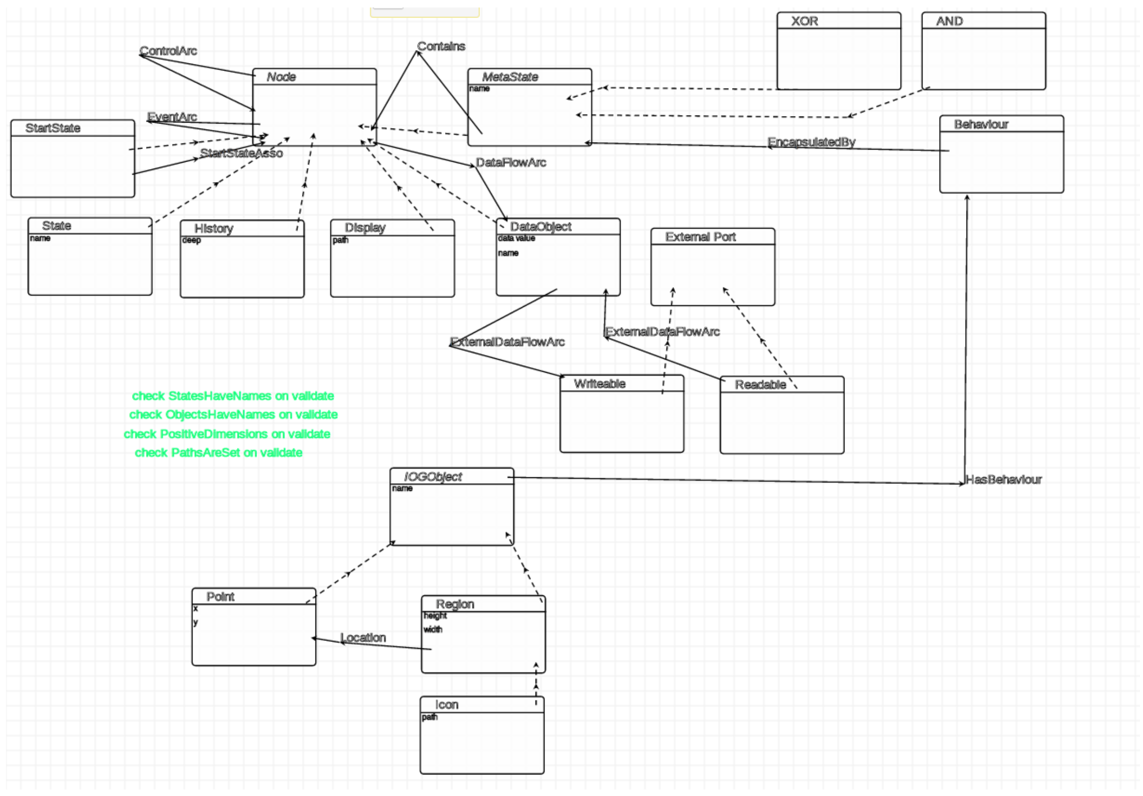

Figure 5: IOG Metamodel

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 have used the Class Diagram formalism to do this. Figure 5 shows our MetaModel. We make a clear distinction between two types of classes, IOGObjects and Nodes. IOG objects have been explained in the above section. We have implemented the Region, Point and Icon objects and have used the built-in AToMPM types for the simple objects. Instead of adding an x and y coordinate to the Region class, we have explicitly modelled the origin of a Region by assigning a Point to it through the association Location. This is also how Carr et al. (1994) describes the abstract syntax.

Each of these objects can have a behaviour assigned to it. The behaviour is encapsulated by either a composite or orthogonal state. In our running example, this is an XOR state. Nodes on the other hand are elements which can be part of the Interaction Object Graph. In our type hierarchy, every different graph element has the Node class as its parent. These children Startstate, State, History, Display, DataObject, MetaState. Some of these are required to have names. There are not that many data attributes in our implementation. History states have a boolean attribute, deep, which specifies if it is a regular history state or deep history state. The DataObject has a data value attribute which is an integer. Classes which have a specific visual representation attached have a path attribute. This path attribute should be set to the file location of the image they want to represent. At the point where the model is simulated, this should be translated in the image appearing on the screen.

Control and Event arcs are defined between Nodes. The special DataFlowArc start at any type of Node and ends in the specific Node DataObject. They are one of the more complex elements of our MetaModel. Aside from a firing condition which the other two types of arcs also have, it has a data value or data expression attribute. This is to support variable names for data as is done in the DragIcon example. These expressions are expected to evaluate to an integer value which can then be stored inside the DataObject it is pointing to. Often, they are application specific. We have also added External Ports. These can be used to link together behaviours of different IOGObjects and access data from outside the MetaState, but this has not been modelled.

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.

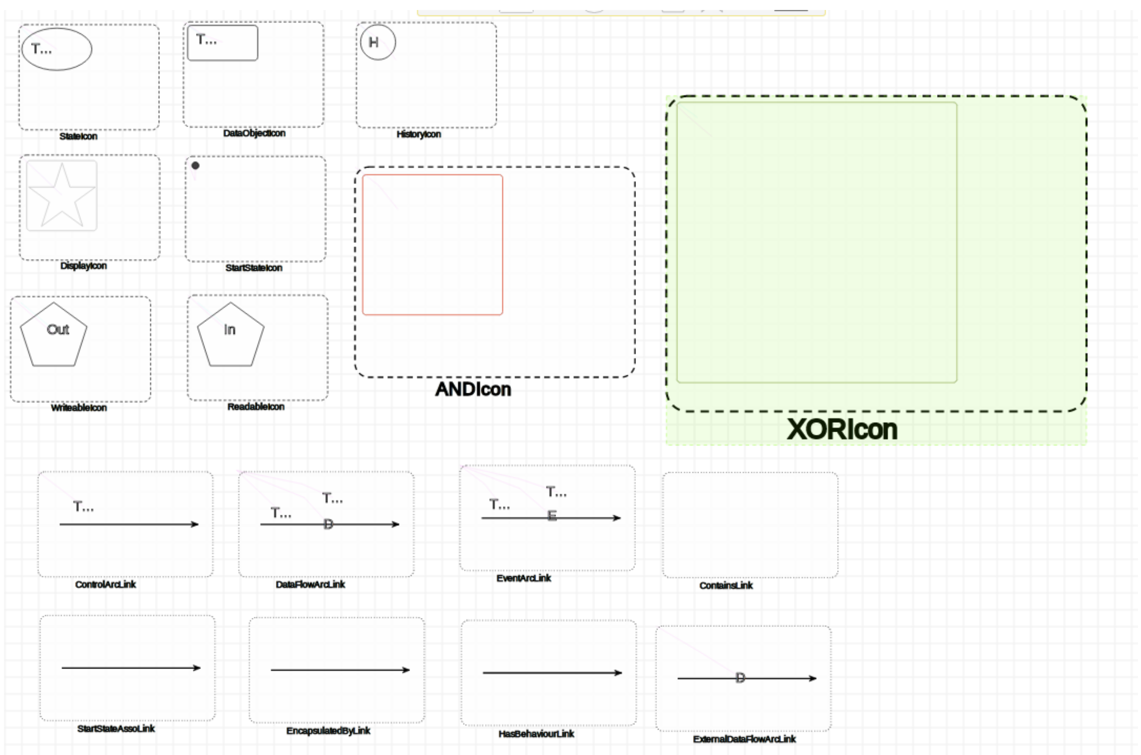

Figure 6: Concrete visual syntax for our Nodes

In figure 6, we see the concrete syntax for the children of Node and the arcs that are allowed between them. We have chosen color to differentiate between the two types of MetaState. XOR states are green, while AND states are red. The look of the other Node children is quite basic. We have worked a lot with the representation of their data attributes by means of text if they have any. For the arcs we have done the same, displaying the firing condition and their payload. For event arcs this the event they are generating, for data arcs this the data value or expression it is storing inside the DataObject. We check if the data expression attribute is non-empty. If it is, we display the data value which is valid in that case. If it is set to some expression, we display this over the data value.

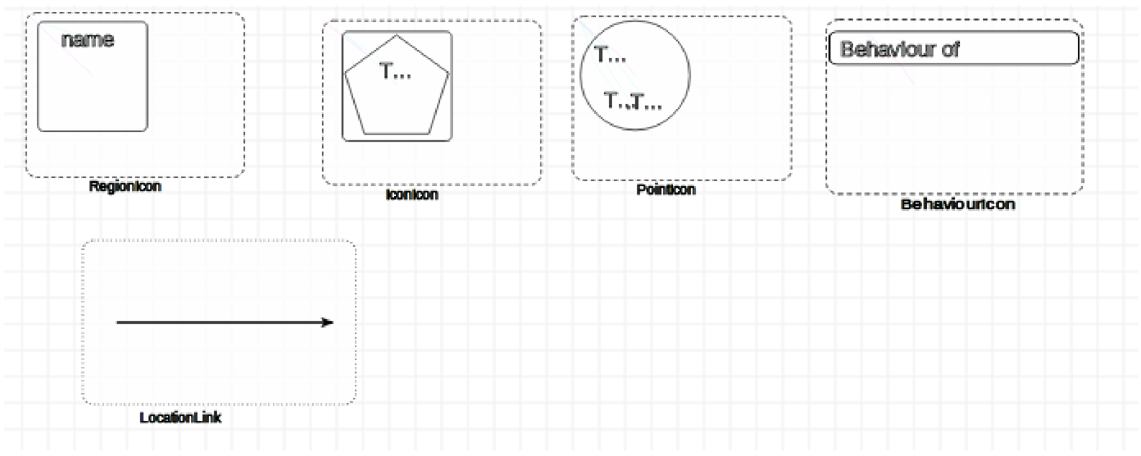

Figure 7: Concrete visual syntax for our Objects

In figure 7 we see the concrete syntax for our IOGObjects. We have used pretty basic geometric shapes again but the RegionIcon is able to scale according to its attributes. This is necessary, because its size and location has a real effect on the behaviour of other objects. A Point displays its name and x and y coordinate. An Icon simply shows its name. The BehaviourIcon is a simple label used to link together Objects and MetaStates encapsulating their behaviour.

We have some constraints on the model. These are pretty basic checks if all required attributes are set, such as names and paths. They also check if all coordinates are strictly positive. More advanced constraints such as checking if all state names are unique should be added in the future to constrain our language even more.

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.

In figure 8, we see an IOG model in AToMPM representing the same as the DragIcon example seen earlier.

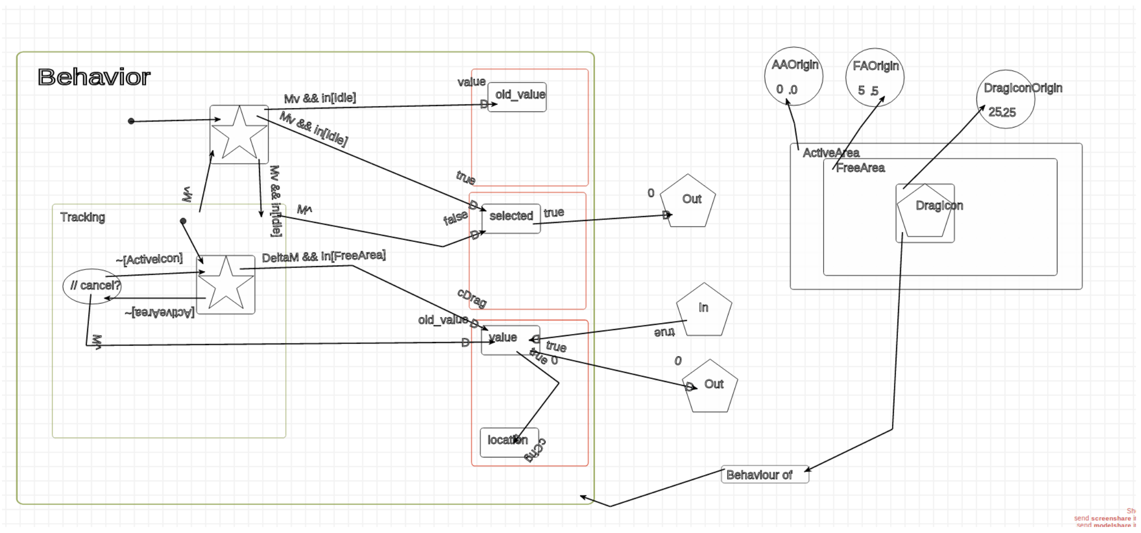

Figure 8: Example model implementing the DragIcon example

4. Mapping IOG to SCCD

In the previous section we explained how we implemented the IOG 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. We have chosen the SCCD Formalism for this, as it already has an implementation in AToMPM.

The transformation to another formalism in AToMPM is done by specifying rules which have left hand sides (LHS) and right hand sides (RHS). They can also have a negative application condition (NAC). A rule is matched when the pattern in the LHS is found. The LHS is then replaced by the elements in the RHS. In the NAC, we can define patterns or conditions on which the rule should not match. These rules are later chained together, in this case with Motif, which specifies the order of their execution.

For our formalism we have implemented only a few transformation rules on the highest level. Due to timing constraints we were not able to fully translate each IOG element to SCCD. We will give a theoretical approach to the lower level constructs such as the Nodes in the graph and how to translate them to SCCD.

Every IOGObject should get its own class in SCCD. IOGObjects which have behaviour assigned to it, will get their own state chart representation of this behaviour. In this first transformation, the attributes of each IOGObject become attributes of the newly created class in SCCD. Figure 9 shows the result of this transformation for the DragIcon example.

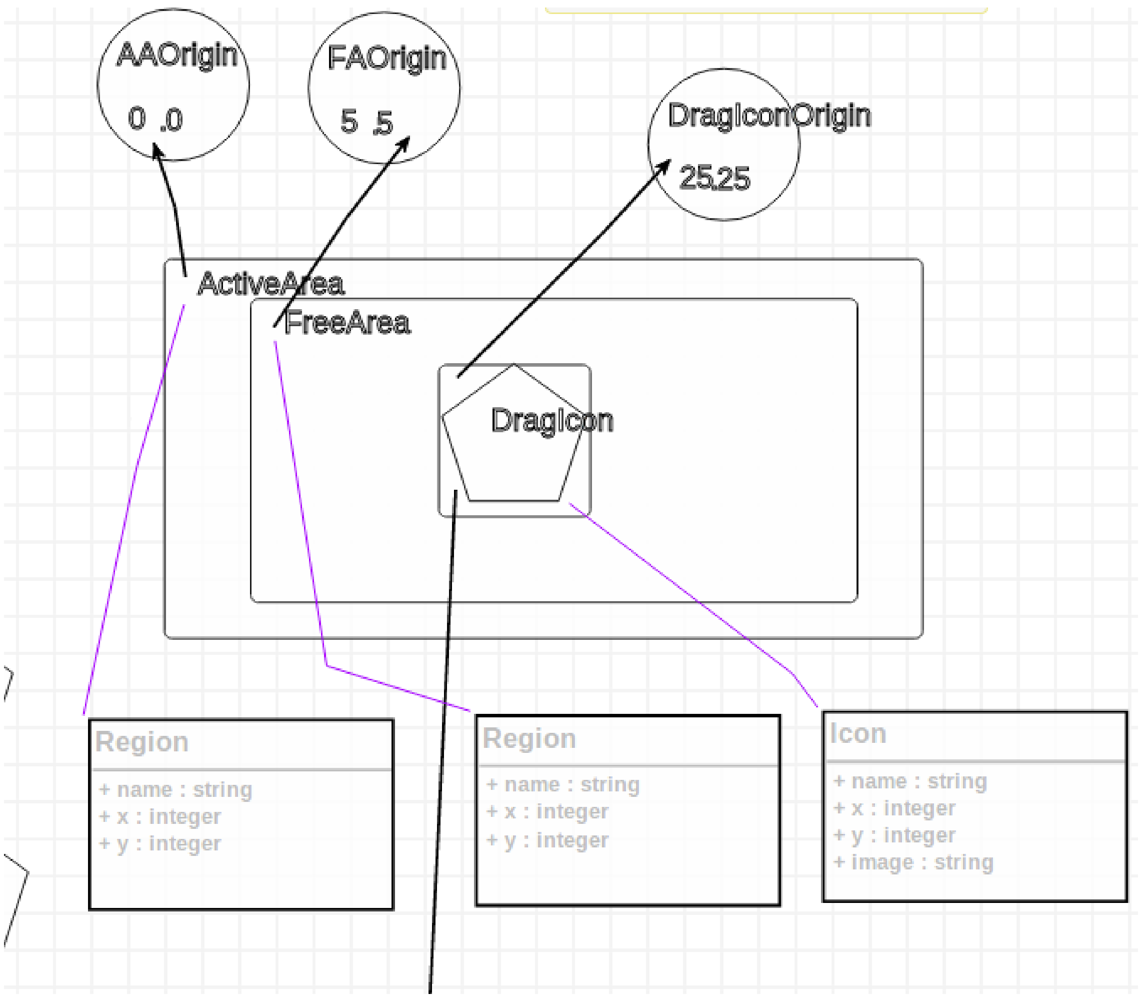

Figure 9: Application of the first transformation rule on our example model

The second step is converting the behaviour of IOGObjects to a state chart representing the same behaviour. We have done this for the highest level, generating a composite state which will hold the state chart. See figure 10. The lower level transformations have not been implemented. However, we will give a theoretical approach to this problem here. There are quite a few challenges. Some constructs have a direct counterpart in state charts and should not require much adaption. These constructs are regular states, history states and metastates.

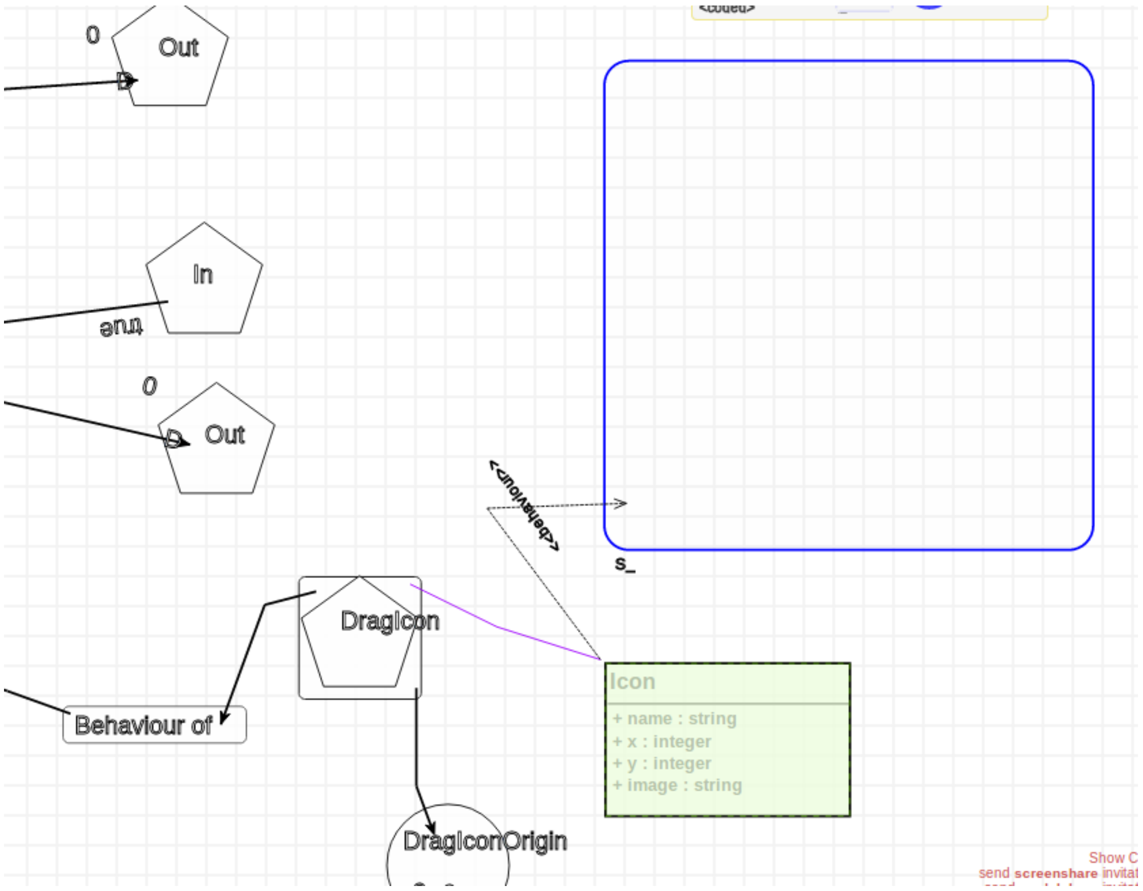

Figure 10: Application of a second rule creating room for the state chart

In state charts transitions have a label of the form event[guard]/action. We will use the semantics of this transition in our mapping. Display states from IOG can be replaced by regular states which have an incoming transition where the event and guard can be derived from the firing condition in IOG. For example, a user input would correspond to an event, where a boolean expression would correspond to a guard. The action should then then be a method call on its class which updates the visual appearance.

Our DataObjects will have to be removed completely. State charts does not support this construct. Each DataObject should get an attribute in its corresponding class. Transition which read or write to a data store will have to be modified by specifying the correct method call in their action. From read operations this is a getAttribute() call, for writing operation this is a setAttribute(data value / expression) operation. The corresponding data value or expression is passed in write operations. This is taken from the incoming data arc. In some cases, a new state may have to be created if no other transitions exist which have the same event/guard signature or have room for another action.

External ports are something which we have not really implemented other than their class existence. If these ports are linked to different IOGObjects, these read/write operations become method calls on other classes, similar to how it works for data objects.

5. Future work

To improve the usefulness of our implementation, the mapping from IOG to SCCD should be completed. As was said in the previous section this part of the implementation was not completely finished. We have given a theoretical overview of how we would execute the rest of this model transformation. Once we have a correct formal mapping and can convert to SCCD without friction, this gives us the ability to transform from SCCD to other modelling languages such as code. This could be by exporting the model to metadepth. From this textual modelling language we could generate python code which uses a graphical user interface library. This would enable drawing on the screen of a specified widget. The existence of a tool like this has been requested in Carr et al. (1994) as it something that was still lacking from their implementation. It would allow for rapid prototyping and testing of new widgets and widget specifications, which increases its use as a modelling language.

6. Conclusions

This paper gives an overview of the Interaction Object Graph formalism and how we can use it to specify graphical widgets. We've shown our way of implementing this formalism by going through the usual steps of language design. We have discussed its meta-model and concrete syntax in the next section, Through this language we are now able to create valid IOG models, which is a major milestone.

We then went on to explain how we would perform a model transformation from IOG to SCCD. While this mapping was not fully implemented, we have suggested how one could solve the different problems that arise from translating constructs from one formalism to the other. A powerful semantic from SCCD are the labels on transitions, which are used perform actions which update data attributes is their corresponding classes. This can be used to replace the date stores which have no representation in the state charts formalism.

While we have not delivered on everything we wanted to do, we have set the first steps in making the IOG formalism useable for modern widget design. As explained in future work, some model transformations need to happen to be able to really reap the benefits of this formalism. Specifically, mapping onto a library for graphical user interfaces which can draw what is specified in our models on the screen has great value for user interface designers. These would allow domain experts to use a tool like this instead of using code to specify their widgets. Overall, we are happy with the work we have done and have certainly learned a lot, although we think it is a bit of shame that the full model transformation has not been completely implemented.

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.