IRIS VISUALIZATION

This document describes the results and development process of the *Iris Visualization*, an example of a multi coordinate visualization that can be described by a State Chart.

Introduction

A *multi coordinate view* can be an example of how different widgets can be used within the same visualization. In this particular case, we can recognize three different visual components:

1. *a scatter plot*, where the user can perform filtering by zoom in / zoom out and panning interactions
2. *a bar chart*, where the user can perform filtering by selecting a particular bin;
3. *a range slider*, where the user can perform a filter by dragging the handlers or the corresponding selected region.

Each visual component corresponds to a particular widget within the finite state machine.

State Chart

To translate the visualization into a *Finite State Machine* (*FSM*), we have decided to use a top-down approach, where each visual component is made by several distinct widgets, i.e. a set of operations and states that represents a particular interaction of the user with the system. In particular, we can recognize the following widgets for each visual components:

- ***range slider:***
    - *drag*: this widget defines when a user alters the underlying region of the slider, by two different entrypoints:
        - *mousedown*: when a user selects one of the two handles or the region between them and performs a *mousemove event,* until the subsequent *mouseup event;*
        - *doubleclick (dblclick):* when a user performs this event on a position of the slider, the nearest handler will be updated with that particular position.

      A variable containing the position of the handlers should be stored to help the FSM reasoning about the direction of the user's movements (left or right)

- ***scatter plot:***
    - *zoom*: the zoom widget gives the opportunity, after a wheel event raised by the user, to increase or decrease the plot's area. A variable containing the zoom percentage should be stored, to help the FSM reasoning about the operation (zoom-in/zoom-out);
    - *panning*: the panning widget gives the opportunity, after a *mousedown event*, to drag the domain of both axes with a *mousemove event*, until the *mouseup*

event is raised. A variable containing those domains should be stored to help the FSM reasoning about the panning direction (up/down/left/right).

- ***bar chart:***
    - *bin hovered:* this widget recognizes when a user enters with a *mouseover event* the region of a particular bin, until the subsequent *mouseout event*. A variable containing the identifier of the current bin hovered should be stored;
    - bin click: this widget recognizes when a user clicks a particular bin to select/deselect it. A variable containing the identifier of the selected bin should be stored, to help the FSM reasoning about the state of that particular bin, i.e. if that is selected or not.

Also, each visualization has as entry point the *hover widget,* which recognizes when a user performs a *mouseover* on a particular visual component: a variable containing the identifier of the visual component should be stored, to help the FSM reasoning about the current visual component that is hovered by the user.

Design

We can use different *Hierarchical State Machines* (*HFSM*) to define separately each component: this approach is useful since it can model transitions within a particular widget without exposing them to the others, and also marking the particular transitions that are necessary to reach and subsequently leave a particular widget. This separation comes is applied at different stages, in particular:

- each visual component is represented by an HFSM;
- each widget is represented by an HFSM.

In this way, each state will have the following naming convention (except the initial state, "*rest"*):

visualComponentName widgetName stateName

The states within each widget are connected with guarded transition, where particular conditions should be satisfied to perform the transition correctly. When those transitions are made, the context, i.e. the model related to the state charts, is updated with the new values. One particular widget that differs from the other ones is the *panning widget,* that exploits the concepts of *parallel state nodes*, i.e. orthogonal states that can be reached concurrently: the needs for this particular representation was necessary to express the situations when a user performs panning with respect to different directions among the two different axes (e.g. a diagonal panning to the top right corner of the scatter will be represented by two different states, up and right).

Implementation

This section describes the implementation choices of the application. We omit the details of the realization of the visual components, which are realized with *Vega*, a visual grammar language that gives the opportunity to make a visual environment with a JSON file filled with different specification to create, transform and visualize the data.

XState

XState is a library that can build an *SCXML model* of an automaton: using an integrated data model, along with guarded/conditional transitions, it is possible to check within the FSM when we can enter/exit different states which require a particular condition to be satisfied. (e.g. to reach the "right" state on the range slider, it's necessary that the new value dragged by the user with one of the handlers is greater than the previous value).
Also, with the *XState Inspection Tool*, it's possible to visualize in real-time the behaviour of the *FSM* attached to the page.

Dashboard

To give the opportunity to see the FSM information inside the visual environment, without recurring an external source, a dashboard containing the following information was implemented:

- the current state of the *FSM*;
- the last transition raised by the user, with the data attached to it and the timestamp of its execution;
- the states that can be reached from all the available transitions for the current state. For each state, a *predictive step* is performed to show also the next states along with the required transition to reach them. Each transition contains information about the condition (if any) necessary to execute it and the latency measured by the *Performance Observer API*, i.e. the time between the execution of the transition and the time when the FSM actually recognizes the transition.

Development

This section describes all the different iterations of the application

First Implementation

In the *first implementation* (*I1*) of the visualization (Fig. 1), realized with *D3js*, each time an *HSFM* triggers the event "*mouseout*", it returns to the initial state "*rest*". When a transition "*mouseover*" occurs, along with them a string containing the hovered visualization will be passed to the data model. "*mouseover*" is implemented as a self-transition, so when the state is re-entered different conditions upon the current visualization hovered are checked: If one of them is satisfied, then an eventless-transition will reach the currently hovered visualization.
One issue raised with this implementation is when different widget interacts concurrently. This behaviour can be seen clearly during a "*mousemove*" event: in fact, the user can move the mouse freely around the visualization, even when some object is panned/dragged, and trigger a "*mouseup*" transition inside all the other widgets. For this reason, we've added a "*mouseup*" transition event inside all the widgets, which updates the context of the FSM.

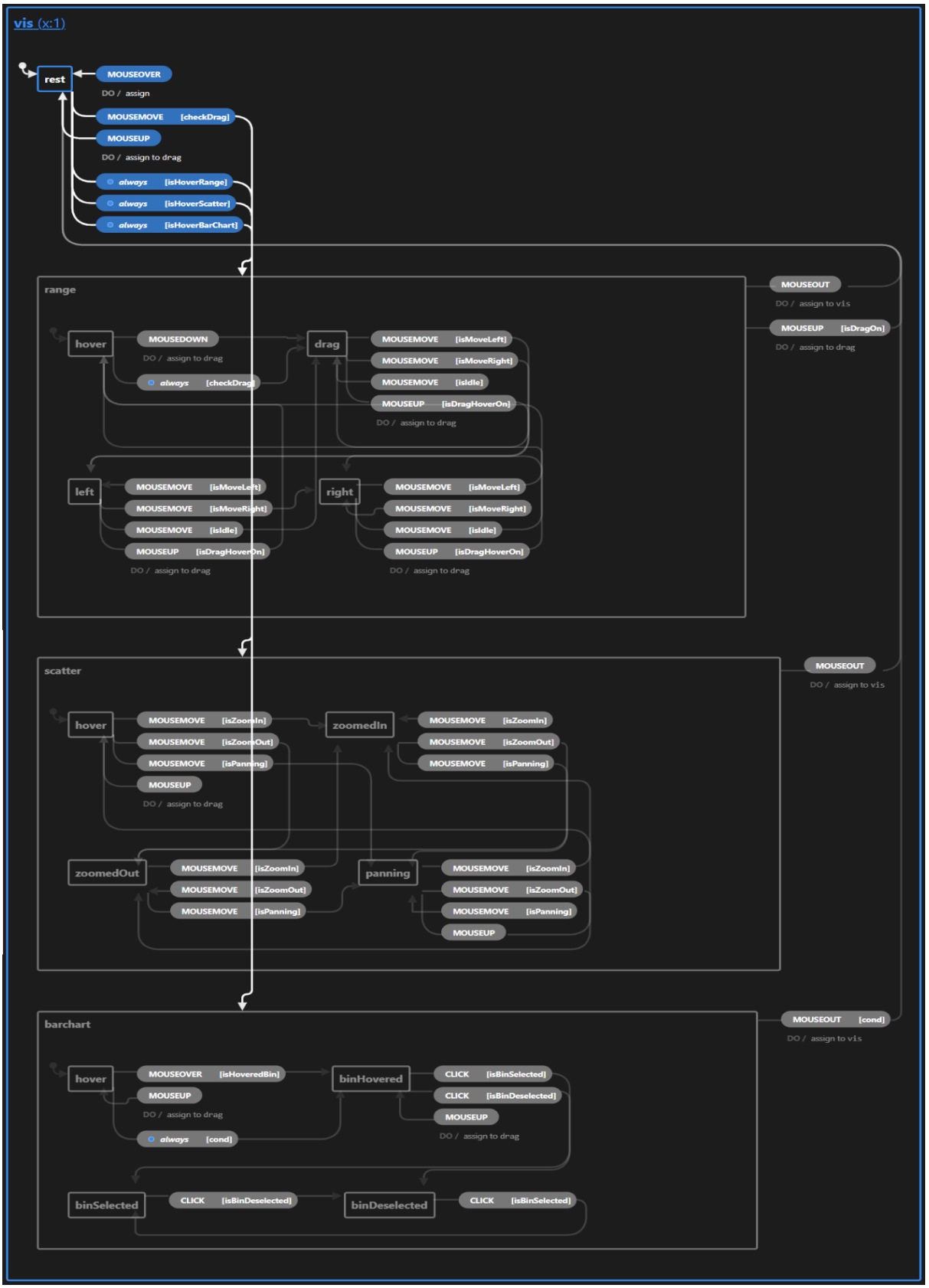

Figure 1: First implementation (*I1*) of the FSM

Second Implementation

While the I1 captures correctly the behaviour of the user, it lacks in showing precisely what a widget should do. In fact, the "*mouseup*" transition is not an authorized transition for many states, such as "*hover*", but it was added to capture some events that may be raised by the users. These events, however, are strictly related to web browser implementations, thus they should not be captured by the *FSM* and ignored, i.e. a "*mouseup*" transition made by the range slider *HSFM* should be seen only within the range slider *HSFM* itself. Also, the drag flag used in the context of *I1* has been removed from the context since the transitions now are filtered based on the current state reached by the user. (Fig.3)

For this reason, the *second implementation* (*I2*) adds a new section (Fig. 2) on top of the visualization which acts as an inspector of the *FSM* attached to the page (see Dashboard Section) by showing the information described in *Fig.2.* This particular version does not contain any "prediction step", besides it only focuses on the states related to the available transitions of the current state.

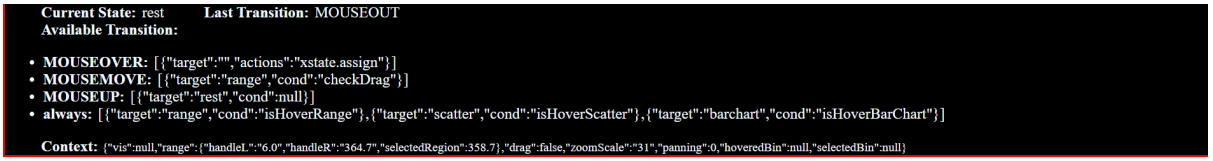

Figure 2: I2 section showing FSM information

Regarding the visualization, *I2* introduces a new version of the visualization written in *Vega*: using this new approach, it was possible to better define in the *FSM* some interactions realized by the user: in particular, inside the scatter plot, the *panning*/*zoom* states can be reached by *wheel*/*mousedown* events respectively.

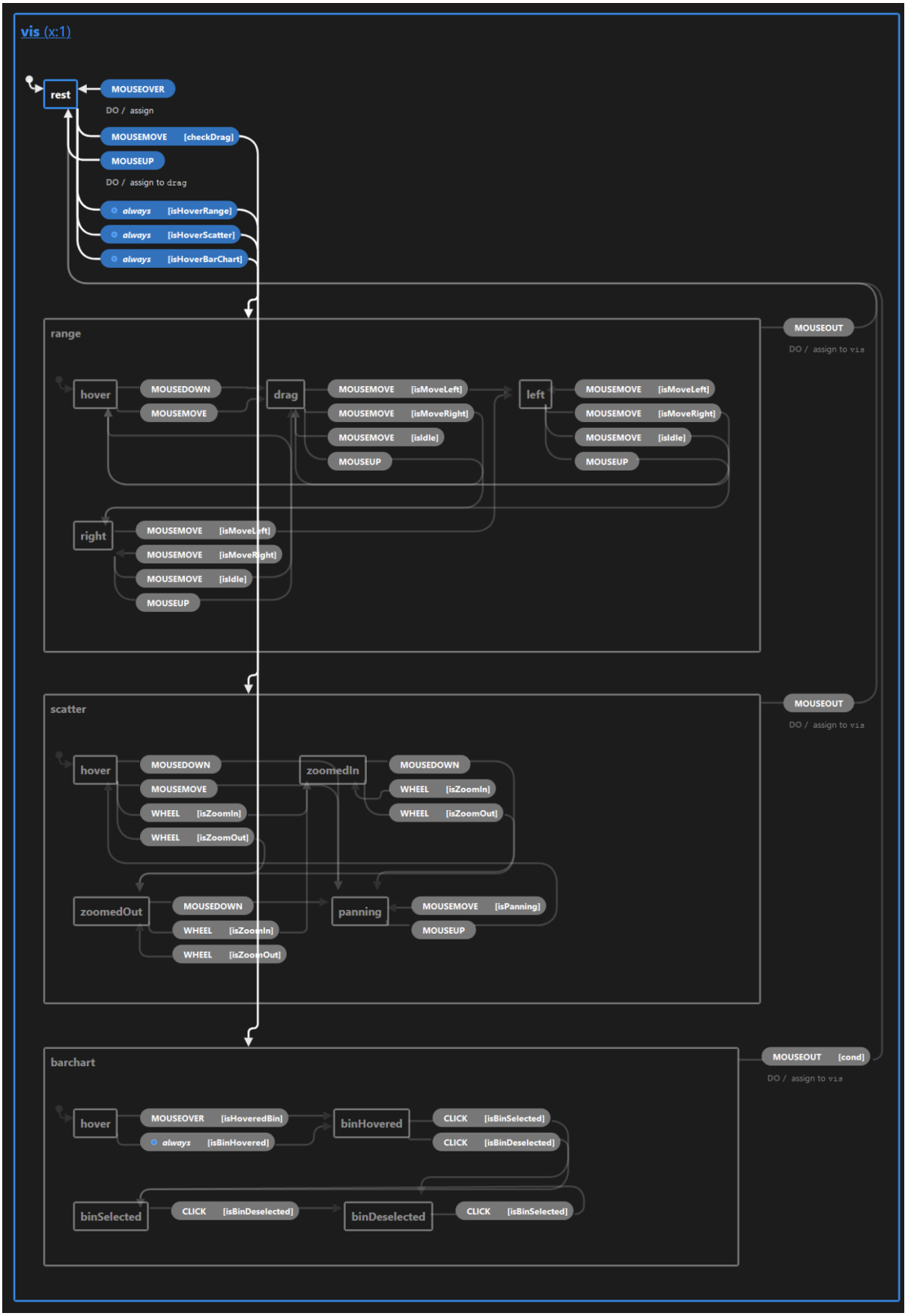

Figure 3: Second Implementation (*I2*) of the FSM

Third Implementation

This implementation (I3) adds the predictive step to the dashboard of I2 and information about transition latency (see Dashboard Section). Also, the XState model (that can be seen here) has been updated to better represent the design choices described in the *State Chart Section*: the widget separation has been added to group several states related to a particular interaction, also the “double click” feature for the range slider has been added.
Regarding the “mousemove issue” of *I1* and *I2*, *I3* does not raise any event besides the *mouseup* during a *mousemove interaction*: If a user, during this particular interaction, hovers a different visualization other than the one where the *drag / panning* is currently performed, and then performs a *“mouseup”,* the FSM performs all the missing transitions occurred during this period of time, i.e. “*mouseout*” and “*mouseover*”, to reach correctly the *hover state* of the visualization currently hovered by the mouse. To do so, *mousemove* should pass along its data information about the hovered visualization.

Demo

A video demonstration of the visualization environment and the corresponding XState model can be seen here: the video show, for each visual component, the corresponding widget and all the different states that can be reached. In particular, the XState Inspector shows (almost) in real-time the FSM state updates made by the user interaction.