ATUAV Experimenter Platform – User Manual

2019

Cristina Conati, Grigorii Guz, Sébastien Lallé, Willis Peng, Dereck Toker

Department of Computer Science, University of British Columbia, Vancouver, BC, Canada

https://github.com/ATUAV/ATUAV_Experimenter_Platform/

CONTENTS

1 GENERAL INFORMATION

Experimenter platform is the environment for designing and evaluating eye-tracking-based User-Adaptive Interfaces (UAI)**,** i.e., interfaces that can adapt in real-time to their users based on their gaze behaviors. Such adaptation can include, for instance, changing the layout of the interfaces, enabling/disabling functionalities, highlighting relevant parts of the interface, adding visual cues… The platform contains the essential functionalities to support *experimenters* (e.g., researchers, companies) in implementing such gaze-based adaptive systems, and test them via a user study. For now the platform supports Tobii Pro eye-tracker and web-based interface, but other eye tracker can be plugged in.

The Platform's source code can be found here:
https://github.com/ATUAV/ATUAV_Experimenter_Platform

1.1 Common definitions

The platform supports user studies with multiple ***tasks*** (i.e., repeated measures experiments) with a web-based interface. For each task, experimenters can define ***adaptation rules***, which are sets of conditions (also referred to as *events*) that need to be satisfied in order to trigger an ***adaptive intervention***.

The *adaptation rules* can trigger interventions based on the following three types of *user events*:

1. Based on the participant's <u>fixations</u> (defined as gaze maintained at one point on the screen) detected by the platform. For example the platform could highlight the legend of a bar chart when the participant fixates on the chart). Fixations are tracked over the entire screen, or within specific regions of the screen called Areas of Interests (AOIs, see below)
2. Based on <u>eye tracking metrics</u> computed by the platform at regular intervals (e.g., every 5 seconds) that informs about the participant's gaze behaviors. Eye tracking metrics are summary statistics (e.g., mean, std.dev, min, max) computed over a user's fixations as well as over her pupil size and the distance of her head to the screen. A comprehensive list of the available features are provided in the technical documentation.
3. Based on a <u>user model</u> provided by the experimenter and plugged in the platform. The user model can be populated into the platform database either beforehand (e.g., by collecting user traits via pre-questionnaires or tests) or during interaction with the platform using a computerized user model (e.g., a machine learning model predicting relevant information about the participant such as her expertise, personality, level of cognitive abilities…).

It is possible to use *Areas of Interest (AOI)* to specify where events of interests (fixations, eye tracking metrics) should be detected on the screen in order to trigger an intervention. The AOIs are specified by the experimenter as an array of tuples of x-y coordinates (See Section 3.2). For example, an AOI can be specified to capture solely gaze events made over a visualization displayed on the screen.

More details on how to define the *rules* and *interventions* will be provided in this guide (Section 3).

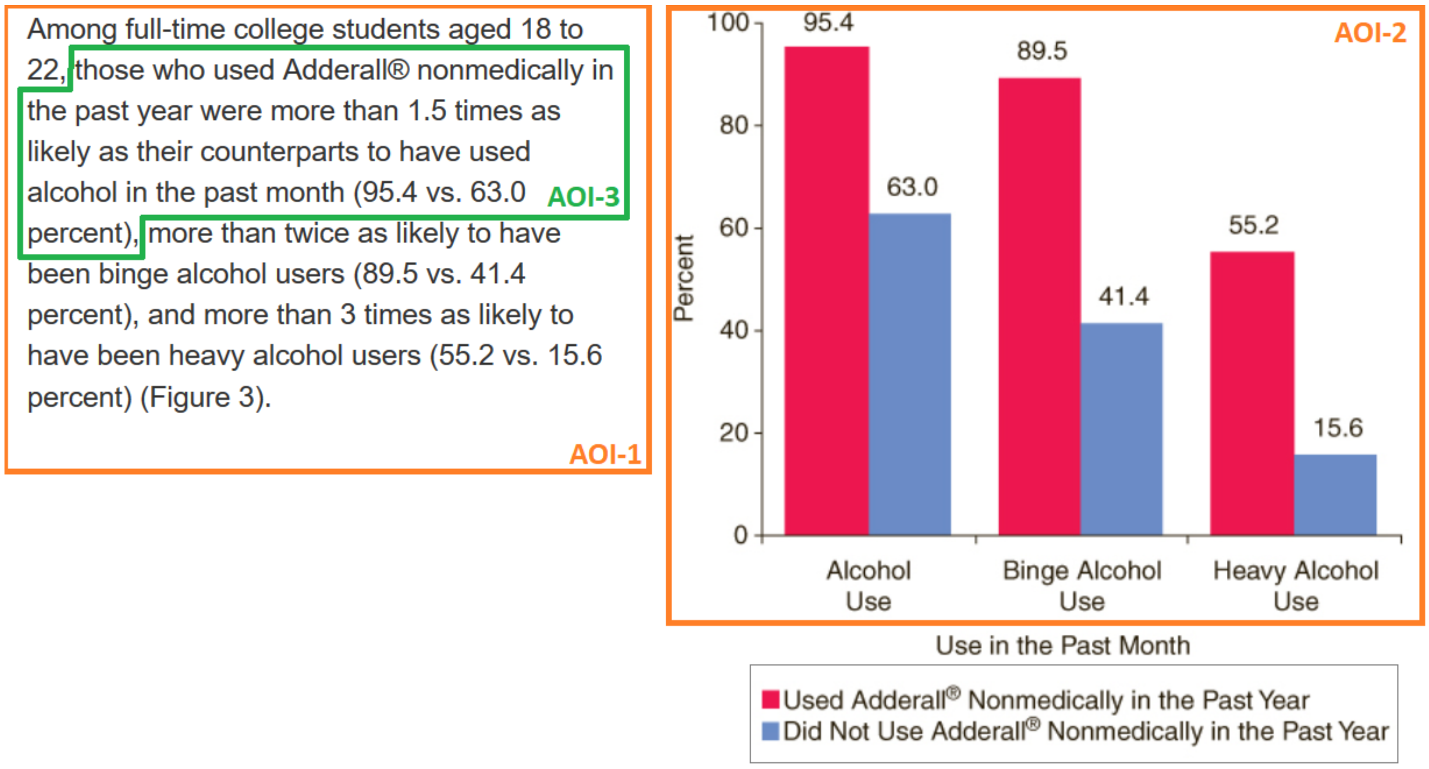
1.2 Sample Application

The platform comes with a sample application that demonstrates the delivery of adaptive interventions to a visualization system. The visualization system consists of a text (left on Fig. 1) describing a bar chart (right on Fig. 1). The following adaptive interventions are implemented in the sample application:

1. Highlight the legend of the bar graph if a fixation on the text (AOI_1) followed by one fixation on the chart (AOI-2) is detected.

2. Highlight the bars referred to by the text in AOI-3 when two fixations on AOI-3 are detected (i.e., highlight the two bars on the left in the chart labelled “Alcohol Use”).

We will use this sample application as a case study in this user guide to explain how to set-up and run an experiment.



Among full-time college students aged 18 to 22, those who used Adderall® nonmedically in the past year were more than 1.5 times as likely as their counterparts to have used alcohol in the past month (95.4 vs. 63.0 percent), more than twice as likely to have been binge alcohol users (89.5 vs. 41.4 percent), and more than 3 times as likely to have been heavy alcohol users (55.2 vs. 15.6 percent) (Figure 3).

Fig. 1. Sample adaptive visualization provided with the platform.

2 PLATFORM SETUP

2.1 Overview of Platform Architecture

The source code of the platform include (see diagram below):

- A ***back-end*** (*/application/backend/*) responsible for fetching the data from Tobii eye tracker, as well as detecting user fixations and generating eye tracking metrics at regular intervals (specified by the experimenter). The back-end can also maintain a user model provided by the experimenter.
- A ***middle-end*** (*/application/middleend/*) that queries and updates the database, as well as evaluates the adaptation rules defined by the experimenter to trigger graphical interventions.
- A ***front-end*** (*/application/frontend/*) that include the experimenter’s visualization system. The platform provides helpers to enable communication between the *front-end* and the other components of the platform.
- ***Databases*** (*/database/*) that stores the rules, interventions, eye tracking data, AOIs and tasks.
- A ***Controller*** (*/application/application_state_controller.py*) that initiates the platform and enables communication among the above components.

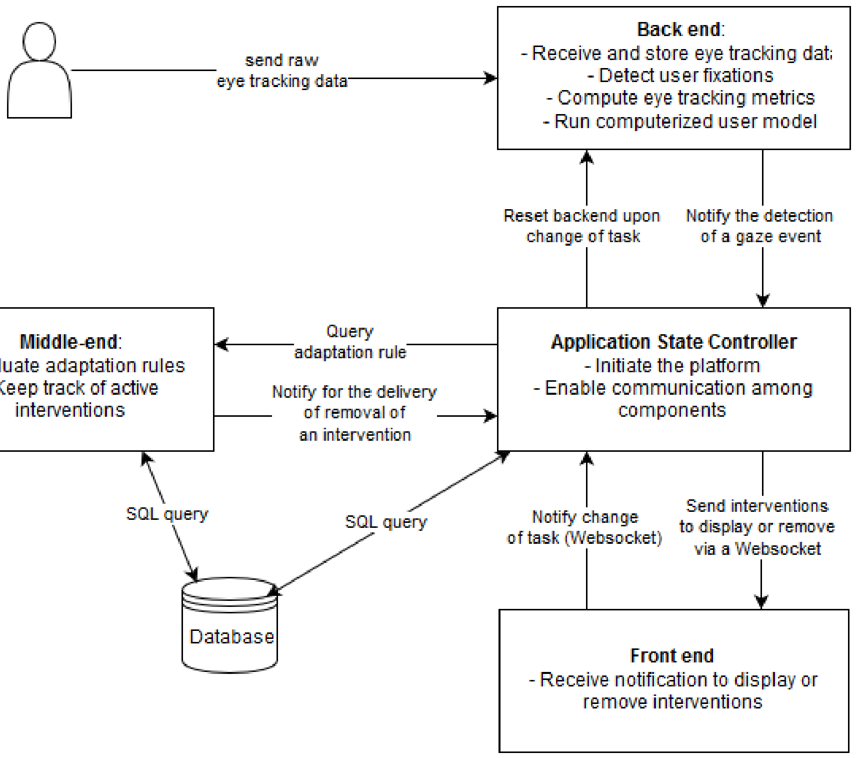

send raw eye tracking data
Back end:
- Receive and store eye tracking dat
- Detect user fixations
- Compute eye tracking metrics
- Run computerized user model
Reset backend upon change of task
Notify the detection of a gaze event
Middle-end:
- Evaluate adaptation rules
- Keep track of active interventions
Query adaptation rule
Notify for the delivery of removal of an intervention
Application State Controller
- Initiate the platform
- Enable communication among components
SQL query
SQL query
Notify change of task (Websocket)
Send interventions to display or remove via a Websocket
Database
Front end
- Receive notification to display or remove interventions

Diagram of the main components of the platform and their interaction.

2.2 Installing the platform

The platform runs on Windows OS and Python 2.7.x (32-bit version). The platform uses Tornado Framework v. 5.1 as its main engine, SQLite 3 (available as a standard Python library) for storing the data, and also requires Numpy 1.15.1 or above. The platform includes a lightweight version of EMDAT, a library developed at the University of British Columbia for computing eye tracking metrics from data generated by an eye tracker.

To run the platform, you'll first need to install Tornado and Numpy:

```
Pip install tornado
Pip install numpy
```

The SQLite database is built in Python standard library, so there is no additional steps for setting-up SQLite beyond installing Python 2.7.x. However, to simplify the interactions with the database (adding new rules, specifying events etc.), we recommend installing DB Browser for SQLite.

The Tobii Analytics SDK 3.0.83 (Python version) required for the platform to connect to your eye tracker is provided with the platform in *./Modules/* (other SDKs can be added, cf. technical documentation).
In order to start the platform, an eye tracker must be connected to your PC. The platform will automatically attempt to detect an active eye tracker on your system, and will throw an exception if none is found.

Finally, you need to retrieve the source code from the Github and copy it in a local directory.

3 EXPERIMENT SETUP

3.1 Configuring Runtime Parameters

In *params.py*, you can specify which components you want to use for your experiment. The description for the parameters is as follows:

- Boolean constants for specifying which types of user events should be tracked by the platform:
  - USE_FIXATION_ALGORITHM - True if fixation detector should be active
  - USE_EMDAT - True if EMDAT should be periodically called to generate statistical eye gaze metrics
  - USE_ML - True if a computerized user model (provided by the experimenter) should be periodically called by the platform.

- Integer constants (in milliseconds) specifying the period for EMDAT and User Model calls:
  - EMDAT_CALL_PERIOD - specifies how much time should be passed between successive EMDAT calls
  - ML_CALL_PERIOD - specifies how much time should be passed between successive user model calls

- Integer constants for the Fixation Detection component:
  - FIX_MINDUR (in milliseconds) - the lower bound for duration of a fixation. Default is 100ms.
  - FIX_MAXDIST (in pixels) - the largest displacement of the first. Default is 35pix.

- Boolean constants for specifying which EMDAT features should be computed:
  - USE_PUPIL_FEATURES
  - USE_DISTANCE_FEATURES
  - USE_FIXATION_PATH_FEATURES
  - USE_TRANSITION_AOI_FEATURES

- Boolean constants for specifying whether and how EMDAT features should be aggregated over time:
  - KEEP_TASK_FEATURES - if True, EMDAT will, after each call, merge the eye tracking metrics with the features computed previously during this task for this user.
  - KEEP_GLOBAL_FEATURES - if True, EMDAT will, after each call, merge the tracking metrics with with the features computed previously during the whole execution of the platform (all previous tasks and the current task) for this user.
  - If both KEEP_TASK_FEATURES and KEEP_GLOBAL_FEATURES are False, EMDAT will simply compute from scratch the eye tracking metrics using solely eye tracking data observed since the last EMDAT call.

3.2 Setting Up The User Model State Database

./database/user_model_state.db includes the definition of the tables storing information about the tasks, the AOIs and relevant gaze events. These tables must be populated by the experimenter for each new experiment, by directly modifying *user_model_state.db*. You can modify the database using any SQLite management software (such as https://sqlitebrowser.org/).

Here is the schema for the database (cf. ./database/user_model_state_schema.db.sql). Note that the database structure is fully explained in the technical specification under Application State Controller > User Model State.

```
BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS `user_state_task` (
   `event_name`   TEXT,
   `task`   INTEGER,
   PRIMARY KEY(`event_name`,`task`)
);
CREATE TABLE IF NOT EXISTS `user_state` (
   `event_name`   TEXT NOT NULL,
   `type`   TEXT,
   `aoi`   TEXT,
   `feature`   TEXT,
   PRIMARY KEY(`event_name`)
);
CREATE TABLE IF NOT EXISTS `aoi` (
   `name`   TEXT NOT NULL,
   `task`   INTEGER NOT NULL,
   `dynamic`   INTEGER,        // currently not supported
   `polygon`   BLOB,
   PRIMARY KEY(`name`,`task`)
);
COMMIT;
```

Let's say we want to set up an experiment with 3 tasks, using the sample visualization described in Section 1.2. In each task we are tracking the fixation events of two AOI's as well as the mean pupil size.

Let's first define two AOI's for all 3 tasks. They can be the same polygon across tasks or vary based on the task:

```
//text AOI (AOI-1 in Fig. 1)
    INSERT INTO aoi VALUES("text", 1, NULL, "[(132,229),(132,555),(606,555),(606,229)]");
    INSERT INTO aoi VALUES("text", 2, NULL, "[(132,229),(132,555),(606,555),(606,229)]");
    INSERT INTO aoi VALUES("text", 3, NULL, "[(132,229),(132,555),(606,555),(606,229)]");

    //Visualization AOI (AOI-2 in Fig. 1)
    INSERT INTO aoi VALUES("vis", 1, NULL, "[(616,94),(616,509),(1090,509),(1090,94)]");
    INSERT INTO aoi VALUES("vis", 2, NULL, "[(616,94),(616,509),(1090,509),(1090,94)]");
    INSERT INTO aoi VALUES("vis", 3, NULL, "[(616,94),(616,509),(1090,509),(1090,94)]");

    //Textual reference AOI (AOI-3 in Fig. 1)
    INSERT INTO aoi VALUES("ref_1", 1, NULL,
    "[(244,233),(244,253),(578,253),(244,233),(578,233),(152,258),(152,279),(196,279),(196,258),(578,233)]");
    INSERT INTO aoi VALUES("ref_1", 2, NULL,
    "[(244,233),(244,253),(578,253),(244,233),(578,233),(152,258),(152,279),(196,279),(196,258),(578,233)]");
    INSERT INTO aoi VALUES("ref_1", 3, NULL,
    "[(244,233),(244,253),(578,253),(244,233),(578,233),(152,258),(152,279),(196,279),(196,258),(578,233)]");
```

Let's now define three *user state events* that use these three AOI's:

State 1: INSERT INTO user_state VALUES("text_fix", "fix", "text", NULL);   //fixation detected on the text AOI
State 2: INSERT INTO user_state VALUES("vis_fix", "fix", "vis", NULL); //fixation detected on the vis AOI
State 3: INSERT INTO user_state VALUES ('ref_1_fix','fix','ref_1',NULL);   //fixation detected on the ref_1 AOI

//Note: it is possible to track fixations and EMDAT metrics over the entire interface, by defining an AOI that encompasses the interface. Same thing for the entire screen. For example for a 1920 x 1080 screen, the full-screen AOI boundaries are: “[(0,0), (1920,0), (1920,1080),(0,1080)]”.

The sample application includes only user events based on the user’s fixation. As said in Section 1.1, it is also possible to define user events based on eye tracking metrics or a user model. For example:

```
// user event based on the user’s average pupil size on the vis AOI
        INSERT INTO user_state VALUES(“avg_pupil_feature”, “emdat”, “vis”, “meanpupilsize”);

        // user event based on the user’s reading proficiency as provided by a user model
        INSERT INTO user_state VALUES(“reading_proficiency”, “ml”, NULL, NULL);
```

We must also add the three user states define above to the user state task table to indicate which events is to be tracked for each given task:

```
INSERT INTO user_state_task VALUES(“text_fix”, 1);
INSERT INTO user_state_task VALUES(“text_fix”, 2);
        INSERT INTO user_state_task VALUES(“text_fix”, 3);
INSERT INTO user_state_task VALUES(“vis_fix”, 1);
        INSERT INTO user_state_task VALUES(“vis_fix”, 2);
INSERT INTO user_state_task VALUES(“vis_fix”, 3);
        INSERT INTO user_state_task VALUES(“'ref_1_fix'”, 1);
INSERT INTO user_state_task VALUES(“'ref_1_fix'”, 2);
        INSERT INTO user_state_task VALUES(“'ref_1_fix'”, 3);
```

Now we have finished setting up the event and AOIs definitions for our test experiment

3.3 Setting Up The Gaze Event Rules Database

The adaptation rules and intervention must be added in the ./database/gaze_event_rules.db database. The schema of this database is as follow (please refer to the technical specification about the database structure under Middle End > Gaze Event Rules for more information):

```
BEGIN TRANSACTION;
CREATE TABLE "rule_task" (
   `rule_name`    TEXT NOT NULL,
   `task`    INTEGER
);
CREATE TABLE "rule" (
   `name`    TEXT NOT NULL,
   `delivery_sql_condition`    BLOB,
   `removal_sql_condition`    BLOB,
   `max_repeat`    INTEGER,
   `active_retrigger`    INTEGER,
   PRIMARY KEY(`name`)
);
CREATE TABLE "rule_delivery_trigger" (
   `rule_name`    TEXT NOT NULL,
   `delivery_trigger_event`    TEXT
);
CREATE TABLE "rule_removal_trigger" (
   `rule_name`    TEXT NOT NULL,
   `removal_trigger_event`    TEXT
);
CREATE TABLE "rule_intervention_payload" (
   `rule_name` TEXT NOT NULL,
   `intervention_name` TEXT
);
CREATE TABLE "intervention" (
   `name`    TEXT NOT NULL,
   `max_repeat`    INTEGER,
   `function`    BLOB NOT NULL,
   `arguments`    BLOB,
   `delivery_delay`    INTEGER,
   `transition_in`    INTEGER,
   `transition_out`    INTEGER,
   PRIMARY KEY(`name`)
);
COMMIT;
```

Where:

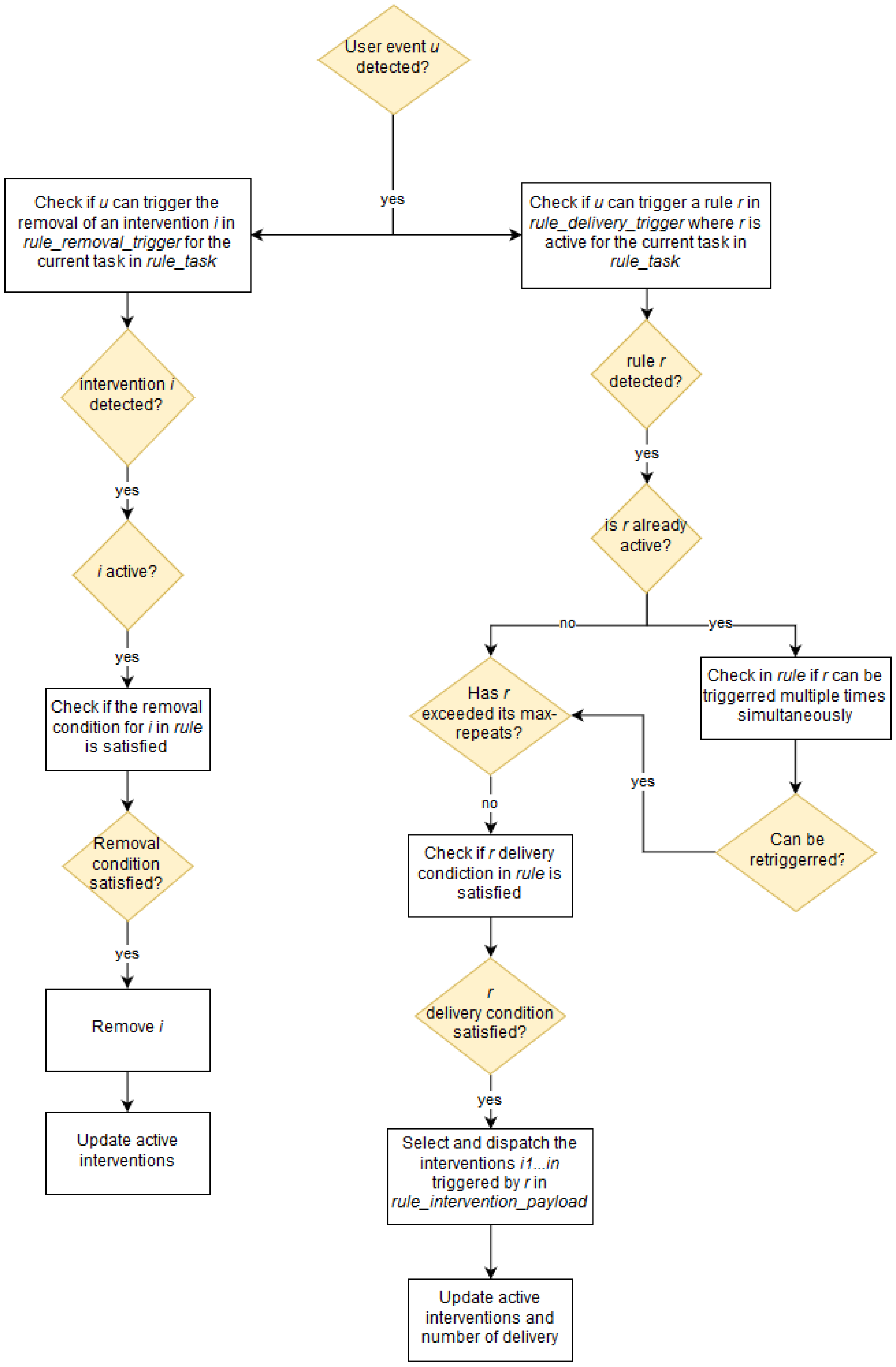
- *rule_task* includes the active rules in each task (i.e., it maps each task to a set of rules)
- *rule_delivery_trigger* and *rule_removal_trigger* include the user events that can trigger the delivery and removal (respectively) of an intervention. User events can include fixations, EMDAT features and user model features, as described in Section 1.1.
- *rule* contains the conditional statements that control for the delivery and removal of an intervention. These statements are written in the form of a SQL query (see examples and explanations below).
- *rule_intervention_payload* maps each rule to the interventions (specified in the *intervention* table) it can dispatch. Each intervention can be mapped to several rules, and vice-versa.
- *intervention* includes the basic information necessary for the front-end to display the intervention (in particular what JS function to call).

The platform also generates two tables to keep track of the rules and interventions that have been triggered so far. These tables can be used by the experimenter to define delivery rules (for instance when an intervention *i2* should be delivered only if another intervention *i1* has been previously delivered, or is currently active). The format of these tables are:

```
CREATE TABLE "intervention_state" (
    `intervention`    TEXT NOT NULL,
    `active    `    INTEGER,
    `time_stamp `    INTEGER,
    `occurences`    INTEGER,
    PRIMARY KEY(`intervention`)
);

CREATE TABLE "rule_state " (
    `rule`    TEXT NOT NULL,
    `time_stamp `    INTEGER,
    `occurences`    INTEGER,
    PRIMARY KEY(`rule`)
);
```

Overall an adaptive intervention can be delivered solely if the relevant triggers (user events) are detected AND the corresponding rules' SQL delivery condition is satisfied. The diagram below summarizes the workflow for the delivery and removal of the interventions (this diagram is implemented in *./application/middleend/adaptation_loop.py*, see technical documentation):


User event *u* detected?
yes
Check if *u* can trigger the removal of an intervention *i* in *rule_removal_trigger* for the current task in *rule_task*
intervention *i* detected?
yes
i active?
yes
Check if the removal condition for *i* in *rule* is satisfied
Removal condition satisfied?
yes
Remove *i*
Update active interventions
Check if *u* can trigger a rule *r* in *rule_delivery_trigger* where *r* is active for the current task in *rule_task*
rule *r* detected?
yes
is *r* already active?
no
yes
Check in *rule* if *r* can be triggerred multiple times simultaneously
Can be retriggerred?
yes
Has *r* exceeded its max-repeats?
no
Check if *r* delivery condition in *rule* is satisfied
r delivery condition satisfied?
yes
Select and dispatch the interventions *i1...in* triggered by *r* in *rule_intervention_payload*
Update active interventions and number of delivery

We now explain step-by-step how to define rules and interventions in the database, using the sample application described above (Section 1.2) as an example.

Let’s first define the two interventions (I1 and I2) defined in section 1.2 for our sample application. For this example, let’s say we perform a legend highlight and a bar highlight (the first intervention is fully commented):

```
I1: INSERT INTO intervention VALUES(
        “legend_intervention”,                    // intervention name
        1,                       // maximum repeats
         “highlightLegend”,             // corresponding javascript function to be called from front-end
         ‘{"type": "legend", "color": "blue", "bold": true, "bold_thickness": 5, "desat": false, "arrow": false}’,
                                                                          // arguments supplied to js function
        0,                        // delay before intervention is delivered
        500,                        // time for the transition in
        500);                        // time for the transition out
```

//this is an intervention called “legend_highlight” which can be dispatched at most once, and when dispatched calls the “highlightLegend” function from the front-end, with the arguments given.

```
I2: INSERT INTO intervention VALUES(“bar_1_intervention”, NULL, “highlightVisOnly”,’{"type": "reference", "id": 1,
"bold": true, "bold_thickness": 3, "desat": true, "color": "green", "arrow": false}’,0, 1000, 1000);
```

//this is an intervention called “bar_highlight” with no limit to the number of dispatches, and when dispatched calls the “highlightVisOnly” function from the front-end, with the arguments given.

Let’s then define a set of rules to dispatch these interventions. The first rule (R1) says that there one fixation on the text (*text_fix* event) followed by one fixation on the chart (*vis_fix* event) is required to trigger the “legend_intervention”.

```
R1: INSERT INTO `rule` VALUES ('legend_rule',
        'Select
          case when count(*) > 0
            then 1
            else 0
          end result
        From
        (select * from text_fix TF, vis_fix VF
        where TF.time_start < VF.time_start        // the text_fix event should happens before the vis_fix event
        group by VF.id
        having count(TF.id) > 0);’ ,
        'select 1 as result;',   //removal condition, will be explain later in this section
        '',NULL);
```

You can notice that the SQL delivery rule for R1 query two tables called *text_fix* and *vis_fix*. These tables are dynamically created by the platform to store all fixations made within the corresponding text and vis AOI (respectively). More generally, the platform always create temporary tables for each *user event* defined in the *user_state* table, to log these events (see Section 8.3 for the definition of these tables). And the experimenter can query these temporary tables in the SQL delivery and removal rules.

The second rule (R2) says that two fixations on the ref_1 AOI (*ref_1_fix* event) are required to trigger the “bar_1_intervention”:

```
R2:  INSERT INTO `rule` VALUES ('ref_1_rule',
'Select
   case when count(*) > 1    // at least 2 fixations required
```

```
        then 1
        else 0
    end result
From
ref_1_fix RF,  rule_state
WHERE RF.time_end > rule_state.time_stamp and rule_state.rule = “'ref_1_rule'”;'   //retain only fixations on the ref_1 AOI since the last delivery of this rule, as this rule can be re-triggerred multiple time as set in I2 above
NULL, //no removal condition
,NULL,NULL);
```

The experimenter can also define rules based on eye-tracking metrics or user model values. For instance, here is a rule requiring the average pupil size of the user to be greater than 5:

```
INSERT INTO rule VALUES(“rule_pupil”,
‘Select case when count(*) > 0 then 1 else 0 end result from avg_pupil_feature where value >  5’, //delivery condition
‘‘Select case when count(*) > 0 then 1 else 0 end result from avg_pupil_feature where value <  5’,  //removal condition
NULL, NULL);
```

For more guidance on the sql_conditionals and **more examples of rules**, check the Example SQL Conditional (section 11.4 of the technical documentation).

Let’s map each intervention to its delivery rule in *rule_intervention_payload*:

```
INSERT INTO rule_intervention_payload VALUES(“'ref_1_rule”, “bar_1_intervention”);
INSERT INTO rule_intervention_payload VALUES(“'legend_rule'”, “legend_intervention”);

// note: one rule can dispatch multiple interventions by adding more entries in this table
```

Let’s define the delivery and removal trigger events for these two rules:

```
INSERT INTO rule_delivery_trigger VALUES(“'ref_1_rule”, “ref_1_fix”) //a fixation on the ref_1 AOI is necessary to trigger the rule
INSERT INTO rule_delivery_trigger VALUES(“'legend_rule'”, “vis_fix”) //a fixation on the vis AOI is necessary to trigger the rule
// note: more than one trigger can be specified for each rule.
```

Let’s add a removal trigger for R1 (*legend_rule*) specifying that the legend intervention will be removed if a ref_1_fix event is detected:

```
INSERT INTO rule_removal_trigger VALUES(“'legend_rule'”, “ref_1_fix”)

//Note: the removal condition for R1 was always true (“'select 1 as result;'”). This means that just the trigger is required to remove the intervention.
```

Let’s finally add these rules to be active for all 3 tasks:

```
INSERT INTO rule_task VALUES(“'ref_1_rule”, 1);
    INSERT INTO rule_task VALUES(“'ref_1_rule”, 2);
INSERT INTO rule_task VALUES(“'ref_1_rule”, 3);
    INSERT INTO rule_task VALUES(“'legend_rule'”, 1);
INSERT INTO rule_task VALUES(“'legend_rule'”, 2);
    INSERT INTO rule_task VALUES(“'legend_rule'”, 3);
```

Now we have finished defining all the rules and intervention definitions.

3.4 Setting up the backend

Next, you'll need to activate all of the detection components in *params.py* as specified in Section 3.1 of this guide. You'll also need to specify how often EMDAT (i.e., the component generated eye tracking metrics) and the computerized user model should be triggered. Specifically, to use EMDAT you need to set "USE_EMDAT = True" in *params.py*. If you want to provide a computerized user model, you need to set "USE_ML = True" in *params.py*. By default, both EMDAT and the user model are set to False and thus won't be used.

In *params.py*, you can set how often EMDAT and the user model should be called (see Section 3.1 above). Note that if you specify adaptation rules that are based on EMDAT or a computerized user model, the interventions can be delivered only after these detection components finish their execution. For example, if EMDAT is called every 20 seconds, adaptive interventions triggered by EMDAT's metrics can be dispatched at best every 20 seconds. Thus, if the specified detection component calling period is too high, there may be delays with intervention delivery.

Next, if you're going to use a computerized user model (optional), you need to add it in the platform and connect it with the back-end. Step by step instructions on how to accomplish that are provided in technical documentation for ML Component, Section 11.6. As noted there, in run() method, you'll need to store the prediction of your classifier in the dictionary self.predicted_features with the key "feature_name", where *feature_name* is the name you used to specify the event for this feature in user model. For example, if the feature you are trying to predict is called "reading_proficiency", the following queries can be used to specify the event beforehand:

```
INSERT INTO user_state VALUES("reading_proficiency", "ml", "", "");
```

and the task:

```
INSERT INTO user_state_task VALUES("reading_proficiency", 1);  //For Task 1
```

Then, during execution of the platform, the user model can simply update the value in as follow in *./application/backend/ml_component.py*:

```
 self.predicted_features['reading_proficiency'] = NEW_VALUE
//cf. Technical documentation for more details
```

3.5 Setting up the route handlers for user study workflow

3.5.1 Implementing the user study workflow

In order to specify the workflow of the study, we must write route handlers in the platform entry file. For the sample application, the entry file is *./experimenter_platform_study_1.py* (we recommend to duplicate and modify this file as needed for each new user study). We use the Tornado Web Framework to handle routing (http://www.tornadoweb.org/).

First we can define the application and its routing table:

```
class Application(tornado.web.Application):
    def __init__(self):
        handlers = [
            (r"/", MainHandler),
         (r"/userID", UserIDHandler), //set the ID of the participant
            (r"/mmd", MMDHandler), //Front-end of the Sample Application
            (r"/websocket", WebSocketHandler)
        ]
        #initializes web app
        tornado.web.Application.__init__(self, handlers, **settings)
```

In this application we have three routes:

- "/" - mapped to the MainHandler, which we will allow to initiate the user study workflow
- "/userID" - mapped to the User ID collection form - will collect the participant unique ID for logging purposes
- "/mmd" - mapped to the front-end handler - in our sample application, it is called "MMDHandler" which includes the HTML and JS template for our sample visualization
- "/websocket" - mapped to the WebSocketHandler, where our websocket can be accessed and our main application loops will run

The experimenter can add more handlers as needed (for example, pre- and post-questionnaires).

Handlers are Python class defined as follow:

```
  class YourHandlerName(tornado.web.RequestHandler):
        def get(self):
                // Action to do when the handler is launched

        def post(self):
            //action to do when the handler is done (only if the handler
     includes a HTML form that will return a POST message)
```

Handlers are also responsible for the workflow of the user study, as each handler calls the next one when they are done using the *redirect* command. In the examples below, the MainHandler redirects to the userID handler, and the userID handler redirects to the front-end handler (called MMDHandler here):

```
    class MainHandler(tornado.web.RequestHandler):
        def get(self):
                self.redirect('/userID)

    class UserIDHandler(tornado.web.RequestHandler):
        def get(self):
                self.render("userid.html")  //HTML  form  for  inputting  the
    user ID

        def post(self):
            //retrieve the user ID
            self.application.cur_user = self.get_argument('userID')
               //load front-end
               self.redirect('/mmd)

    class MMDHandler(tornado.web.RequestHandler):
        def get(self):
                self.render('MMDExperimenter.html') //main HTML file of our
    sample front-end

        def post(self):
            //...
```

The path to all the front-end files (HTML, JS, XML) called by the handler can be specified in *params.py*.

3.5.2 Specifying messages that can be sent through the WebSocket

The platform includes a specific handler (WebSocketHandler) that hosts the websocket used by the platform to communicate with the front-end. The WebSocketHandler also initiates the main application loop. In the *on_message* function of this handler, the experimenter can define messages that can be sent by the front-end to the platform, as well as how to react to those messages. See below for examples of messages.

```
class WebSocketHandler(tornado.websocket.WebSocketHandler):
     //…

    //Define how the platform should react to messages sent by the front-end.
Here one message is implemented: "switch_task". The experimenter can define
more messages here in the WebSocketHandler (if the front-end send a message
that is not implemented, the platform will send a "unexpected message" error.

    def on_message(self, message):
        if (message.find("switch_task") != -1):
         //upon receiving a "switch_task:" message, switch task
            result = message.split(":")
            next_task = int(result[1])

            self.stop_detection_components()
            self.tobii_controller.stopTracking()
            self.tobii_controller.flush()

            self.app_state_control.changeTask(next_task)
            self.initialize_detection_components()
            self.start_detection_components()
            self.tobii_controller.startTracking()
            return

        //… define other messages here

        else:
            self.stop_detection_components()
            self.tobii_controller.stopTracking()
            self.tobii_controller.destroy()
            self.app_state_control.resetApplication()
           print("unexpected message")
            return

    //what to do when closing the application
    def on_close(self):
       ... // omitted for the tutorial
```

The sample application provided with the study provide a comprehensive example of a user study entry file, that includes loading pre- and post-questionnaires as well as delivering multiple tasks in a randomized fashion: */experimenter_platform_study_bars_final.py*

3.6 Setting up the Front-End

We provide a sample front-end platform with support the sample visualization application defined in Section 1.2. However you should add your own front-end functionalities and files in the /frontend/ directory (we recommend using a specific directory for each front-end). Currently the sample application front-end includes two directories:

- /application/frontend/templates/ that includes the HTML files
- application/frontend/static/ that includes the JS, CSS and sample study data file

The core data-flow between the front-end and middle-end occurs through a websocket at the */websocket* endpoint (“ws://localhost:8888/websocket” if running on the local machine).
The front-end client receives data from the middle-end in the form of a JSON string. Delivery of interventions will have a “deliver” field with a list of interventions, each having properties matching the columns of the **intervention** table (see above). For example:

> { “deliver”: [{ “name” : “bar_highlight”, “function”: “highlightVisOnly”, “transition_in”: 1000, “transition_out”: 1000, “delay”: 0, “arguments”: {"type": "reference", "id": 1, "bold": true, "bold_thickness": 3, "desat": true, "color": "green", "arrow": false}, …] }

Similarly, removal of interventions will have a “removal” field, followed by a list of the intervention names to be removed. For example:

> { “remove”: [“bar_highlight”, “legend_highlight”, …] }

Once data has been received, it is passed through a handler, which either applies the appropriate javascript function to apply all the required graphical changes. For example, we might define a simple handler:

```
var ws = new WebSocket("ws://localhost:8888/websocket");
    ws.onmessage = function (evt){
       var obj = JSON.parse(evt.data);
       if (obj.remove != null) {
           handleRemoval(obj);          //removes graphics

       } else if (obj.deliver != null) {
           handleDelivery(obj);      //highlights graphics
       }
     }
```

Upon a task change the front-end can notify the back-end via the websocket connection, so long as there is a websockethandler class to receive messages from the front-end and change application accordingly. For example in the provided sample application, the front-end sends three possible *string* messages:

- “Close”, closes the websocket and terminates the adaptation loops
- “Next task”, switches to the next task in the queue

- “Switch task: *”, switches to task number *

For a more complete example of the front-end and the source code see https://github.com/ATUAV/ATUAV_Experimenter_Platform/blob/master/application/frontend/static/js/angular/app.js

3.7 Summary

Here is a summary of the different steps described so far to set-up a new experiment:

1. Define tasks and AOIs in the database as needed
2. Define relevant gaze and user events in the database
3. Define the rules, triggers and interventions in the database
4. Add and connect the ML user model if needed (optional)
5. Tune the parameters in *params.py* as needed
6. Add the route handlers in the entry point file
7. Add your visualization(s) in the *./application/frontend/* folder (preferably in a specific subdirectory)
8. Implement the JS functions dispatching the graphical adaptive interventions
9. Implement the JSON parser in the front-end to communicate with platform

4 RUNNING THE PLATFORM

The Experimenter Platform can be started with:

```
python entry_point_file.py
```

Note that you need to calibrate the eye tracker beforehand to collect reliable data.

The sample application can be accessed by running:

```
python experimenter_platform_study_1.py
```

And then by opening up a web browser to http://localhost:8888/

5 RETRIEVING LOGGED DATA

Once Experimenter Platform has terminated, log files for the state of the databases will be created for each of the completed tasks. If the platform unexpected crashes during execution, only the completed tasks so far will be logged. All logged files will be stored in the form of an executable Sqlite command under *./log/log_for_user_{i}_task_{j}.sq*l where *{i}* is the current user number (default 9999) and *{j}* is the relevant task number. The format of the log files can be customized via the *LOG_PREFIX* parameter in *params.py*.
In order to view the state of the database, open up a DB management application for Sqlite (eg. DB Browser) and execute the contents of the log file as a single sql command.

6 DEBUGGING COMMON ISSUES

1. Make sure to assign user_events and rules to the tasks that you wish for them to be active in. In addition, make sure that there is a definition for the necessary AOI for any given task in the aoi table.

2. Close other programs which might be using the databases (user_model_state.db and gaze_event_rules.db) when running the platform.
3. Make to write any changes made to the databases before running the platform.
4. Check that the SQL conditionals defined for the rules are not malformed sql queries.