**Intro to IoT project: A graphical interface for modelling IoT in BPjs**

**A project by:** Omri Basch – 203536917 & Roi Pillar-308292135

**Table of contents:**

**1. Introduction………………………………………………………………………** 2

**2. The diagram editor……………………………………………………………** 3

**2.1 Defining new blocks ……………………………………………….** 3

**2.2 Code generation………………………………………………………** 5

**3. The embedding Java application………………………………………..** 6

**4. Examples………………………………………………………………………….** 8

**Introduction**

The topic of our project is Modelling Behavioral programming via a diagram editor, and then interacting with that model using a Java application.

Our diagram editor is an extension of Google’s Blockly (see section 2), that enables the use of custom Behavioral Programming blocks. One can use the diagram editor to create a block scheme that in turn could be compiled into executable JavaScript code (using the BPjs library developed at BGU).

Given a JavaScript file, one can use our Java application to execute and interact with the code  
as further explained in section 3.

**The diagram editor**

Our work on extending Blockly included the following tasks:

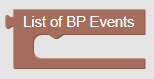
* 1. Defining new blocks – in order to support BPjs, we defined 10 new blocks, all of which we found to be essential to the task of generating BPjs code.

These blocks include:



* BP Event:

This is the basic building block of our BP blocks, it represents a bp.Event().



* A List of BP Events:

This block is useful when waiting/requesting/blocking a list of events rather than 1 (we also support untyped lists of BP events). It can be populated by regular BP Event blocks.

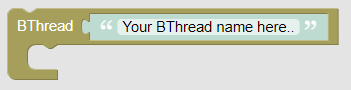


* A “bsync” statement:

This block represents a single bsync element (this is the statement version - no output).

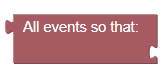
It’s meaning and appearance are quite straightforward, however the way in which it generates to code is interesting (see 2.2).

One can plug events or groups of events into its 3 inputs (wait/request/block).

* A BThread:

This block has a 1:1 correspondence(usually) with a node in an IoT network, e.g. an AC unit in a smart house), and all other blocks enrich its capabilities.

This block can include any number of statements, including loops and any other control blocks not necessarily related to BP.



* An EventSet:

This block allows stating a group of events implicitly, using a predicate over all events in consideration. It is very useful when targeting an infinitely large set of potential events that interest us in a given context.

* Some miscellaneous blocks:



String concatenation



Checking a prefix of a string against another string



Parsing a String as an Integer

* 1. **Code generation:**

The blocks generate to code as one would expect, apart from the bsync block’s generation, which is the first step in our downstream application’s event detection system – we generate not only the bsync statement corresponding to the bsync block that the user built(bsync({…}), but also some lines of code that our Java application parses and turns into suggested events for when the BPjs program is waiting for some external input. More on this in section 3.

**The embedding Java application**

The embedding Java application is the user’s interface to create, run and interact with running BPjs programs.

We will now go over the application’s functionality, and then present a recurring use case which will help understand the workflow in our system.

* **The “Editor” button:**

Use this button to open the diagram editor. where one can create, save and compile BPjs programs (as explained in section 2).

* **The “open” button:**

Use this button to browse and select .js files of BPjs programs.

* **The “Run” button:**

Use this button to run the selected BPjs program. As the program runs, you can read the log at the log area to debug your program.

We have created a custom logger that decorates the original output of the BPjs loggers.

* **The log area:**

Displays the running program’s log.

* **The Events area:**

While a running BPjs program is waiting for an event to be selected by the BPjs arbiter, these waited-for events are dynamically added the the “Events” list. This means that as the program executes, this list is populated according the current status of the program.

* **The Enqueue button:**

After selecting an event from the events area, one can enqueue the selected event to the external events queue of the BPjs program’s context by clicking the “Enqueue” button.

In addition to enqueuing events from the list, one can enqueue a different event by inserting free text as the event’s name in the text field below the “Events” list.

**A use case:**

1. The user starts the application.
2. The user clicks the “Editor”

2.1. The user creates a block scheme, and saves it.

2.2. The user exports the generated .js file.

3. The user clicks “Open” and browses for his .js file.

4. The user clicks “Run” and reads the trace of the execution.

5. If the program reaches a point where it is waiting for external input, the user  
 may use the “Events” area to enqueue external events.

**Examples**

We prepared some interesting use cases that could be run on our system:

1. A Smart House simulation: The nodes in the network (the house) are the AC unit, A digital thermostat, and lighting. The sensors (events) are: The door of the house opening/closing (this is a simplification meant to enable us to tell whether the tenants of the house are present in it or not.), a temperature sensor (streaming natural numbers), and an indicator of critically high power consumption in the house). The AC will not react to the temperature sensor while the tenants of the house are not home, as would the thermostat. Once the tenants get home, all 3 nodes will be up and running, with the AC node adjusting to temperatures defined as “too hot” or “too cold”.
2. A BP implementation of selection sort: This is done by defining each cell of some array as a BThread (with the appropriate block), and having 1 “selector” node (granted, this is not very decentralized as one would expect in an IoT environment, but we thought it was pretty neat), which requests the array members to be swapped in the fashion described by the selection sorting algorithm.

3) An airport simulation: The nodes in the network are the control tower, the landing strip crew, the refueling crew, the boarding crew and the runway crew.  
Whenever an airplane want to land in the airport (via external event “Airplane arriving”), the tower waits until the landing strip crew announces “clear for landing” and then allows the plane to land. Once the plane had landed, the refueling crew refuels the plane. Once they announce they are done refueling, the tower allows the boarding crew to board the passengers. After the boarding crew announces that the boarding is complete, the tower allows the plane to depart.

4) A mobile device’s brightness configuration management tool – the device’s brightness setting is set according to the device’s battery life, its camera brightness input, and the device’s user-defined brightness (if such exists). If and when a user defines his desired brightness setting, inputs from all sensors will be ignored, and when the user turns on “automatic brightness setting”, the sensors’ inputs will control the brightness setting.

5) A water tank’s sensor simulation: a water tank in an isolated location has a sensor on it that track the amount of water in it. whenever it rains, the water tank fills up (via the event “Rain: 25” for example, which means it had rained enough to fill 25% of the tank) and the sensor informs the user of the current water level. Whenever the user draws water from the tank, the sensor informs the new water level.