Debugger Plugin for Scraper Framework

Erdem Yildirim

March 2022

1 SCRAPER

Scraperflow (scraper) is a programming framework that inherits flow-based programming (FBP) paradigm. Every scraper program has a static control-flow graph (CFG) and a start node. Execution starts with this start node. Generated information packets, the so called flow-maps (FMs), flows through nodes following the paths in CFG. Nodes can be seen as "black-boxes" or algorithms that take incoming data, do some operation, and forward the modified (or unmodified) FM to next node in the path. Also, nodes can introduce concurrency.

In the following an interactive debugger for scraper framework will be introduced. We will first give a visual motivation of the debugger and then explain

the technical details.

2 VISUAL MOTIVATION

Before Connecting: Debugger desktop tool starts after the workflows are parsed and before their execution. To inspect dataflows we need to connect our debugger (frontend) to the program (backend).

After Connecting: We now see the CFG in a tree-like format in area A. Breakpoint setting happens here (with ctrl + l.click). Breakpoint means, execution of any dataflow stops when reaching that selected node (stops before the node's execution).

After Ready Signal: Ready signal sent, execution started. We see a visual tree structure in area B. This is the CFG itself, **not** the dataflow graph. For each CFG-node in B we can observe the **incoming** FMs in area C and the waiting flows in area D. Buttons in P_1 continue or step the overall flows, while buttons in P_2 continue or step

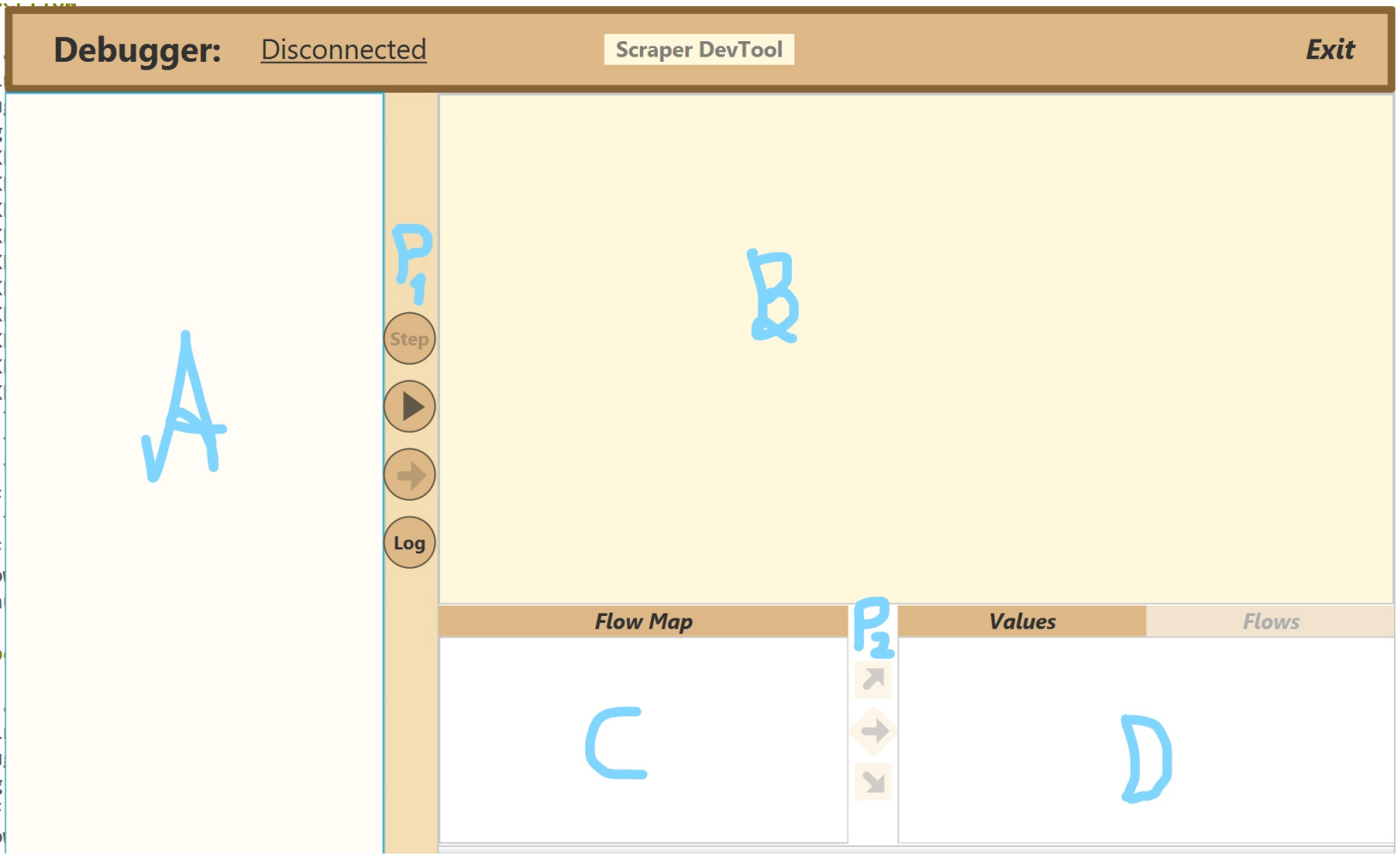

Figure 1: Before Connecting

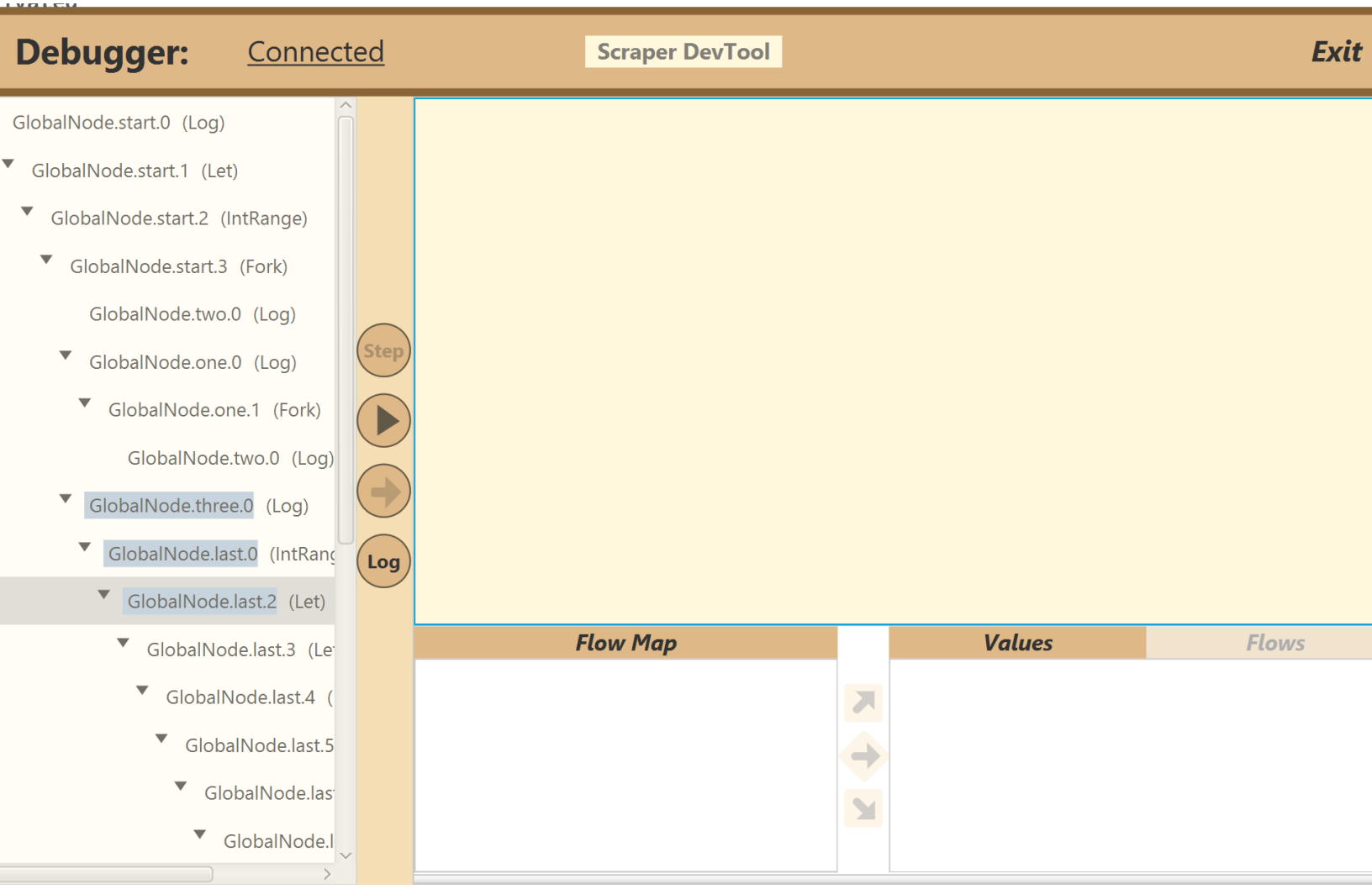

Figure 2: After Connecting

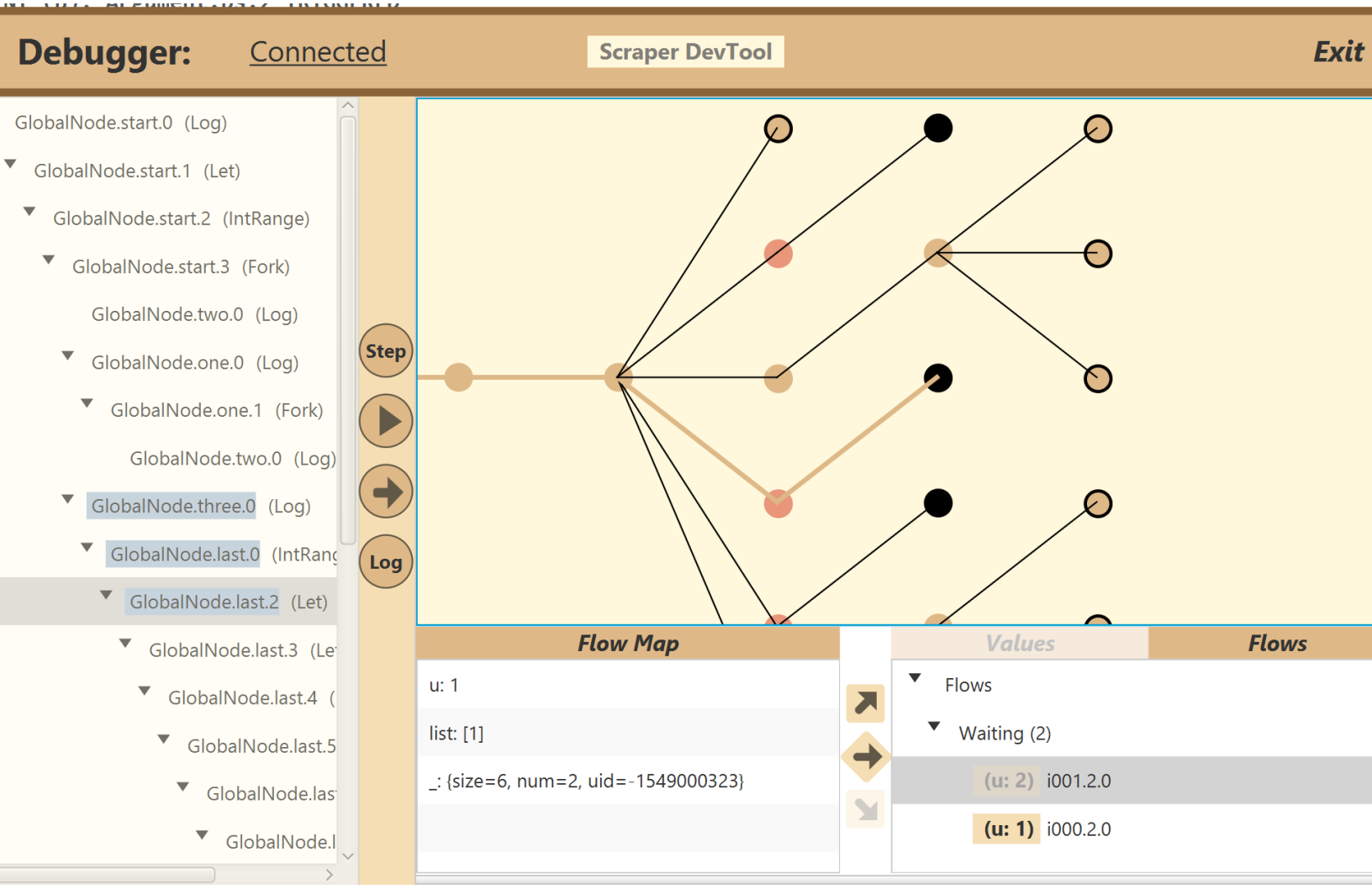

Figure 3: After ready signal, flow inspection

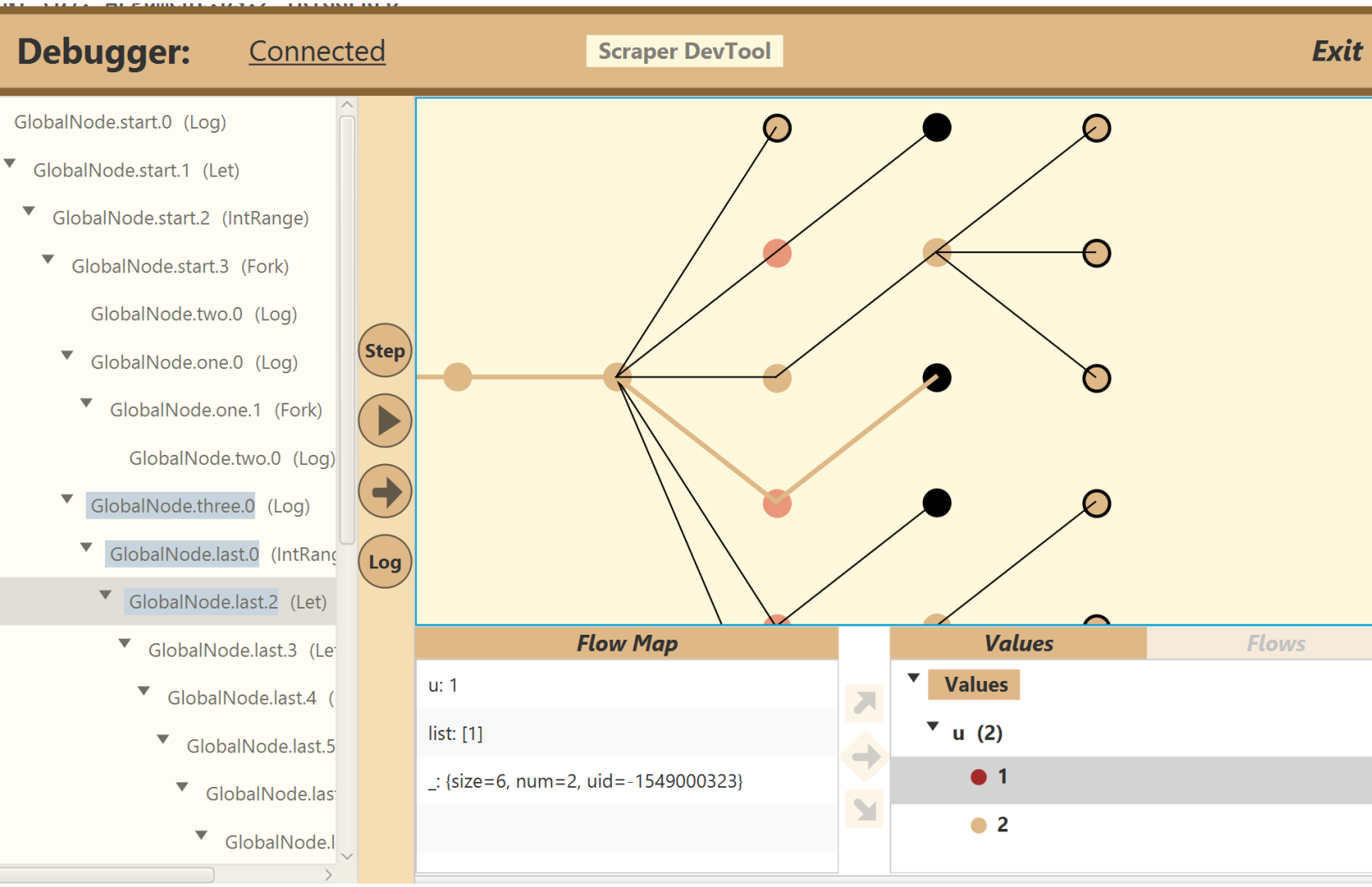

Figure 4: After ready signal, content inspection

user-selected flows. (to see flows: ctrl + l.click in runtime tree)

This being said, we can now explore the engineering behind the scene.

3 BACKEND

The backend part can be seen as a complete debugger add-on to scraper framework. There is a debugger *core* that can be added to program (scrape job) dependencies. After that, every hook, user defined or from framework, will have access to that core.

To make it clear, let´s give the definition for **Addon** and **Hook** in the framework. Both of them are interfaces from scraper-api. A class that implements a(n)

- **Addon** will be executed **before** any workflow specification is loaded, and has access to arguments and dependencies.

- **Hook** will be executed **after** workflow specifications are parsed and **before** workflows are executed, and has access to arguments, dependencies and parsed scrape jobs.

Both can be seen as framework commands that inject user-defined behaviour to program before its execution.

We make use of **Addon** and **Hook** commands in the project package *addon*. This package is responsible for the "injection" part, while *core* package for the "behaviour" part, but we will come to this point later.

Debugger-**injection** has three phases as seen in *addon* package:

1. **DebuggerAddon:** Adds core components to program dependencies.
2. **DebuggerHook:** Retrieves already added core components from dependencies and commands to start the server.
3. **WaitHook:** Retrieves the core from dependencies and commands to wait program exe-

cution until a ready signal (from frontend) arrives.

(1) is executed as first by its nature. (2) and (3) will be sorted such that (2) ordered before (3), thus executed before (3). Note that above defined phases as a whole allow us to inject further commands for debugging purposes, such as a frontend[1], in between (2) and (3).

Debugger-**behaviour**, the core, has six components:

1. **DebuggerActions:** Defines actions on flows that a frontend allowed to request for. For example, with *setReady()* a ready signal arrives, while *setBreakpoint(adr)* requests a break point on a node.
2. **DebuggerNodeHook:** Impelements the interface *NodeHook* from scraper-api. A node hook allows injecting behaviour **before** or **after** a node is starting or has finished processing a flow. This being said, its main purposes are

[1]This is what we are going to do in frontend part.

identifying each flow, checking node breaks and exceptions, and sending updates to frontend.

3. **DebuggerServer:** Encapsulates a *WebsocketServer* and defines send actions, e.g. *sendIdentifiedFlow()*, *sendBreakpointHit()*. User can provide his own websocket-server implementation or configure it as default.
4. **DebuggerState:** Provides blocking mechanism for underlying threads of flows. Also stores defined breakpoints.
5. **FlowIdentifier:** Identifies each flow with a unique string *IDENT* (not UUID) such that, when placed in a trie, all *IDENT*s build the whole dataflow tree.
6. **FlowPermissions:** Acts as a permission databank. Whenever a continue signal arrives, all waiting threads are notified, however each thread will check immediately if it has permission to continue. If no permission exists, it will wait again.

What is the **aim** of backend? Well, its aim is not storing FM content, instead sending it, leaving

frontend the work of structured FM storage. To make frontend this work easier, backend supplies two mechanisms:

1. An identification mechanism that generates for each flow a unique string *IDENT* such that, when placed in a trie, all *IDENT*s will build the dataflow tree. Of course, this *IDENT* will also be sent together with the FM.
2. The trie data structure as a map.

In this way, our frontend can build its database upon a trie, can store each incoming FM update one by one all the while remaining faithful to real dataflow graph.

4 FRONTEND with JavaFX

As explained above, after adding backend to our dependencies we can inject a new debugger behaviour: a frontend. To this end, we have a **FrontendHook** which executed after **DebuggerHook** and before **WaitHook**. The injected behaviour

is giving command to start our application, the javaFX platform.

Furthermore, frontend acts like a client and implements a client *websocket*, through which the bi-directional communication to backend server realised.

The GUI implementation combines known patterns MVC[2] and MVVM[3]. We can summarize the whole GUI under **DebuggerController** and **FlowTreeViewModel**.

1. **DebuggerController:** Mainly defines overall button actions and screen components that are
   - *FlowTreeController:* The controller of dynamic runtime tree. Defines node click actions for displaying desired FM contents.
   - *ValueSelectionViewModel:* Desired FM is displayed by selecting emitted variables. This class gives a selection mechanism for that.

[2]Model-View-Controller
[3]Model-View-ViewModel

- *FlowSelectionViewModel:* Supplies mechanisms for selecting which flows to continue. In addition to this, information of waiting or processed flows is stored.
- *ColoringUtil:* Defines semantics for runtime tree circle colors. For example, black and red means "this node is/was a breakpoint node".

Furthermore, whis controller tries to create runtime dynamic execution tree as a SVG[4]. For this work there is no additional library in play[5].

2. **FlowTreeViewModel:** Since backend does not store FMs, a frontend should undertake it. This class acts like a database that stores sent FMs in a trie, like explained in *backend*. But this class is also a view model for the CFG. As mentioned in *Visual Motivation*, the displayed runtime tree is the CFG itself, **not** the dataflow graph. This CFG is also stored in a tree structure with three kind of nodes:

[4]Scalable Vector Graphics

[5]I couldn't be able to find a suitable tree library in javaFX. Although binary tree libraries exist.

- *StaticNavNode:* to be explained
- *DynamicNavNode:* to be explained
- *DynamicForkNavNode:* to be explained

References