Debugger Plugin for ScraperFlow

Erdem Yildirim

April 2022

Write down the problem, think very hard, write down the answer.
Richard Feynman (1918 - 1988)

1 SCRAPER

ScraperFlow (scraper) is a programming framework that inherits flow-based programming (FBP) paradigm. Every scraper program has a 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[1].

1.1 Quasi-static Flow

During program execution, data flow forms another graph than CFG, the dataflow-graph. Here arises a new requirement, that this data-flow should be abstracted as a **quasi-static flow**. According to Scraper framework, *the data flow should be visualizable as a data flow graph, which abstracts from the actual data flow in a quasi-static way, i.e. combining multiple concurrent data flows in a single (crossed) arrow*[1, p.14]. To meet this requirement, framework introduces the notion of a **flow graph** (FG), which enhances the CFG with solid, dashed and crossed arrows[1].

1.2 Quasi-static Flow Tree

The capability of this debugger is limited to scraper programs, whose CFG forms a tree. Also, it is not possible, during program inspection,

to form the FG itself, since there is no support for dashed and crossed arrows. Instead, we introduce the notion of a **quasi-static flow tree** (CFT), that is the CFG itself with solid arrows, however nodes store all incoming data flows implicitly.

In the following, an interactive debugger for scraper framework will be introduced. The design consists of two high-level parts: the backend and the frontend.

2 BACKEND

Backend is designed as a complete debugger plugin. There is a *core* for debugger-behaviour, and an *addon* for debugger-injection. As names suggest, the *core* itself implements debug actions while the *addon* provides these actions to the framework.

Let's start with debugger-**injection**. This part implements the following interfaces from scraper-api:

- **Addon:** Executed **before** any workflow specification is loaded, and has access to arguments and dependencies.
- **Hook:** Executed **after** workflow specifications are parsed and **before** workflows are executed, and has access to arguments, dependencies and parsed scrape instances.
- **NodeHook:** Executed **before** or **after** a node is starting or has finished processing a flow.

Above interfaces are framework commands for injecting user-defined behaviour to program. In its implemented form, injection happens in four phases:

1. **DebuggerAddon:** The *core* is initialized and added to program dependencies.
2. **DebuggerHook:** Stores the parsed scrape instance for debugging and starts the server.

3. **WaitHook:** Executed as the last hook. Waits the workflow execution until a start signal from frontend arrives.
4. **DebuggerNodeHook:** Uses the *core* to inspect flows before node executions.

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

1. **DebuggerActions:** Defines program execution actions that the frontend allowed to request for. For example, with *startExecution()* the program execution start could be requested.
2. **DebuggerServer:** Extends a *WebSocketServer*[1] and defines the data sending protocol between backend and frontend. This protocol uses JSON strings that are serialized data transfer objects (DTOs), for communication[2].
3. **DebuggerState:** Implements a blocking mechanism for not permitted flows. Whenever some continue signal arrives, all blocked flows will be notified. However, each flow will then check immediately its permission. If no permission exists, flow will wait again.
4. **FlowIdentifier:** Identifies and stores each flow with a unique string **ident**, such that, when placed in a prefix tree, all **ident**s form the whole dataflow-tree of the program. The aim of this identification is to provide a more natural, more tree-like ID mechanism than that of UUIDs.
5. **FlowFilter:** Filters an incoming flow, by checking breakpoints and exceptions, to the group permitted or not permitted. In case of not permitted, suspends the flow until a continue signal arrives.
6. **FlowPermissions:** Acts as a permission databank. Each flow has initially permission to continue. This permission is removed by flow-filter if needed.

It is worthy to note that core components are useful, if and only if they are properly used by some **NodeHook** implementation. This backend provides a default **DebuggerNodeHook**. However, it is possible to implement a new one using the core and using it instead of the default one.

[1] https://github.com/TooTallNate/Java-WebSocket
[2] https://github.com/FasterXML/jackson

2.1 Designing Backend

Backend has a component-based approach to debugging problem. Each component is designed to *do one thing and do it well,* following the Unix-philosophy. Besides that, *don't clutter output with extraneous information* also applies: Backend produces outputs, frontend takes them as input. To reduce the output, **DebuggerNodeHook** omits inspecting data flows **after** node execution. This implies breakpoints to be set only before node execution.

Actually, inspecting data flows after node execution is not necessarily needed. For example, to inspect the FM after the execution of some node n_0, define a breakpoint on the next node n_1. If no such n_1 exists, it is possible to insert a *Log* node as n_1.

3 Frontend

Frontend, as a whole, acts like a client that connects to backend, requests actions on program execution, receives debug data, and displayes the state of the program. To design the frontend as modular as possible, the model-view-controller (MVC) pattern is applied with the additional support of viewmodels (VMs). Now, let's give an overview of this design.

3.1 Model

Our model implementation is dedicated to communication task. Hence, it extends a *WebSocketClient*[3] and communicates to backend through a websocket. Received debug data are stored in a tree structure that respects the actual quasi-static flow tree of the program.

3.2 ViewModel

According to wikipedia, a viewmodel *is a value converter, meaning the viewmodel is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented*[4]. There are two viewmodels: a specification-viewmodel for inspecting nodes

[3] https://github.com/TooTallNate/Java-WebSocket
[4] https://en.wikipedia.org/wiki/Model-view-viewmodel

in the program, and a values-viewmodel for inspecting flows, their content and their state.

The **specification**-viewmodel presents the control-flow tree within a **TreeView** to the user.

The **values**-viewmodel defines properties to display flow data within a **TableView**. On demand, all of the flow data in some node n will be converted to a tabular form using cell-value factories, then displayed in the table view.

3.3 Controller

In a debugger frontend, users control the program behaviour by clicking buttons. Buttons, on the other hand, internally define request actions, such as an execution stop request. In this sense, our controller defines the button click-actions.

3.4 Implementation

Frontend implements a **FrontendHook**, which is executed after **DebuggerHook** and before **WaitHook**. The main purpose of this hook is to start up the javaFX application thread.

After connecting to backend, frontend receives the program specification, which consists of node configurations and the CFG. Then, each node will be abstracted and stored as a **QuasiStaticNode** (QSN), which are able to store data flows. The **specification**-viewmodel lists them in a tree-view. At his point, one can define breakpoints on nodes.

After the execution start, the model maps incoming flow data to responsible QSN within a prefix tree, where keys are the flow **ident**s. Also, the state of the QSN changes whenver a flow arrives to it. It is this state that is displayed, when user demands an inspection on some node.

During execution, defined QSNs are dynamically visualized (with respect to CFT) within a **TreePane**, which defines a way of structuring javaFX **Circle**s in a tree.

References

[1] Albert Schimpf. *Formal Semantics for Composable Workflows in Scraper [Master's Thesis], https://git.server1.link/akp/master-thesis/blob/master/opsem.pdf.* 2019.