Features for debugging in a visual dataflow environment for end-user programmers

AN ABSTRACT OF THE PROJECT OF

Laxmi Ganesan for the degree of Master of Science in Computer Science presented on December 9, 2016.

Title: Features for debugging in a visual dataflow environment for end-user programmers.

Abstract approved:

_____

Christopher Scaffidi

Debugging, an integral part of software development, is difficult for end-user programmers, especially in the case of complex programs. The process of isolating errors is time consuming without the help of debugging support provided by the tool. For example, the visual programming tool LondonTube supports creation of custom mobile-cloud-web applications, but previous research indicated that the users of LondonTube had questions on usage of program components and run time status of a program even while creating a simple application. To address these issues, this project was focused on creating two features, 'Analyze' and 'Trace', improving the visual programming tool by providing static analysis and runtime status for helping users to find/fix errors. A laboratory experiment evaluated the effectiveness of the prototype in comparison to the baseline (without the new features). The results of this study revealed that the users of this prototype were more satisfied with the system, took less time to complete assigned tasks, and asked fewer questions about usage of program components.

Features for debugging in a visual dataflow environment for end-user programmers

by
Laxmi Ganesan

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 9, 2016
Commencement June 2017

Master of Science project of Laxmi Ganesan presented on December 9, 2016

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Laxmi Ganesan, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my major advisor, Dr Christopher Scaffidi, for providing me with the opportunity to work on this project and for the continuous support. His motivation and guidance helped me stay on track towards accomplishing my academic goals.

I would like to extend my appreciation to my program committee members, Dr Mike Bailey and Dr Anita Sarma.

I would like to thank Andrew Dove from National Instruments for providing his valuable thoughts and feedback on the project, fellow researcher Sam Lichlyter who helped me during the implementation phase of the project, Nicole Thompson and all staffs of EECS department.

Finally, I would like to thank my family for always believing in me, for their constant support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Figure Page

1 Introduction

In software development, debugging is the process of identifying issues and fixing them so that the program works as intended. As the process of isolating errors is time consuming, software developers use debuggers for locating errors in the programs. There are numerous tools with debugging support, for example Microsoft Visual Studio [1], IntelliJIDEA [2], and LabVIEW [3].

Debugging programs are difficult for programmers irrespective of the level of experience that one has, but can be even more challenging for end-user programmers and others with minimal experience with programming. End-user programmers frequently lack this experience because they are defined as people who write programs, but not as their primary job function [4]. Like professional programmers, end-user programmers also create bugs and therefore must perform debugging [5].

End-user programming tools are often domain specific and differ from one another in terms of what kinds of debugging features they offer. For example, Microsoft Excel auditing tools 'Formula Auditing' shows the relationship between formulas and cells [6], WYSIWYT has fault localization [7], Whyline debugger allows programmers to ask "Why did/didn't" program-related questions [8], LabVIEW Debugger has execution highlighting, single-stepping, and breakpoints [3]. But when a new kind of tool appears for a new kind of a domain, this can create the need to support debugging in new kinds of ways. An example is LondonTube [9], which is a visual programming tool, for creating mobile-cloud-web applications. The intended end-users in this context

are health researchers using this tool. A typical application involves collecting data from a mobile device, sending data to cloud/server where the data gets saved in a database and when there is a request the data is downloaded from database to be displayed on a browser. Each LondonTube program, written in a dataflow language, is converted into pieces of JavaScript that the system deploys on mobile devices and servers.

Prior research [9] indicated that LondonTube users could create simple mobile-cloud-web applications much faster than users of traditional textual programming tools (e.g., Eclipse), thanks to the domain-specific dataflow language. Yet that study also indicated that people using LondonTube tool had questions on how program components work, and how to tell if program is running. These questions indicated that the users had difficulty in using program components and identifying the status of a program even for simple applications. LondonTube had no debugging features at all to help users to answer these questions. Also, in general, as the complexity of the program increases debugging also becomes more difficult [10]. Identifying errors in a large program requires time and effort. Debugging a program that runs simultaneously on multiple machines, as in the case of LondonTube programs, is even more challenging. If the tool provides some support for the users to identify problems in their code, it would make their debugging experience better and efficient.

This project, therefore, presents new features enabling end-user programmers to more effectively perform debugging on the mobile-cloud-web applications that

LondonTube supports. It incorporates static analysis aimed at helping to reduce users' time spent on debugging. Static analysis is the analysis of program performed before executing it to uncover errors. In addition, the tool takes advantage of runtime data to provide runtime status of the executing program. The new features for LondonTube were called 'Analyze' and 'Trace'.

'Analyze' statically analyzes the code and indicates static errors in the program. Letting the users know of the static errors/warnings in the program (for example, a missing mandatory connection, unused program component, missing optional connection) was expected to help them in understanding the usage of the components better and uncover errors on the fly as and when they create the program.

'Trace' determines the status of a running program to find/indicate what parts of the code run well and what parts of code don't run well using run time statistics. This was also expected to help users in debugging by locating nodes that aren't running.

A laboratory experiment was conducted to evaluate the effectiveness of the prototype by comparing the results of a group using the tool with the debugging features available to them to the baseline i.e. group without having access to the debugging features. The results of this study indicated that the new features help users in localizing errors in the program. The users with access to these features also could accomplish the tasks more quickly and asked fewer questions about program

components' usage during implementation than those who did not have access to the features. The users found 'Analyze' and 'Trace' features easy to learn and use.

2 Related Work

This section discusses existing tools that perform static code analysis and tools that provide support, that helps users with debugging, using execution specifics.

2.1 Static Analysis

Programming environments, such as visual and textual, offer different kinds of static analysis support for debugging. The 'Analyze' feature of LondonTube is based upon the static analysis performed in these tools that aid debugging. 'Analyze' performs static analysis of the program based on set of rules pre-determined for each node.

Visual programming environments offer different kinds of static analysis support for debugging and these tools listed below are used by end-user programmers.

- Yahoo! Pipes [11] is a web based visual programming environment that has anomaly detector that statically analyses code based on predetermined rules including the presence of incoming and outgoing wires, links etc. The results of the study show that these debugging enhancements helped end user programmers localize and fix bugs. Unlike Yahoo! Pipes, LondonTube does not check for data format, type.
- LabVIEW [12] [13] has static code analysis to examine source code to look for potential problems without having to compile or execute the program. The errors/warnings are shown to the user as an error list. Likewise, the 'Analyze' feature highlights each node in the program indicating whether there is a static error/warning or not which also gives users feedback on all errors/warnings in

the program. Unlike LabVIEW, LondonTube doesn't check for type mismatch, performance problems, style conventions etc.

- Hairball [14] a tool for Scratch visual programming environment that helps students to identify errors and unsafe practices while creating animations (formative). This tool is also used by graders to inspect the implementations (summative). The study shows that Hairball helps in identifying correct instances with 0.5% false positive and has been suggested as a tool that can be used in addition to manual analysis.

Textual languages also offer static analysis support. Some tools(not limited to) have been listed below.

- FindBugs [15] that helps in finding bugs in Java programs. Some errors found using this framework are, for example, null pointer exception, infinite loop, incompatible comparison, ignored return value, cast exception etc. Study [16] confirms the usefulness of FindBugs during software development.
- Lint [17] is a static analysis tool used for finding defects in C programs. It checks for unused variables, infinite loop, validates arguments of functions, unreachable code and a lot of other checks. It was the first widely used static analysis tool for finding errors [16].
- The static analysis framework [18] for students writing Java program for an introductory course received positive feedback. It includes the following checks, unused variables, hardcoded values, whether each case block in switch

statement has break, number of characters per line etc. The analysis is more focused on problems that students face while programming in Java.

- PMD, JLint [19], like FindBugs, performs static analysis.
- Other static analysis tools are Microsoft's PREfix which simulates execution to suggest errors and PREfast [20] that does syntactic and semantic analysis to detect errors, Codan framework [21] that detects common defects like syntax errors in C/C++, CppCheck [22] to uncover issues not caught during compilation of C/C++ programs, IntellijIDEA [23] has static code analyzer that analyzes Java code on the fly as the users type the program, PyCharm [24] that performs code analysis for Python.

Some of these tools allow users to customize the kind of check performed by analyzer. As one can observe, for each tool, the kinds of static analysis performed may vary depending on the language, domain and/or target users. Likewise, the rules determined for static analysis in LondonTube is to help end-user programmers who may face issues while using this tool.

2.2 Runtime Status

Some existing tools make use of the runtime information to provide status of a running program by means of execution highlighting. For example, LabVIEW [3] highlights the execution flow of the program while the program runs, FIELD [25] shows the call graph by highlighting methods that are currently executing, Whyline [26] lets users do interactive debugging.

- Execution Highlighting in LabVIEW [3] allows the programmers to view what part of the code gets executed and in what order along with the values that are passed from one component to another. Tracing slows down the execution so that the progress is noticeable. It is also possible, in LabVIEW, to set breakpoints and step through the components along with an option to probe the value of a component.
- FIELD [25] programming environment has call graph visualizations that shows what node is currently executing. The active nodes are also colored to differentiate from other nodes. These colors can also be changed by the user if preferred. Alternatively, there is also an option to view the current executing method in a class hierarchy.
- Whyline [26], a debugging interface/tool for Alice visual environment, allows users to ask 'Why did' and 'Why Didn't' about program's runtime failures and provides answers using execution specifics. The execution history shows the runtime actions along with what data was sent/received for each action after the program stops because of failure. Study [26] shows this debugger helps in reducing debugging time by a factor of 8.

Some tools use runtime specifics for error localization. Error localization is a technique that identify sections of code that can be buggy. In the event of failure, localizing errors helps programmers in debugging. For example, error localization feature is provided by WYSIWYT (What You See Is What You Test) for spreadsheets [27], Yahoo! Pipes [11], Gneiss [28].

- Yahoo! Pipes [11], a visual programming environment uses execution trace to detect anomalies like range checks, dynamic validation of data types.
- Gneiss [28] tool extended spreadsheet model to interact with web service data. During run time, in case of an error, the erroneous cells are updated with error data. Also, the values that change during runtime are visible to users which makes debugging easy in the tool.
- WYSIWYT help users in locating errors in the program which aid users with debugging. [8] [27] specifies that to start debugging users should be able to identify the problem in the program. WYSIWYT for spreadsheets provides fault localization based on the cells that have been marked and unmarked to locate cells containing faults by providing immediate visual feedback [27]. The users can also look for more information in the form of tooltips. Like WYSIWYT, 'Trace' in LondonTube provides visual feedback based on runtime information for each node.

Similar to the tools discussed above, 'Trace' feature of LondonTube also uses execution specifics to provide information about program status for the nodes in the program. By providing the status of running program, this feature also helps users with debugging by indicating the sections of code that is still executing, hasn't executed and finished executing. It also does error localization by showing nodes that did not run which means these nodes didn't receive data.

Most of the tools like Visual studio [1], PyCharm [24], IntelliJIDEA [2], Chrome Dev Tool [29] provide debugging support by means of letting the users set

breakpoints, step through the program by controlling program execution, watch variable values. ‘Trace’ feature added to ‘LondonTube’ aim at providing runtime status of a program at high granularity (for each node) and thereby providing sections of program that might need user’s attention. When the traditional debugging support is added to the tool in future, ‘Trace’ feature would help the programmers by letting them know where to set breakpoints and watch values.

Some valuable features that can potentially be added to LondonTube, based on the review of related work, are discussed in Section 7.

2.3 LondonTube

Prior to this project, LondonTube lacked any debugging features of the types described above. However, a clear understanding of the LondonTube environment is essential for understanding the context of the current work. Therefore, this section provides a brief summary of the LondonTube language, the programming tool, and the runtime environment, as explained in detail by prior work [9].

LondonTube system has two editors ‘Front panel’ and ‘Back panel’. ‘Front panel’ is used for creating the User Interface layout and ‘Back panel’ is used for creating programs that define application’s behavior. The editor has a widget palette which consists of the components that can be used in a program and an editing area as shown in Figure 1. The widgets are dragged and dropped into the editor area and these widgets can be resized.

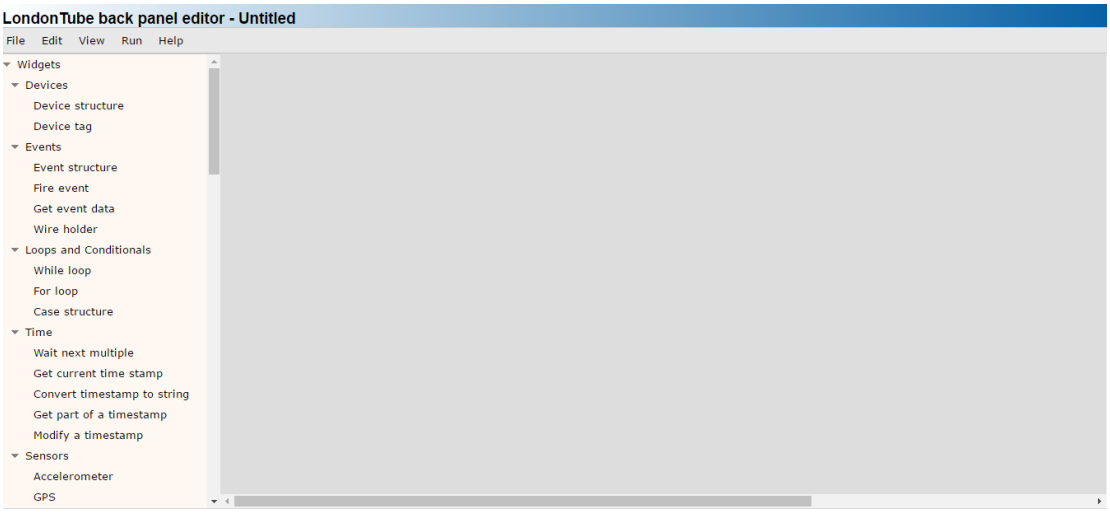

Figure 1 LondonTube Back panel editor

LondonTube language follows structured dataflow model consisting of a set of computational elements (each element represented as a node). These elements can contain one another. Each node has some inputs/outputs. A node runs (asynchronously) when it receives data as inputs from other components connected to it and sends the results as output to a node to which it is connected. Below are some language specific constructs explained in order to understand the illustrated example in Figure 2.

- Android device must have LondonTube application installed and should be registered so that the device can send/receive data.
- Each device used in ‘Back panel’ program will be represented by ‘Device Structure’ which is the structure that specifies where the program should run. i.e. Cloud, Android, Browser.
- Most of the code will be inside ‘Event Structure’. Whenever the event is triggered, the code that it contains would start running. Event can be user-defined or is triggered when device starts (which runs just once). If events are

user defined, ‘Event Structure’ will have an incoming wire. Event related nodes are associated with golden yellow color.

- A Database is used in Cloud device structure for storing data. Data can be loaded and retrieved from database. Database related nodes are associated with Black color.
- Sensors are used in Android device and the available sensors are Temperature, Accelerometer and GPS.
- While and For loops are represented by grey rectangular structures.
- UI is used in Browser to show the user interface elements. The user interface layout is created by the programmer in Front Panel and used in Back Panel. UI used in Back Panel is associated with orange color.

Figure 2 is an example of an application [9], created using LondonTube. There are three devices used in the program Android, Cloud, Browser. The intention of the program is to send the accelerometer values to the server every 60 seconds where it is saved in a database. When browser loads, a request is sent to server. The accelerometer values are downloaded from the database and sent back to browser where it is visualized in the form of a chart.

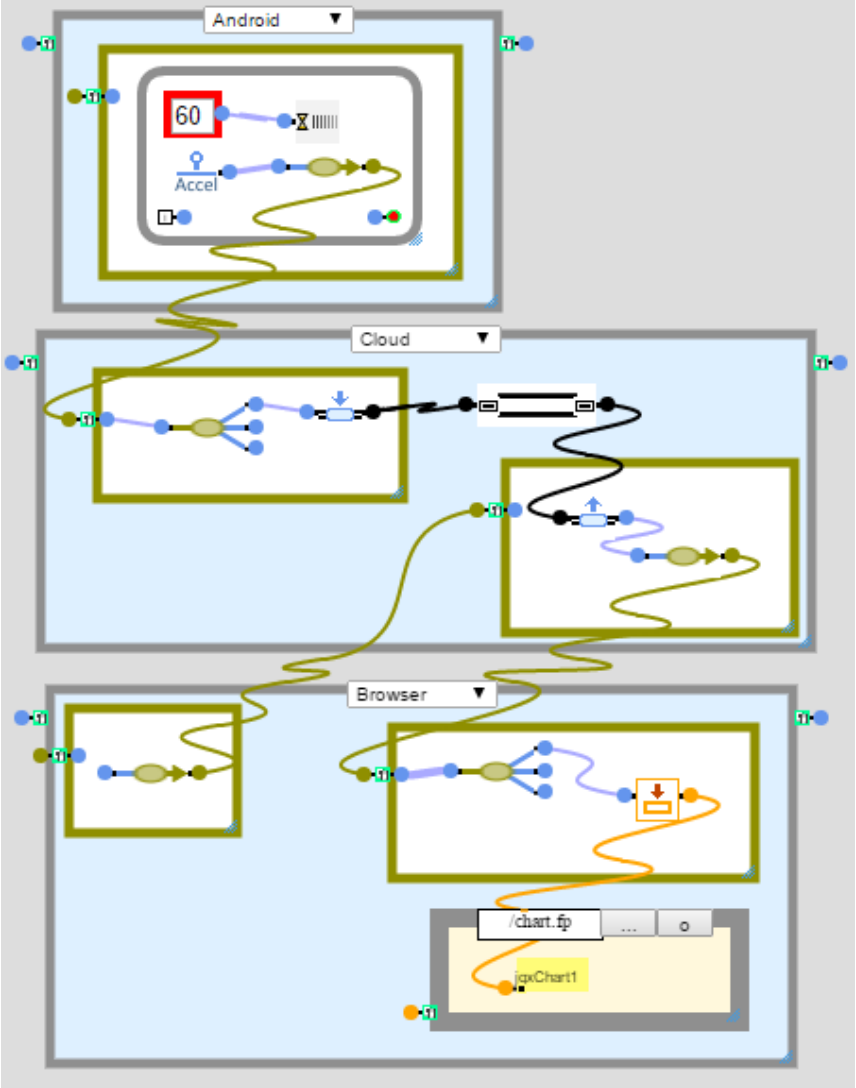

Figure 2 An application created using LondonTube language

Once the application is created and ‘Run’ is chosen, a message is sent to the server for the program to be compiled and deployed. Compilation involves creating a JavaScript file for each device (Android, Cloud, Browser) used in the program and Execution of code involves running the JavaScript files on corresponding devices. When user chooses ‘Stop’, the execution terminates.

Some additional information on the terminology specific to LondonTube, used in this document, has been described below.

- Endpoint/Terminals: These are depicted as filled circle attached to each node. These endpoints can receive or send data. The endpoints found to the left of the node receives data and endpoints on the right of the node sends data.
- Variant: A variant can hold properties (name, value).
- Dashboard: it is the browser component where the output can be viewed (along with UI if any). Here is where the code specified in 'Browser' device structure runs.
- Menu: The Back panel editor has several menu options that users can choose. For example, operations related to File, Edit, Run etc. are available off the menu.
- Error dialog: It is similar to print message. Any message sent to error dialog will be printed.
- Fire event: Fire event triggers an event. It has an optional endpoint to which data is wrapped (event name) and a mandatory endpoint which sends the data by triggering an event.

3 Design and Implementation

The design of the prototype involved adding two features ‘Analyze’ and ‘Trace’ to the tool.

- ‘Analyze’ helps with identifying static errors in the program.
- ‘Trace’ runs the program and indicates the run time status of the program.

System’s feedback on compilation error

Below is an example of the system’s status when there is a static error to show what the user would see in case of an error. Figure 3 shows a program having a missing connection to UI component textOutput1. The intent of the program is to send a random number to text output to be displayed on Browser. In this case, the user sees an error message, shown in Figure 4 and wouldn’t be able to run the program since this is a compilation error. The wire from ‘Write to UI’ has been connected to a different endpoint instead of textOutput1 It is difficult for the programmer to comprehend the error message and fix the issue since programmer will not be aware of the code in the background that gets compiled. Therefore, in case of such issues, it would help the end-user programmer if the error message is shown associating the visual program that user created.

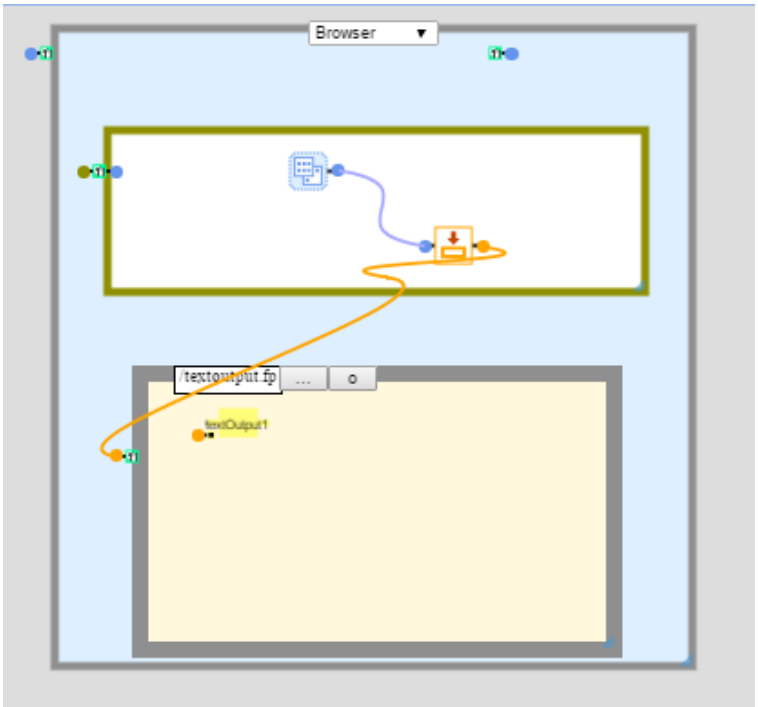

Figure 3 Program that causes compilation error

```
------------------------------------
compilation error on '/379443213132053.bp'
------------------------------------
edu.orst.pxp.bp.compiler.BpCompilerException: Widget name not specified
        at edu.orst.pxp.bp.compiler.visitor.writetoui.generate(Unknown Source)
        at edu.orst.pxp.bp.compiler.visitor.BpNodeVisitor.generate(Unknown Source)
        at edu.orst.pxp.bp.compiler.visitor.eventstructure.generate(Unknown Source)
        at edu.orst.pxp.bp.compiler.visitor.spaceloop.generate(Unknown Source)
        at edu.orst.pxp.bp.compiler.visitor.panel.generate(Unknown Source)
        at edu.orst.pxp.bp.compiler.BpCompiler.compile(Unknown Source)
        at edu.orst.pxp.bp.compiler.BpCompiler.compile(Unknown Source)
        at edu.orst.pxp.eup.server.FileServlet.compileFile(Unknown Source)
        at edu.orst.pxp.eup.server.FileServlet.buildFile(Unknown Source)
        at edu.orst.pxp.eup.server.FileServlet.doPost(Unknown Source)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:648)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:292)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:207)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:240)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:207)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:212)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:502)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:141)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:616)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:88)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:522)
        at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1095)
        at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:672)
        at org.apache.tomcat.util.net.AprEndpoint$SocketProcessor.doRun(AprEndpoint.java:2500)
        at org.apache.tomcat.util.net.AprEndpoint$SocketProcessor.run(AprEndpoint.java:2489)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)
------------------------------------
```

Figure 4 Compilation error displayed on clicking Runtime status page for program in previous figure

System's feedback on runtime error

Figure 5 shows a program which may not produce desired results. The intent of the program is to send random number to Browser, every 3 seconds, when there is request from Browser. User will see the output as shown in Figure 6.

The fire event which sends data to Browser is outside the while loop which will never receive random number. This is because the loop is infinite. So, the node fire event will never run and the subsequent nodes depending on it will also not run. However, user may not be aware of this situation since the tool doesn't provide any information on the runtime status.

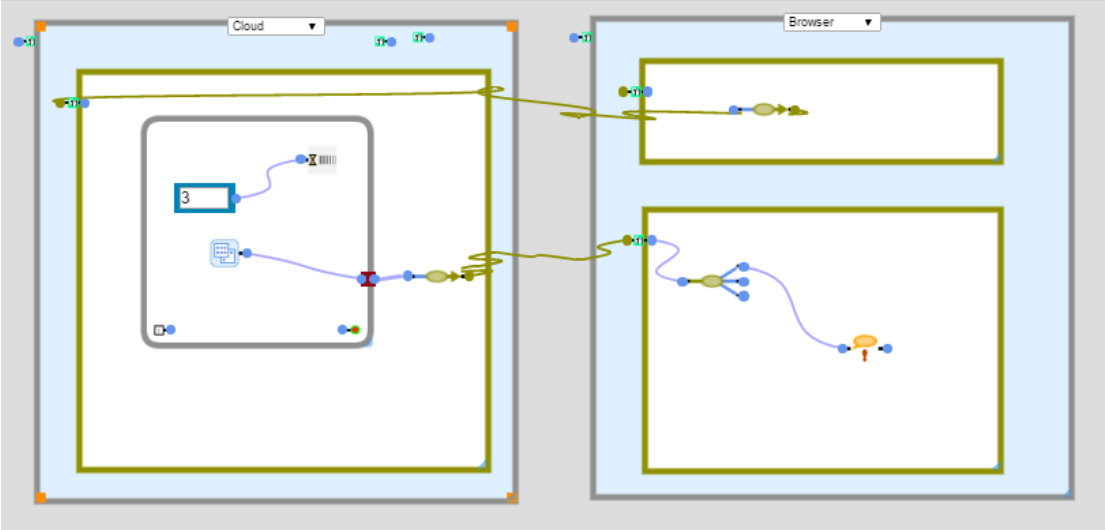

Figure 5 Program that causes runtime error

Figure 6 'Dashboard'(where results are viewed) for program in Figure 5

For both the features, it was important to let the users know of the status of each node/component in the program so that it helps the users with debugging. The program components appear as nodes on screen, though they are associated with unique identifiers in the background, these aren't visible to the programmers. For example, if there are two fire events (described in section 2.3) used in a program, there can two identifiers in the background used for compilation/execution (e.g. Fireevent1, Fireevent2). However, the appearance of these components on the tool looks the same. Therefore, in order to associate a status with each node, it was decided to have an indication (a specific color) surrounding each node.

IFT design pattern

IFT (Information Foraging Theory) helps predict/explain information seeking behavior. [30] provides information on various design patterns, based on IFT that can help tool designers to incorporate during development which would potentially help developers using the tool to find information that they seek.

Two IFT design patterns 'Fault Localization' and 'Software Visualization', as described in [30], have been used for implementing 'Analyze' and 'Trace'.

- Software Visualization: The intent of this pattern is to provide visual cues to identify sections of code related to a metric. Analyzer colors the nodes in green indicating there aren't static errors, yellow to indicate some optional endpoints aren't connected, red to indicate mandatory endpoints are not connected. Trace also colors the nodes in green to indicate it has finished

running, yellow to indicate they are not a part of running instance, orange to indicate they are still running and red to indicate the node hasn't run.

- Fault Localization pattern: The intent of this pattern is to let the developers know of the parts of program that is responsible for undesired behavior based on runtime events. By adapting this concept, 'Trace' uses the run time statistics of each node to determine whether it is running, not running or has finished running. This information would help the developers identify parts that aren't running and eventually help them debug the code.

Cue decorator is another pattern that is related to changing the appearance of a cue to grab the attention of developer by making it more noticeable (for example, highlighting) and by provide more information (for example, tooltips). However, this pattern wasn't considered because cues are information features associated with links and the information features (nodes) in LondonTube program weren't navigable.

3.1 Analyze

'Analyze' helps in evaluating the program for static errors i.e. whether a program component/node has been used appropriately or not. Every component has some endpoints/terminals associated with it. A common problem that one can experience when learning LondonTube is remembering to connect endpoints; sometimes, it is even difficult to determine which endpoints even need to be connected, since some are optional. For example, the 'Fire event' node has two endpoints (terminals), one input endpoint to receive an event name and an output endpoint to fire event. The purpose of 'Fire event' event is to fire an event to be handled by 'Event structure'. The input

endpoint event name is optional, since an event can be used for sending an empty message just to signal that the 'Event Structure' can start executing the code. On the other hand, the output endpoint event is mandatory. Without the help of analyzer, it is difficult to know that event name endpoint is optional.

- When some endpoints are not connected, this may lead to compilation errors and it may not be possible to run the program.
- When some endpoints are not connected, the program may not produce results as intended.
- Some endpoints of a node are mandatory/optional.
- For a program that's complex involving lots of components with overlapping wires, it becomes difficult to identify a missing connection.

'Analyze' feature is available off the menu and can be turned on anytime except when the program is running because it is intended to identify static errors before program execution. When this feature is turned on, the analysis happens and the nodes are highlighted in a specific color to provide feedback. Table 1 shows the different colors and their significance.

Color	Implication
Green	All endpoints are connected
Yellow	Optional endpoints are not connected
Red	Mandatory endpoints are not connected

Table 1 Indication of Static errors

This feature automatically detects changes done to a program and updates the indication. For example, if there is an error in the program because of an endpoint not connected, as soon as the endpoint is wired up the indication changes from ‘red’ to ‘green’.

Hovering over the highlighted area will indicate what endpoints haven’t been connected. Turning the analyzer off will clear the indicators.

The current option of whether the analyzer has been turned on or not will be saved in local storage. (Key: analyzer, Value: true/false). Local storage is used by web applications for storing data locally within browser without having to transfer information to server. Whenever page gets refreshed/reloaded, this value is read and appropriate action is taken i.e. if user had chosen ‘Analyze’ prior to page refresh, then the static analysis would happen and the nodes would be highlighted.

In the subsections below are some examples that indicate static errors, which are based on my own experience when learning the LondonTube language.

3.1.1 Compilation Errors caused by absence of a connection

The intention of the program shown in Figure 7 is to send a random number to cloud and store it in a database. When browser sends a request, the value should be retrieved from database and printed. The connection to/from database is missing which leads to a compilation error. This program could be a part of a larger program where identifying a missing connection is difficult. During static analyses, these nodes get highlighted in red grabbing the attention of users to fix them. Some nodes are highlighted in yellow which indicates that some optional endpoints are not connected.

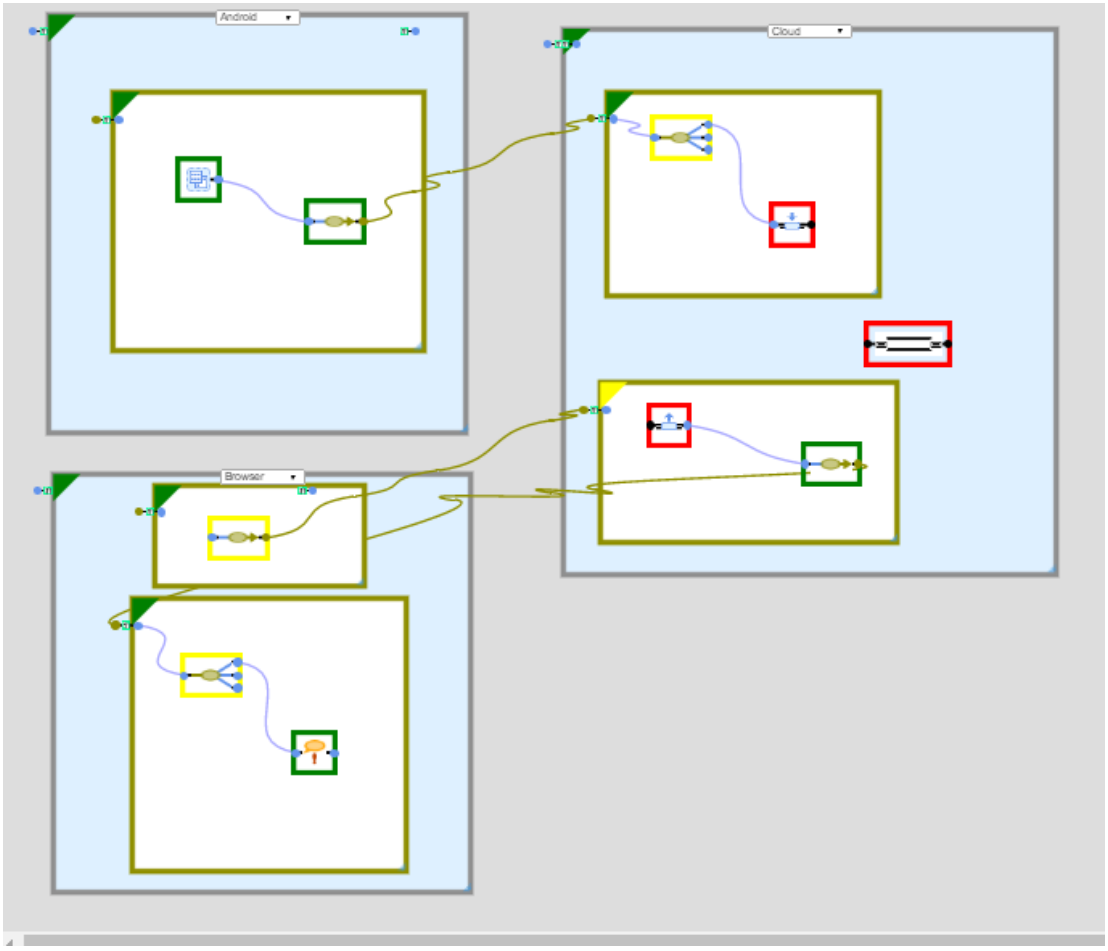

Figure 7 Program with missing connection that leads to compilation error

3.1.2 Mandatory/Optional connection

The purpose of the program shown in Figure 8 is to print a default message on browser continuously. The grey rectangle is a while loop which is tagged in red. This is because the shift registers added (visible on the boundary of the loop), having mandatory endpoints, have not been connected. Error dialog is highlighted in yellow since it doesn't receive a message and this is optional.

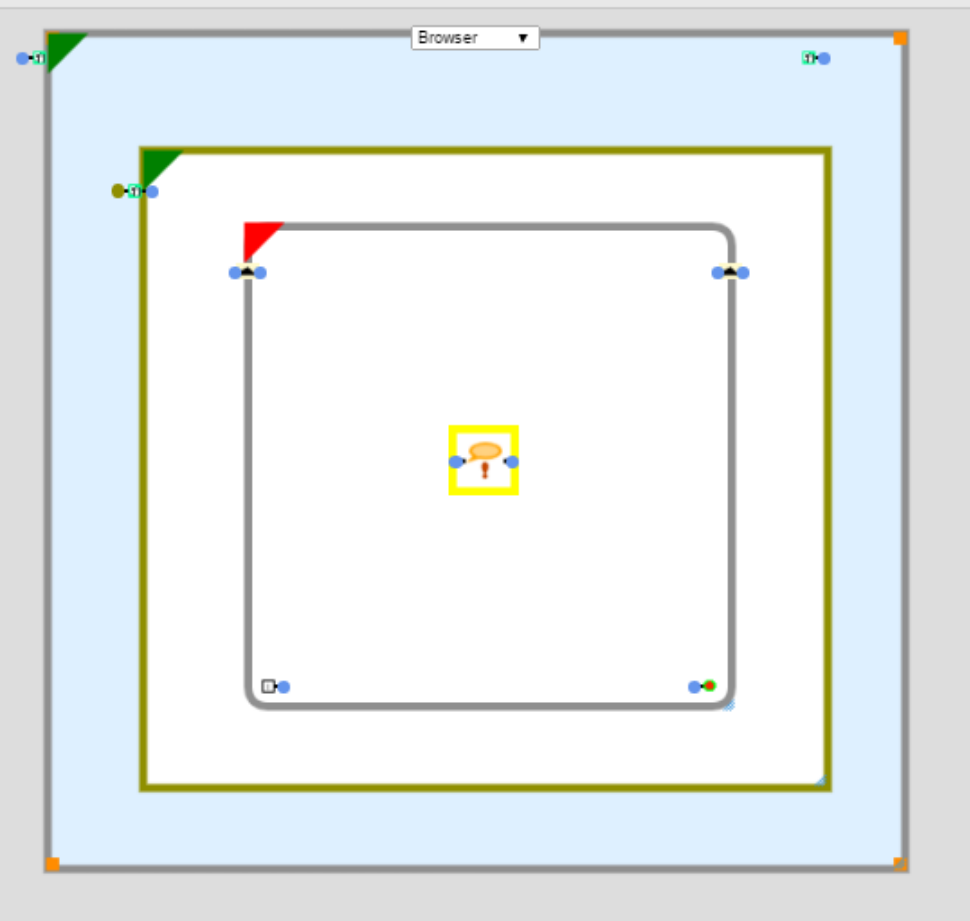

Figure 8 Program with Mandatory endpoint and Optional endpoint not connected

3.1.3 Missing connection in a large program

The purpose of the program shown in Figure 9 is to validate a survey and accelerometry depending on which an appropriate message is shown to browser. The program is similar to the kinds of programs that LondonTube was intended to help health researchers to implement for monitoring exercise activity [9]. This program has lots of components and the ‘get variant property’ node is highlighted in red to indicate that one of the mandatory endpoints has not been connected.

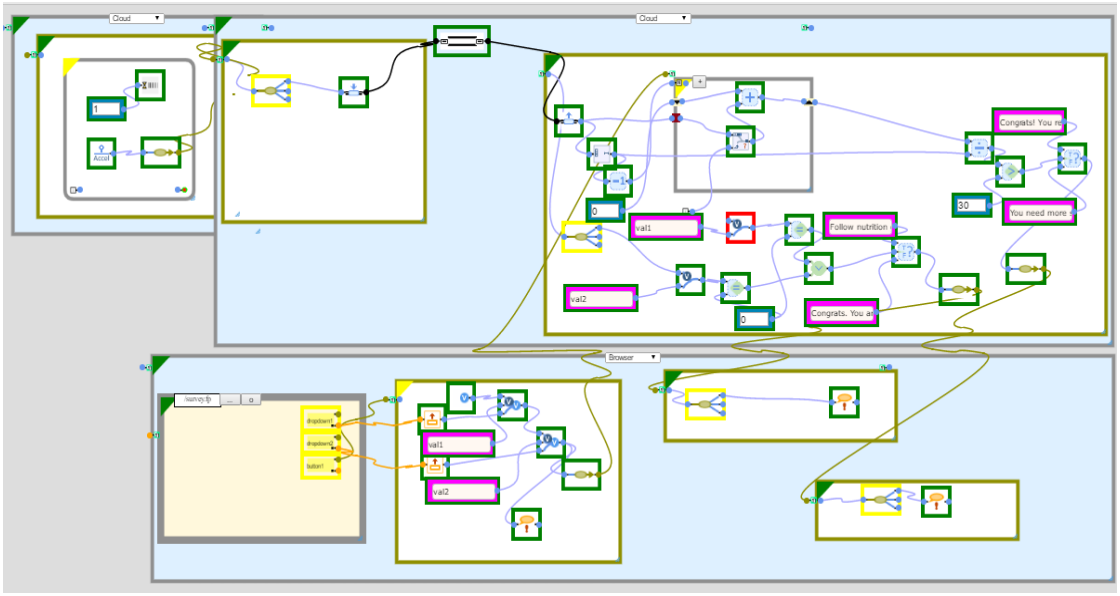

Figure 9 An application with missing connection

3.2 Trace

‘Trace’ feature runs the program and also helps users by indicating the run time status of a program, for example, what parts of the program are executing, what parts of the program are not executing and the ones that finished executing by comparing the current snapshot of runtime statistics with previous snapshot.

- When a program starts executing, it becomes difficult to identify what parts of the program have run and the ones that didn’t run.
- Sometimes not starting up the Browser or not having the Android device registered may not trigger the event that should be executing.
- The program components are stacked on top of one another. For some cases, it is difficult to know which component is at the top of the stack.
- When a program is executing, it becomes difficult to localize errors, when it does not produce intended results. For example, a node that hasn’t received data will not run.
- When a program is edited while the program is running, it becomes difficult to differentiate the newly added components from the ones that are already a part of the running instance.

‘Trace’ feature is available next to the ‘Run’ feature off the menu. Upon choosing ‘Trace’ feature, the program is first analyzed statically and if there aren’t any static errors the program runs and the status gets updated periodically (say, every 10 seconds) until the program stops running. If the program has static errors, then an alert message

is shown to the user and the analyzer is turned on to indicate static errors. Otherwise, the nodes are highlighted in a specified color, as listed in Table 2, to indicate run time status.

Color	Status
Green	Node has finished running successfully
Yellow	Node is not a part of running instance
Orange	Node is currently executing
Red	Node has not executed

Table 2 Indication of Runtime status

To identify the run time status of the program, run time statistics are tracked, indicating how many times each node has run. By comparing the current snapshot of the run time statistics with the previous snapshot of run time statistics, it is found whether a node is running and/or has finished running. If a node doesn't have run time information, then it didn't start executing. Providing this information to the users would help them in localizing errors in the program because a node wouldn't run if it did not receive data from the node to which it is connected.

This feature also detects new components added to the program and the status gets updated. Hovering over the highlighted area will indicate the status of the node and how many times the node has executed if it has run. Hitting 'Stop' button will clear the indicators.

If Trace is chosen by the user, this preference gets saved in the local storage and it is referred to whenever there is page refresh/reload. The runtime statistics are retrieved by making an AJAX call to the server for sending the run time statistics. These statistics are automatically collected by the runtime environment that executes JavaScript generated by the compiler [9]. A JSON message is sent back to the programming tool, which the ‘Trace’ feature uses for highlighting nodes.

Below are some examples that indicate run time status that helps with identifying errors in the program, again based on my own experience learning the language.

3.2.1 Browser not loaded

The below program in Figure 10 subtracts two numbers and prints the result. All the nodes are highlighted in red. This is because to trigger the event, the page where results can be viewed (Dashboard) should be opened. Since the page wasn't loaded, the event wasn't triggered and the nodes weren't executed.

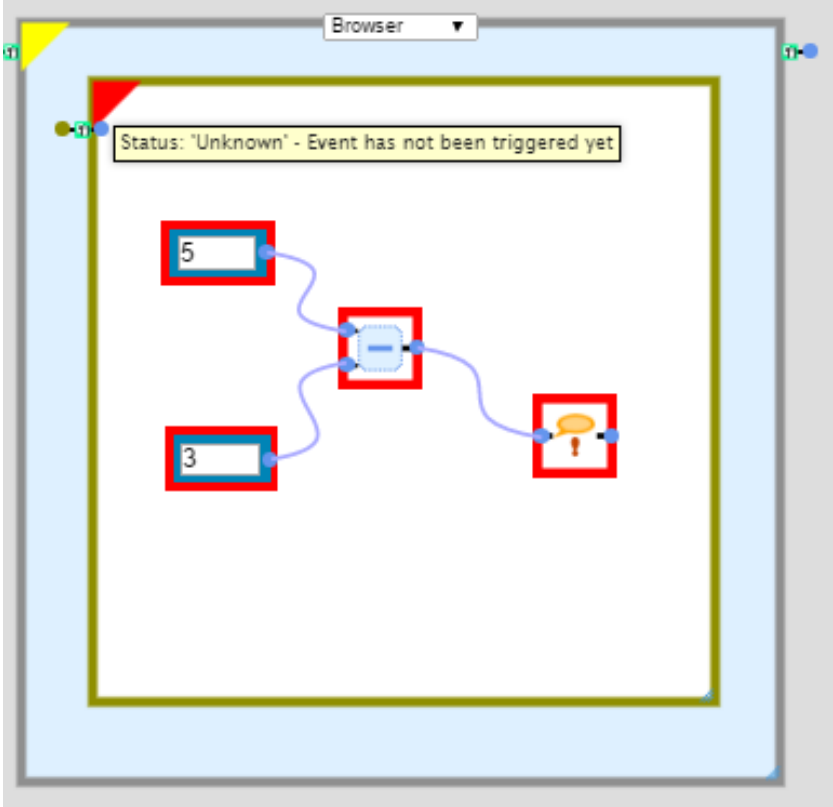

Figure 10 Program with unloaded browser/dashboard

3.2.2 Print message based on result from infinite loop

The intention of the program in Figure 11 is to multiply a random number and index of the loop and print the result. The while loop and components inside it are running which is indicated in orange. However, the Error dialog is highlighted in red. This implies that the error dialog did not receive any data. This is because the loop is infinite and the error dialog is outside the loop, so the Error dialog will not receive value. So either the error dialog has to be inside the loop or stop criteria should be specified for the loop.

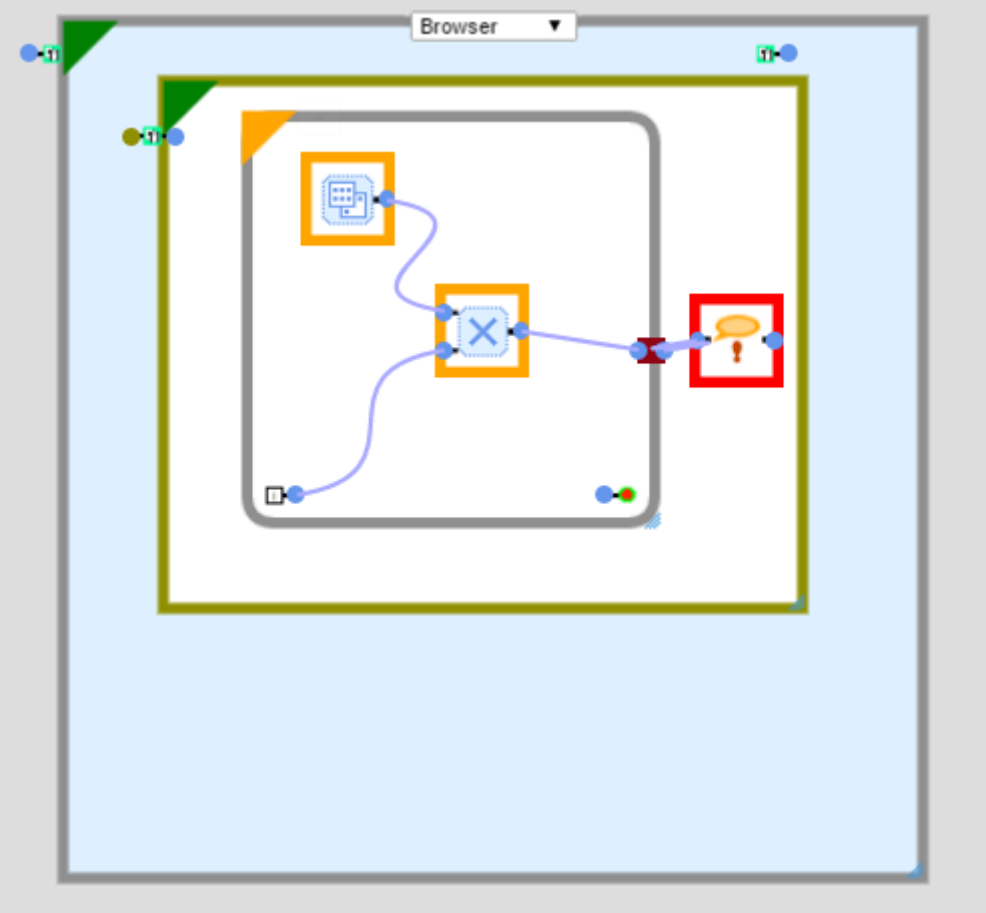

Figure 11 Program with print message outside infinite loop

3.2.3 Fire Event outside infinite loop

In the program shown in Figure 12, every 3 seconds a random number is sent to Browser upon receiving request where it gets displayed. Some nodes are highlighted in orange indicating they are running and some are highlighted in red indicating they haven't executed.

The fire event is highlighted in red which means it has not received any data because of which the event structure in Browser has not been triggered. Similar to the previous example, the loop is infinite so the fire event will never receive any data as the loop would not terminate. Therefore, the fire event has to be moved inside the loop so that every 3 seconds the trigger happens. Or, the loop should have a stop criteria which will send the final value to browser.

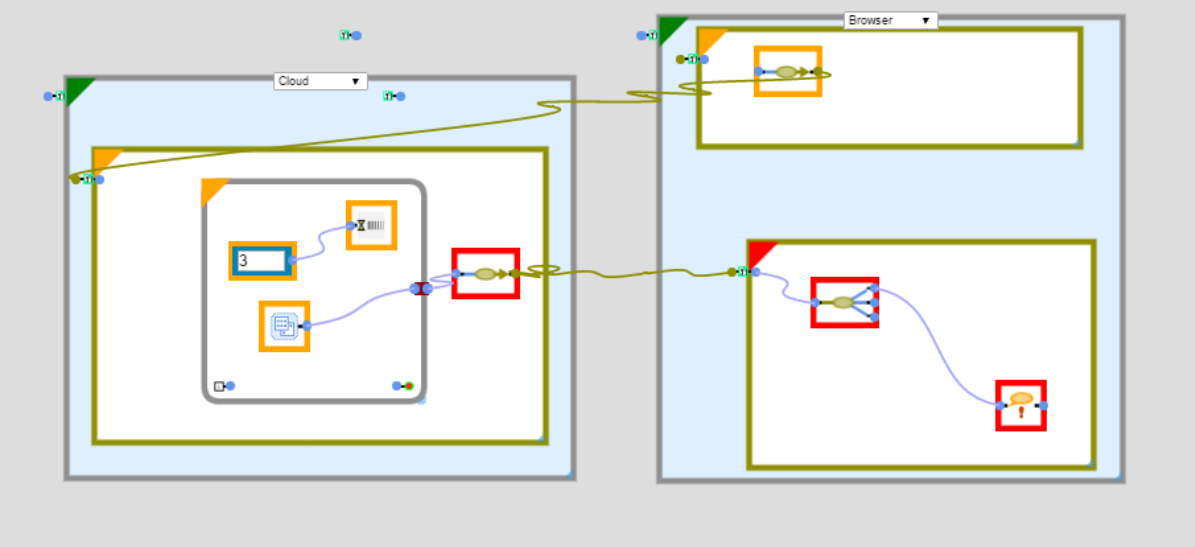

Figure 12 Program with fire event outside infinite loop

3.2.4 Invalid index

In this program, shown in Figure 13, an array is created and the value of the array at a particular index is retrieved and printed. The error dialog is highlighted in red which means it has not received any data. On backtracking, it can be noticed that the index value specified is invalid as the size of the array is 21.

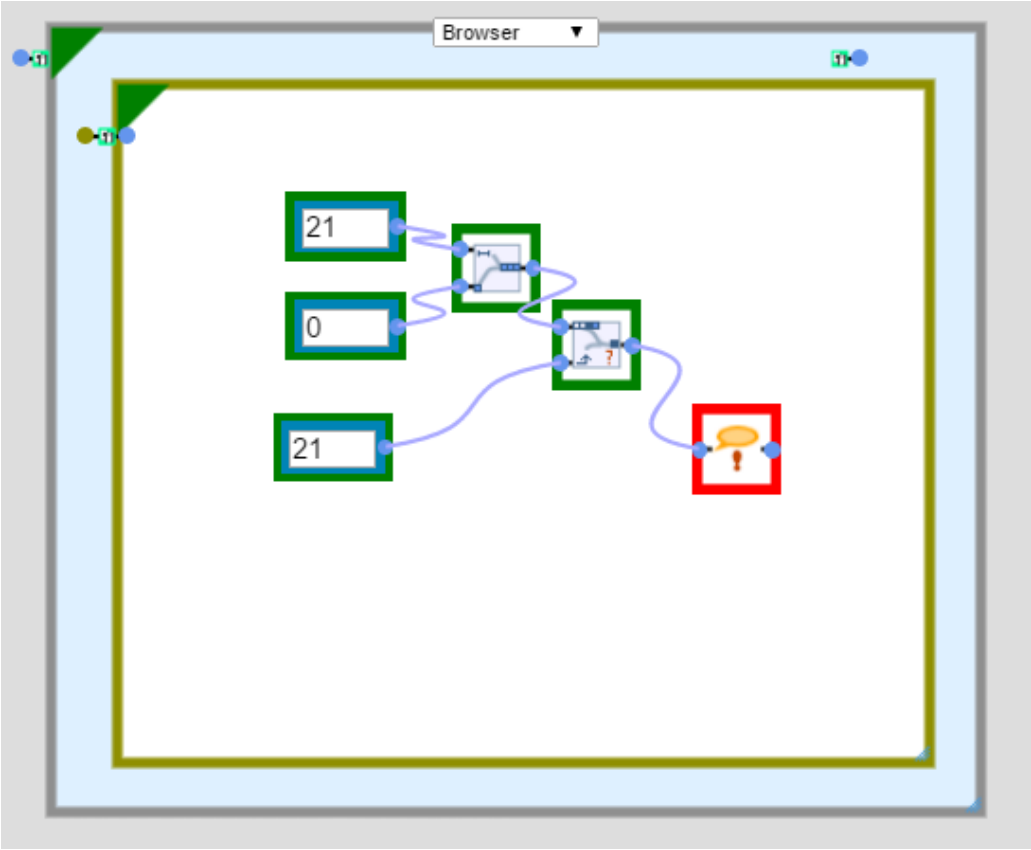

Figure 13 Program with invalid index

3.2.5 Invalid variant property

The intent of this program in Figure 14 is to send a GPS value to browser upon receiving request from browser where the value of latitude is retrieved and printed. This program could be part of a larger application of the type LondonTube was intended to help solve [9] for enabling epidemiologists to correlate location and exposure to airborne particulates. In the example program here, the error dialog is highlighted in red which indicates that it hasn't received any data. On backtracking, it can be noticed that the value of variant property is incorrect. ('Latitude' instead of 'latitude')

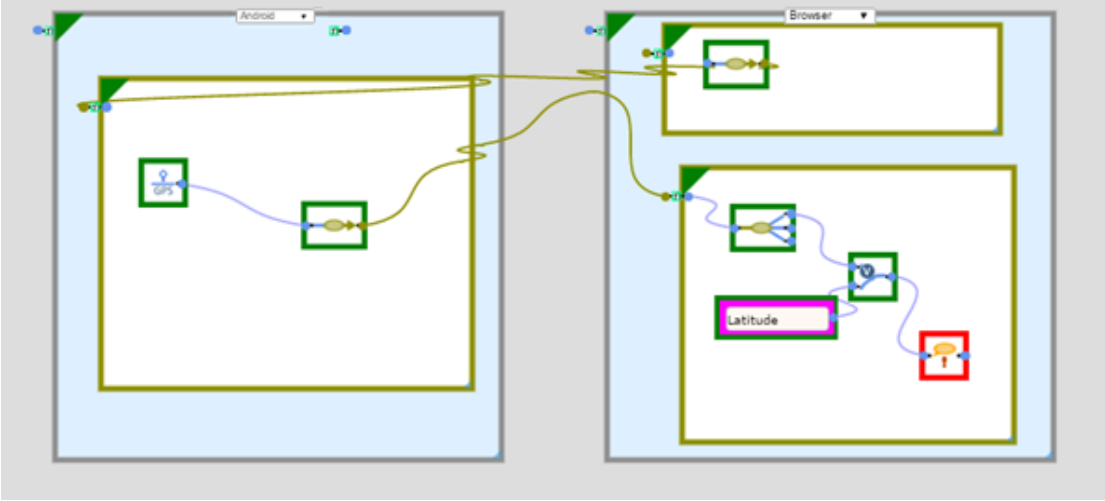

Figure 14 Program with incorrect variant property

3.2.6 Z-Index order

The widgets dropped on top of one another are stacked in such a way that the recently dropped widget is on top. In this example, shown in Figure 15, the random number and Write to UI widgets are dropped and a while loop is dropped over it. The loop does not contain the nodes but it appears as if it contains them. The loop is highlighted in orange because it is still running. The random and Write to UI are highlighted in green because they are not inside the loop and have just run once and finished executing.

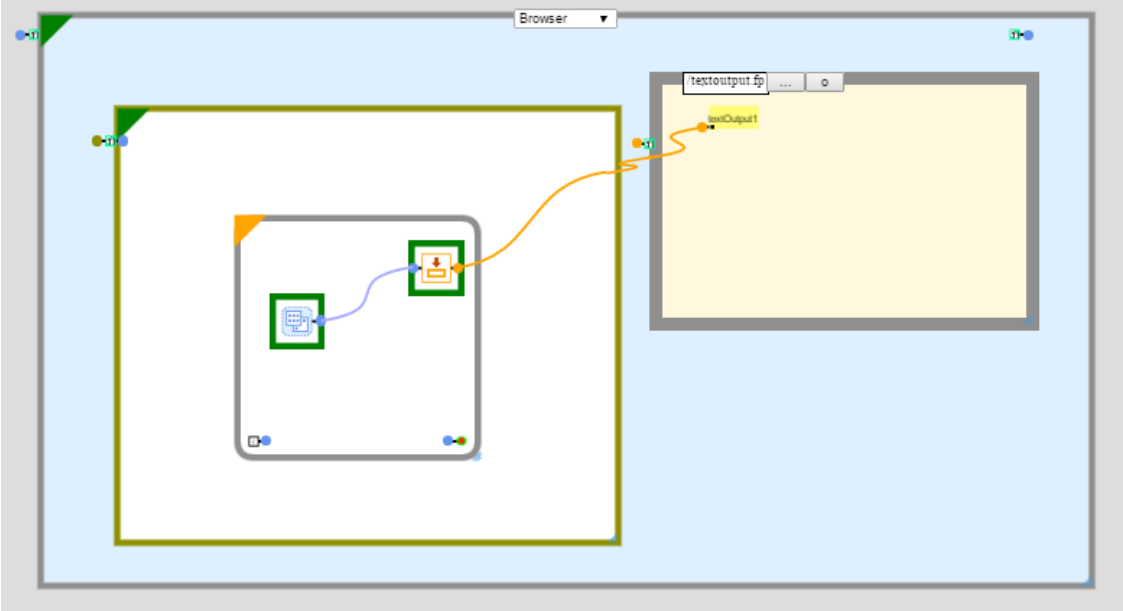

Figure 15 Program with while loop at the front of Z-Index

3.2.7 Program edit during execution

The program in Figure 16 adds up the indices of a loop. While the program is executing, new widgets could be added by the user to the program. In order to indicate that these newly added widgets are not a part of the running instance, they are highlighted in yellow.

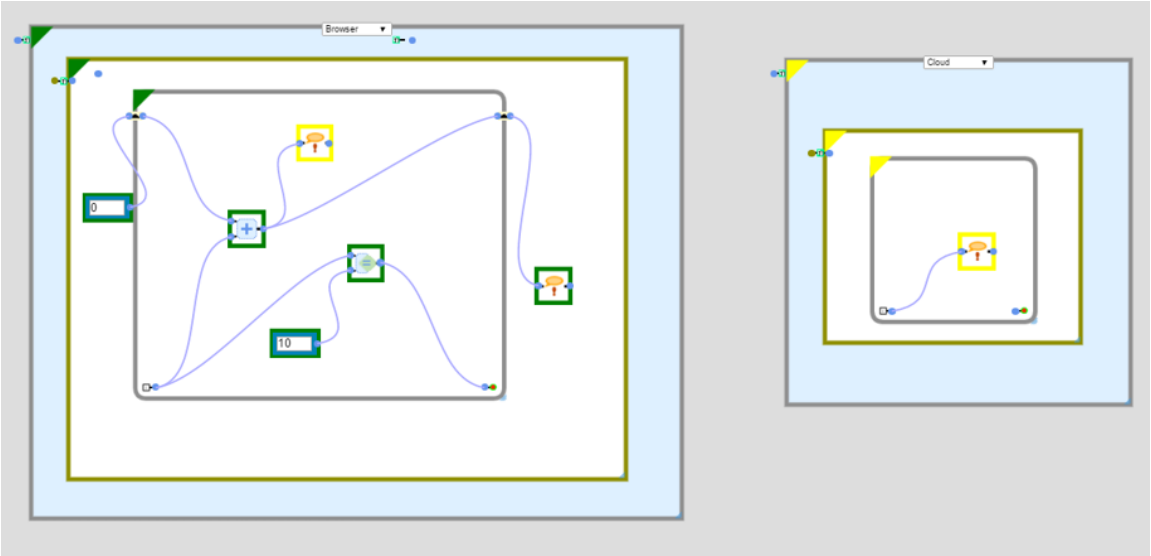

Figure 16 Program with new nodes added while it is running

3.2.8 Unregistered Android device

The program in Figure 17 sends accelerometer value every second to cloud where there is a check whether the value is greater than a specified constant. The result is sent back to Android where it gets displayed. The nodes are highlighted in red because the Android device is not registered. Therefore, the code specified in Android device structure doesn't execute since the event will never be triggered.

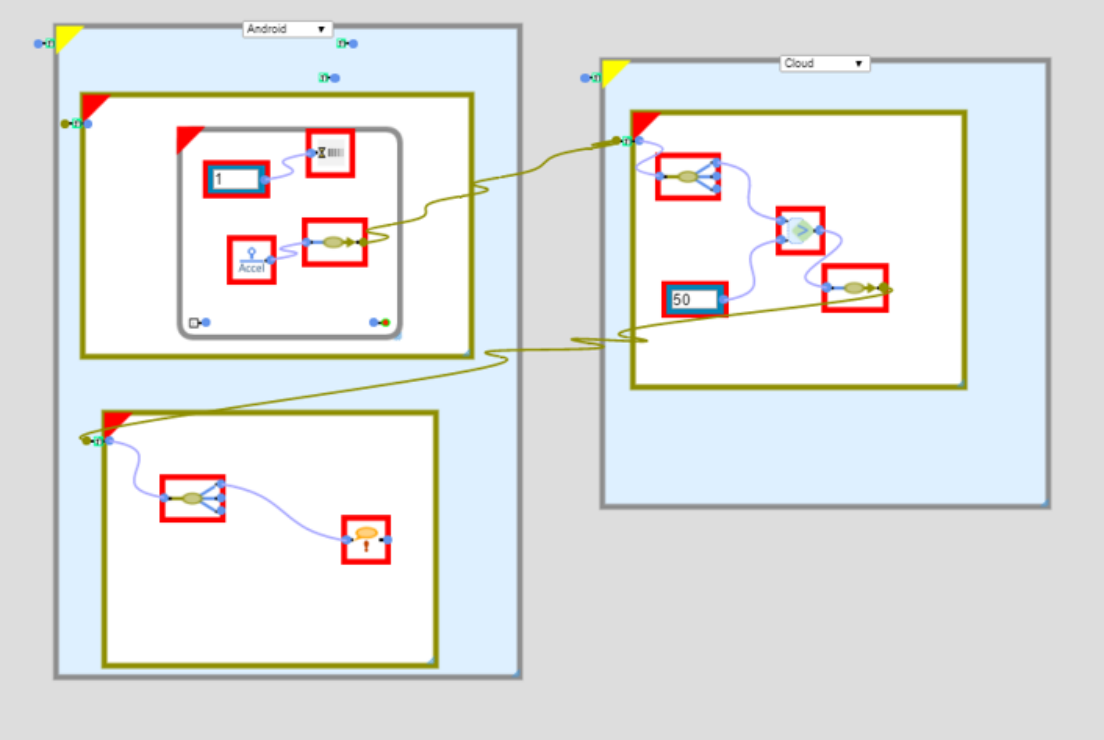

Figure 17 Program with unregistered Android device

3.3 Implementation

Figure 18 depicts the high-level design of the changes.

When Analyzer is turned on by the user, the program available in the form of a tree structure is traversed through and static analysis is performed for each element in the tree. Based on the results of static analysis, the nodes are highlighted. When Analyzer is turned off, the indication is cleared.

When Trace is turned on by the user, program runs and current runtime statistics is compared to previous runtime statistics to determine the status of nodes. The current program structure is compared with the saved back panel program (‘.bp’) to determine the nodes that have been newly added and not a part of running instance. The nodes are then highlighted in a specific color. When Trace is turned off, the indication is cleared.

The ‘analyze’ and ‘trace’ flags are stored in local storage to know what option was chosen by user prior to page refresh. Once page get reloaded, the values of these flags are referred to and the appropriate action is taken i.e. if user had chosen ‘Analyze’ prior to page refresh, then the static analysis would happen and the nodes would be highlighted.

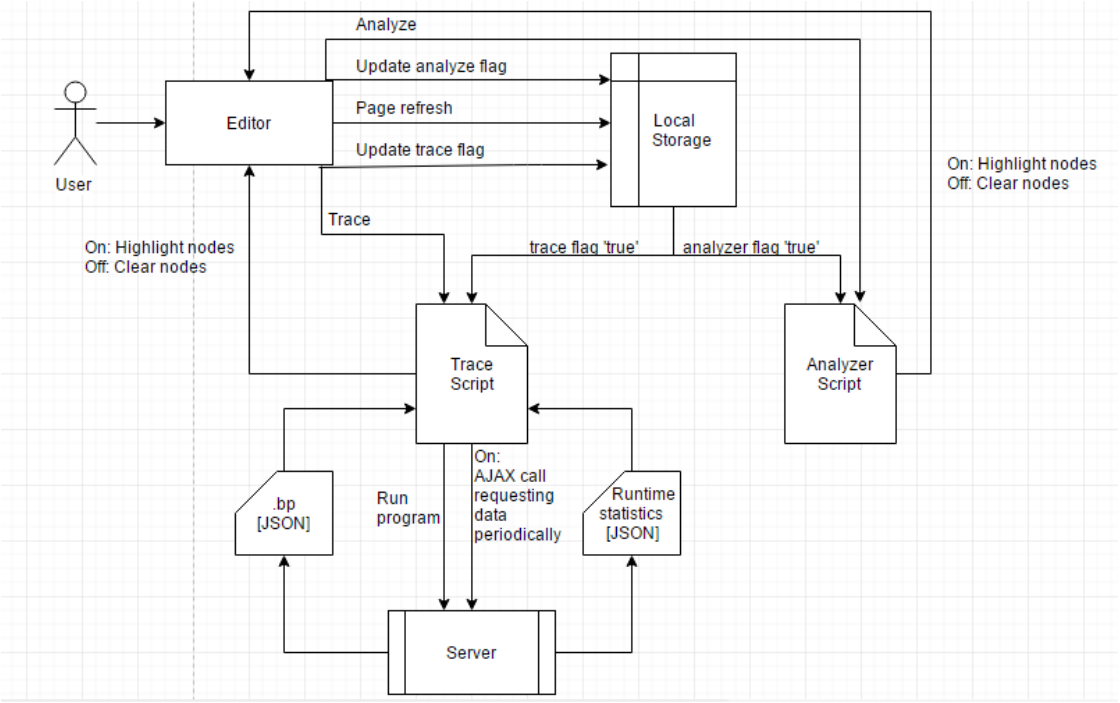

Figure 18 High level design of 'Analyze' and 'Trace'

4 Evaluation

A controlled laboratory experiment was conducted to evaluate the effectiveness of the prototype with new features added to the tool. The experiment involved two groups, participants in one having access to debugging features and another without access to the debugging features.

The specific research questions of this study were,

RQ1: Did access to the new features improve user satisfaction?

RQ2: How satisfied, overall, were participants with the usability and learnability of the new debugging features?

RQ3: Did the new features help users to understand LondonTube?

RQ4: How much faster were the participants with access to debugging features?

The between-subjects study was conducted across different sessions with 8 participants in total. Each participant was randomly assigned to implement the task using the new debugging features or using the baseline tool. A total of five sessions were conducted, with a total of four participants in each condition:

- Session 1: One participant without debugging features (Control)
- Session 2: Two participants with debugging features (Treatment)
- Session 3: One participant without debugging features (Control)
- Session 4: Two participants without debugging features (Control)
- Session 5: Two participants with debugging features (Treatment)

4.1 Methodology

4.1.1 Participant Recruitment

The recruitment criteria for the study was to consider students (18 years or older in age) who weren't professional programmers. The study was advertised via email, with study description, along with IRB consent form for the students to review and a link to sign up for the study. Interested students filled up the form and submitted. The responses for the timeslots were reviewed and the participants were contacted to confirm their availability for the study. Participants were not informed of the group they would belong to.

4.1.2 Procedure

On the day of the study, the participants were assigned to a machine in the lab and were given IRB consent form. The participants were also made aware of the screen recorder that would be running in their machines while they perform the programming task. Once the form was signed by all participants in the session, the study commenced. The planned agenda for the session was shared with the participants followed by a brief introduction to the study and the tool that was going to be used by the participants to implement tasks.

The duration of each session was a maximum of 8 hours including breaks: 3 hours for tutorial and 4 hours for actual programming task. The implementation of the programming task was captured using a screen recording software 'Camtasia Studio' [31]. At the end of the session, participants filled out a feedback form. Questions asked

during the session and/or any verbal response were noted down. Each participant was paid $200 and signature receipt was obtained.

4.1.3 Tutorial

The primary purpose of the tutorial used in the study was to educate the participants on how to create programs using LondonTube tool. The tutorial consisted of 13 tasks with increasing levels of difficulty starting from ‘how to print a string’ to ‘how to create mobile-cloud-web app’. For each task in the tutorial, the goal of the task was explained along with how the program should be created and how it would work. Participants then went over the material for that particular task and created the program. Any questions asked were clarified. Participants in both control and treatment group were given this tutorial.

The tutorial had the following tasks,

- Printing a string
- Looping
- Monitoring sensors
- Computing sum using shift registers
- Using comparison operators
- Understanding Boolean operations and Concatenation
- Array Handling
- Firing events
- Variant Handling

- Storing and loading data
- Displaying data on Browser
- Programming a Mobile-Cloud-Web system

Participants in the treatment group received an additional tutorial on how to use the debugging features ‘Analyze’ and ‘Trace’. This tutorial focused more on usage and purpose of the feature through sample tasks and examples.

Regardless of treatment/control condition, all the necessary information needed for the participants to implement the actual task were covered in the tutorial.

4.1.4 Task

The goal of the study was to evaluate how useful the new features would be for the programmers while they implement a task using this tool. The likelihood of using the debugging feature would be high when programmers run into issues. An easy task decreases the chances of a programmer to run into issues. The paper on LondonTube tool [9] described a task that was completed in 8-9 minutes by all participants using this tool; the task in that paper was intentionally kept simple in order to provide a first investigation of LondonTube’s usefulness. (It is worth noting, however, that the control participants in that study still weren’t able to complete that task in 4 hours using traditional textual programming languages.) Therefore, the task for the current study clearly needed to be more difficult, and it was designed to be challenging enough that some but not all participants could complete the task in the allotted time. Moreover, it

was selected based on expected goals of health scientists. The entire task was divided into 4 subtasks and a description to implement certain rules was included.

The brief description of the task given to participants was as follows:

> "Device should continuously send temperature to cloud which must be saved in a database. Browser should send a request to cloud periodically. When browser sends a request to cloud, cloud should compare the latest temperature with average (excluding latest temperature) and a predetermined constant. Then on browser, latest temperature must be displayed on a gauge, a message whether the latest temperature is within range or not should be shown in text box output. Cloud should also get current location of the device and send it to browser to be displayed on a map along with latitude/longitude values in a text box output."

In addition, the actual task sheet included more specifics like the desired user interface, timer details, and other instructions. For questions asked by the participants about how to complete the task, direct answers were not provided, all participants were asked to refer to the material covered during tutorial. For questions asked about bugs in the programming tool itself (including, for example, some bugs in how the tool handles drag and drop operations), answers were provided.

4.1.5 Feedback

The participants were asked to provide feedback by navigating to a feedback form via the tool editor. The survey included 10 Likert scale questions [32] concerning usability for debugging, as described in detail below. It also included a paragraph type question to provide suggestions. Participants in treatment group had additional 3 questions on their overall perception of the new debugging features.

4.2 Measures and Analysis

In order to understand the effectiveness of the prototype and to find answers for the research questions, data was collected through the following methods.

4.2.1 Screen recording

The programming task was recorded. Initially OBS Studio [33] was planned to be used. However, it caused performance issues i.e. slowed down other running applications. Therefore, Camtasia Studio [31] was used for screen recording. The screen recording was done in order to measure the time taken by a participant to complete a task, to identify the troubles that participants faced while implementing task when they weren't able to finish the task. The screen recording made it possible to compute the total time used by each participant on the task, as well as to assess whether the participant actually completed the programming task.

4.2.2 Feedback form

The feedback form had Likert scale questions [32]. For example, the choices for some of the questions were Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree. Each answer was associated with a point scale ranging from 0 to 4. The questions are listed in Table 3.

Question	Likert Scale
1. It is easy to tell what endpoints/terminals of a node(widget/component) are mandatory/optional.	0 – Strongly Disagree 4 – Strongly Agree
2. It is easy to tell whether a program has static errors. (for example, a mandatory endpoint/terminal of a node NOT being wired up/connected)	0 – Strongly Disagree 4 – Strongly Agree
3. It is easy to locate and fix static errors in the program.	0 – Strongly Disagree 4 – Strongly Agree
4. Once the program starts running, it is easy to tell whether the program has errors.	0 – Strongly Disagree 4 – Strongly Agree
5. Once the program starts running, it is easy to identify/find what components of the program are running, not running and/or the ones that finished running.	0 – Strongly Disagree 4 – Strongly Agree
6. While editing a program that is already running, it is easy to identify/recognize new changes made to the program from the original code that is already running.	0 – Strongly Disagree 4 – Strongly Agree
7. In general, it is easy to localize errors in the program.	0 – Strongly Disagree 4 – Strongly Agree
8. Rate the overall difficulty to debug a program when it does not produce intended results.	0 – Extremely Easy 4 – Extremely Difficult
9. Rate the overall feedback that the system provides that helps with debugging.	0 – Extremely Poor 4 – Extremely Good
10. Rate the overall speed with which you were able to fix errors in your code towards accomplishing the task.	0 – Extremely Poor 4 – Extremely Good

Table 3 Feedback form questions for control and treatment groups

The questions were initially intended to be combined into three scales. However, the Cronbach's alpha [34] for 2 out of 3 scales were not in the acceptable range. Therefore, the questions were independently analyzed.

Specifically, as in the prior work [9], the average score on each item was calculated based on answers from the control group. Then, each answer of each person in the treatment group was compared to the corresponding per-item average among the control group. We hypothesized that the treatment group answers would generally be higher than the corresponding averages. No statistical tests were performed.

The standard deviation of responses within group for each question (per group, per question) was computed. This was to see how well the participants within a group agreed to each question prompt. By comparing the values, it was noted that question number 10 (“Rate the overall speed with which you were able to fix errors in your code towards accomplishing the task”) had a very high scatter for both control and treatment groups, 1.71 and 1.91 respectively. This high scatter value indicated that question could be unreliable. Also, the participants in one session encountered bugs in the programming tool environment that could interfere with their perceived speed. For these reasons, we eliminated question #10 from the analysis.

Table 4 shows the 3 addition questions, for treatment participants, specific to the prototype.

Additional Questions for Treatment group	**Likert Scale**
Rate the learnability of 'Analyze' and 'Trace' features.	0 – Extremely Poor 4 – Extremely Good
Rate the overall usability of 'Analyze' and 'Trace' features.	0 – Extremely Poor 4 – Extremely Good
Would you prefer to use 'Analyze' and 'Trace' features while programming using this tool?	0 – Strongly Disagree 4 – Strongly Agree

Table 4 Additional feedback form questions for treatment group

4.2.3 Questions asked during the study

The questions asked by the participants were noted in order to support qualitative analysis. The questions asked during the study and the comments made were categorized based on the type they belong to (listed out in

Table 5). Open coding was considered for determining codes for the questions/comments. There was a total of about 174 questions during the entire study including tutorial. A randomly-selected sample of 20% of the data was coded by two researchers independently and achieved 84% agreement. Questions coded as being about program components (PGM) were then further coded in sub-categories, and this sub-code had a 91% agreement.

The computation was done based on Jaccard Index which is the 'Intersection of Codes/Union of Codes') [35]. After that, one of the researchers coded the rest of the questions independently.

Code	Sub Code	Description
ENV	-	Issues faced during runtime. e.g. Not being able to save a program.
EDT	-	Issues face while working in editing area. e.g. Not being able to delete an icon.
NAV	-	Searching for an icon in palette and/or search that involves navigation in the tool. e.g. Where is 'Wait next multiple' icon?
TSK	-	Questions specific to description of programming task. e.g. What exactly needs to be done with GPS component for this task?
PGM		Questions related to a program component. (see subcodes in rows below) e.g. Fire events, Arrays etc.
	PGM-USG	Clarifications on usage of a program component. e.g. Why is it not possible to connect fire event with device structure?
	PGM-CHS	Selecting a widget for implementation. e.g. What icon should I use for combining two strings?
	PGM-ARR	Array specific questions. e.g. Is array indexed from 0 or 1?
	PGM-LOOP	Loop specific questions. e.g. Will while loop stop when stop criteria is true?
	PGM-VAR	Variant specific questions. e.g. Is GPS a variant property?
OTH	-	Generic questions that don't belong to any of the above categories. e.g. Why is the tool called LondonTube?

Table 5 Qualitative codes for questions asked during study

Comments by participants were also recorded, as well as their typed feedback in the survey, but not further analyzed.

4.3 Results and Discussion

4.3.1 Comparison of treatment to control on user satisfaction

The first analysis focused on our first research question.

RQ1: Did access to the new features improve user satisfaction?

There was a great difference between control and treatment groups (one having access to prototype model and other not having access to prototype model). By considering the average scores provided by all participants in control group as baseline, every score of participants in treatment group was compared. Out of a total of 36 responses (4 participants * 9 questions), 26 responses were higher than control group's average.

Feedback form item	**#Treatment participants > Control Group's average**
It is easy to tell what endpoints/terminals of a node(widget/component) are mandatory/optional.	3
It is easy to tell whether a program has static errors. (for example, a mandatory endpoint/terminal of a node NOT being wired up/connected)	4
It is easy to locate and fix static errors in the program.	4
Once the program starts running, it is easy to tell whether the program has errors.	1
Once the program starts running, it is easy to identify/find what components of the program are running, not running and/or the ones that finished running.	3
While editing a program that is already running, it is easy to identify/recognize new changes made to the program from the original code that is already running.	3
In general, it is easy to localize errors in the program.	3
Rate the overall difficulty to debug a program when it does not produce intended results.	2
Rate the overall feedback that the system provides that helps with debugging.	3
Total	26 out of 36

Table 6 Feedback form responses

Out of 9 questions in Table 6, for 7 questions the number of participants' scores that exceeded control group's average is >=3. This evidence shows that the participants in treatment group were more satisfied with the system using the prototype.

The biggest differences between treatment and control were related to the static analysis (i.e., the first three items in the table above), for which the majority of participants had answers higher than those of the control group. Participants also generally gave relatively high scores to the final five items in the table above, which were all related to ease of use, particularly for diagnosing and debugging runtime behavior.

There was one item for which most treatment users gave lower scores than the control users. This item was the statement, "Once the program starts running, it is easy to tell whether the program has errors." The relatively poor scores for the tool on this question might be due to the fact that the Trace feature does not actually indicate directly if a program has an error: it only indicates if part of the program is not running. Thus, the Trace feature imposed an additional step of mental interpretation on the users that the baseline tool did not.

For the additional three questions, listed in Table 7, about the features 'Analyze' and 'Trace', that were presented to treatment group participants, each question had an average score (of all treatment group participants) >=3.25 and each response was at least 3.

Additional Questions for treatment group	Treatment Group's Average
Rate the learnability of 'Analyze' and 'Trace' features.	3.5
Rate the overall usability of 'Analyze' and 'Trace' features.	3.5
Would you prefer to use 'Analyze' and 'Trace' features while programming using this tool?	3.25

Table 7 Feedback form responses concerning overall satisfaction

4.3.2 Overall satisfaction

The second analysis addressed the overall satisfaction of participants in the treatment group, based on their responses to the three questions in

Table 7.

> RQ2: How satisfied, overall, were participants with the usability and learnability of the new debugging features?

Learnability

Figure 19 indicates that half of the participants thought the learnability was extremely good and half of the participants thought the learnability was good. This indicates the participants were satisfied with how learnable the features were. During tutorial, there were minimal questions asked by participants. Some participants tried using 'Analyze' while the program was running and asked why it is disabled and got it clarified during tutorial.

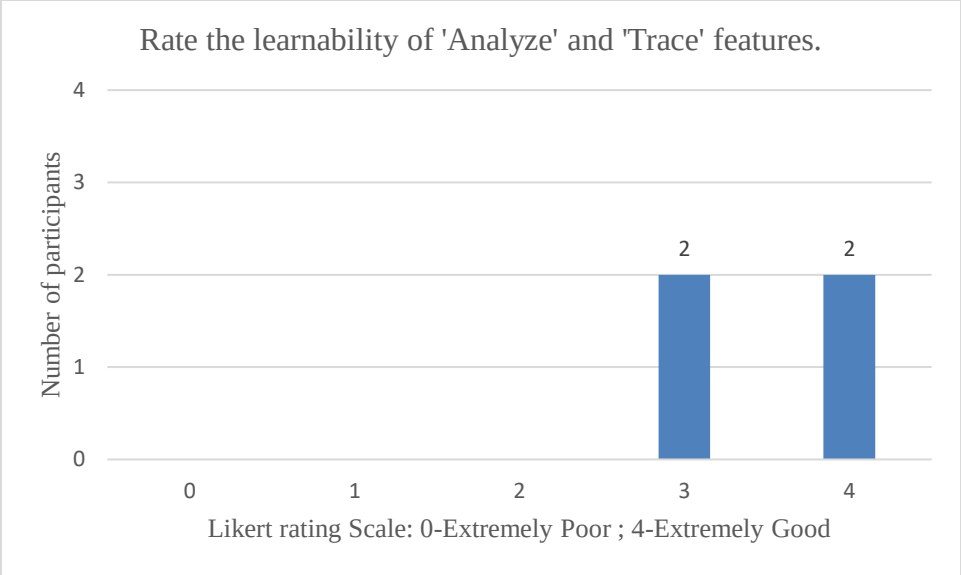

Figure 19 Responses for question on learnability of 'Analyze' and 'Trace'

Usability

Figure 20 indicates that half of the participants thought the usability was extremely good and half of the participants thought the usability was good. Since the responses are on the positive side, this shows the features have good usability.

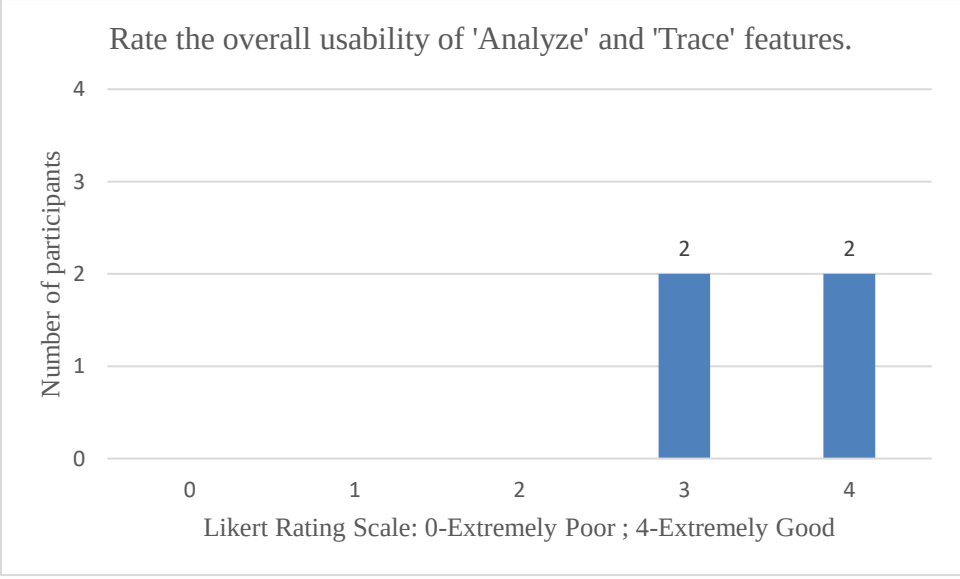

Figure 20 Responses for question on usability of 'Analyze' and 'Trace'

Participants indicated several areas where we could improve usability. For example, one felt when there are lots of nodes in the program, it becomes difficult to hover over the highlighted area, and another participant felt it was difficult to identify which endpoint/terminal had the problem when the node is highlighted in red. Several quotes illustrate these and related concerns:

> *"when there are a lot of elements on the screen, it becomes hard to hover and see the iterations a specific element went through. It would have been better if the popup had some kind of arrow pointing to the element, or some distinct color so it's more obvious"*

> *"As mentioned in one of the earlier questions, in the Analyze feature it might be helpful to highlight the nodes and terminals themselves that are causing static errors (for example, when getting event data only the top of three terminals is mandatory but you can't tell that just from the tools, maybe highlight that terminal red)"*

Preference

Figure 21 depicts that most of the participants agreed that they would prefer to use the features and one of them strongly agreed. This is a good indication that the features have helped the users in debugging.

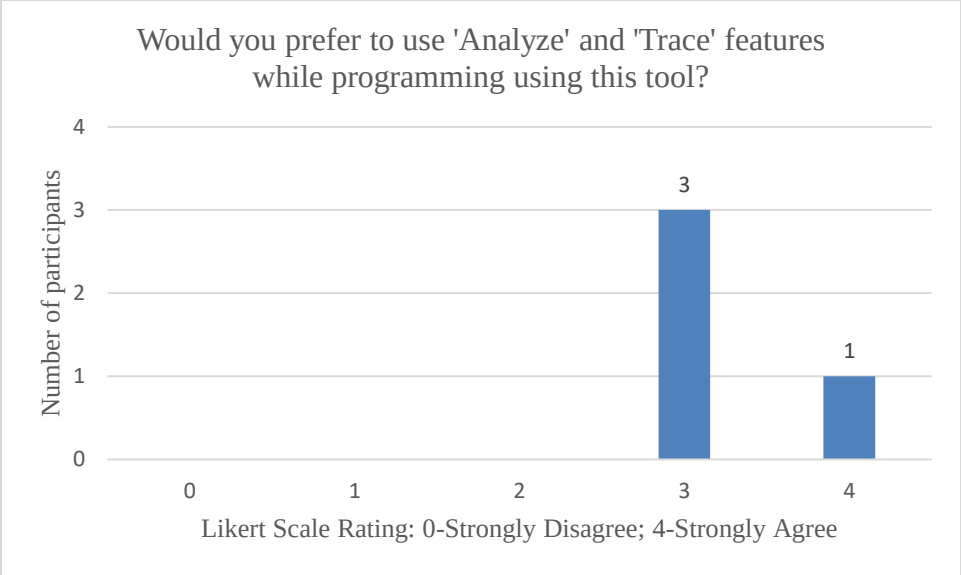

Figure 21 Responses for question on preference to use 'Analyze' and 'Trace'

Some participants thought some additional information about the run time status would help, for example, the previously executed node, why the node did not run, values passed from one node to another. The following quotes illustrate these issues:

"I really liked the trace and analyze options, they are very helpful in learning to use and debug."

"I think that the Analyze/Trace tool worked well but it would have been easier to use if whenever the red boxes appeared during trace (unexecuted nodes) it showed the last node that was successfully executed"

"The trace and analyze are very good and are very helpful while debugging. I would suggest a few more notes about why an event has not run while in trace mode."

"..for the Trace tool, it might be helpful to add on to some nodes the most recent value that was passed through that node, so if the node is running but not in the way you intend (i.e. a comparison operator returning a value you don't expect) you could see what the values were at that point and work back from there."

4.3.3 User understanding

The third analysis examined the new features' impact on users' understanding.

RQ3: Did the new features help users to understand LondonTube?

Analyzing the questions asked during the task (summed across treatment and control) yielded the following distribution (Figure 22).

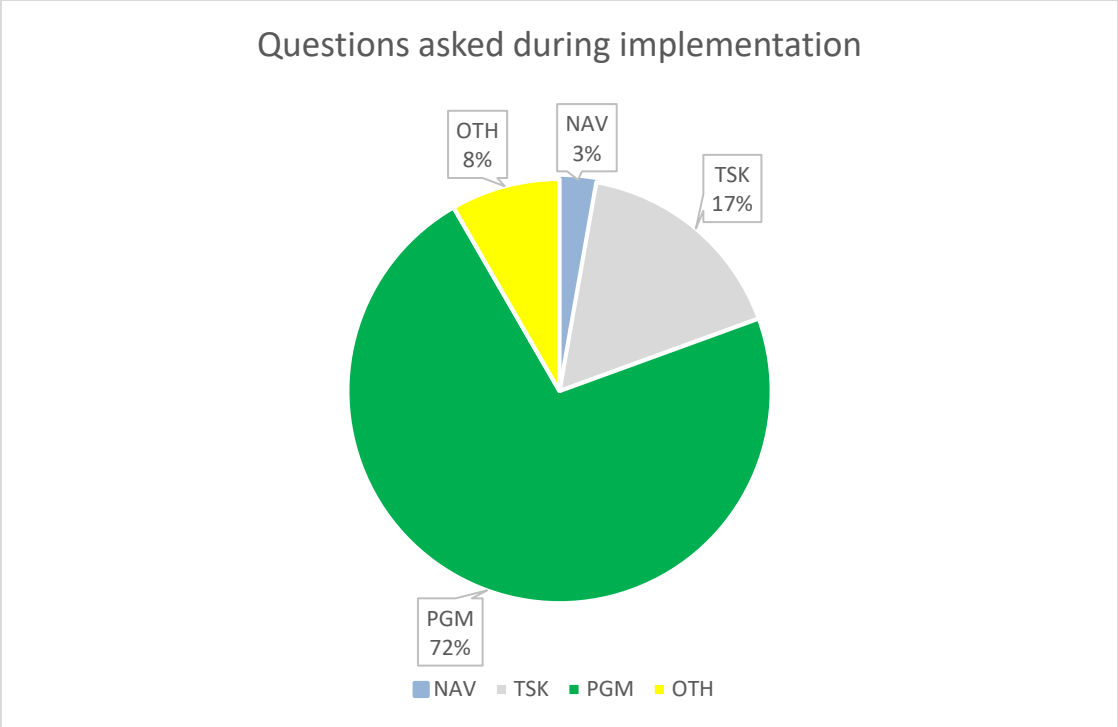

Figure 22 Distribution of questions asked during implementation of the given task

There were questions on the task given for implementation (TSK), searching for a widget and/or feature on tool (NAV) and miscellaneous questions about the tool (OTH). Most of the questions were about program components (PGM) consisting of questions on component's usage (USG), array specific questions (ARR), loop specific questions (LOOP), variant specific questions (VAR) and what widget to use for implementation (CHS).

Figure 23 shows the breakdown of questions, by control and treatment group participants, across the different categories.

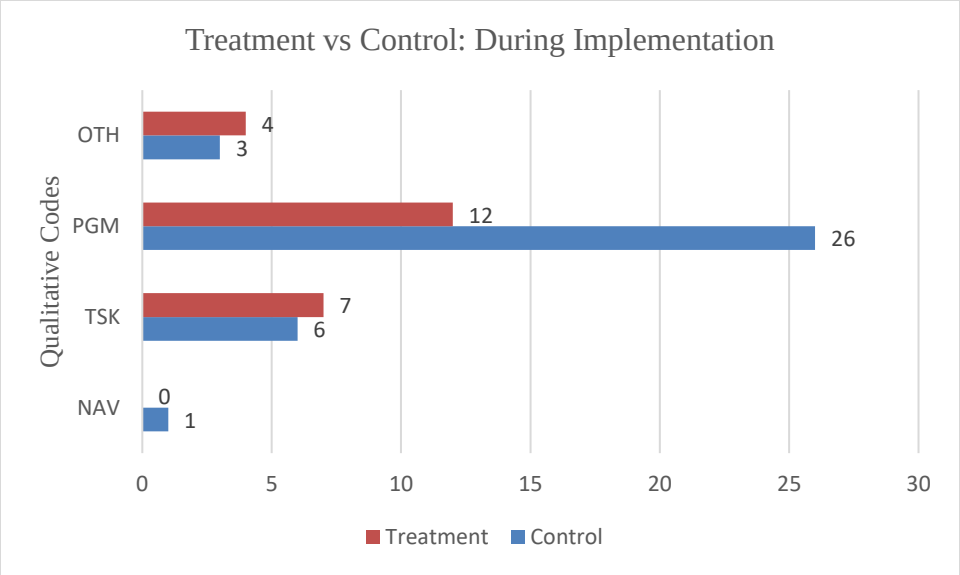

Figure 23 Control vs Treatment group: Questions asked during implementation of the given task

Figure 23 clearly shows that participants in control group had many more questions on how program components must be used, as expected. This indicates that the 'Analyze' feature helped in understanding the usage of program component as to how they need to be connected.

4.3.4 Comparison of programmers' efficiency

Finally, we examined the impact of the debugging tools on overall task time.

RQ4: How much faster were the participants with access to debugging features?

The time taken by each participant for completing the task assigned was recorded. In this analysis, we did not include the time taken by the 4 people who did not complete the task in the allotted 4 hours; in some of these cases, the participants encountered bugs in the programming tool itself, which interfered with their progress, which could have interfered with their speed.

The two treatment users who completed the task required only 70 and 81 minutes, each, compared to the 135 minutes and 141 minutes for the two control users who finished the task. Thus, as anticipated, the programming tool features may have enabled users to complete tasks more quickly. Additional data with more participants would be required to confirm this difference statistically.

5 Limitations

During the study, the questions asked were noted down. There were very few instances when the things observed during the study was also noted down (for example, participants running into an error).

The laboratory study was conducted with 8 participants because of the high compensation that was required for the study. The results were not of statistical significance because of the small size.

One of the control group participants decided to drop out because of not being able to find the error. Two treatment group participants faced some environment specific issues while they were implementing the task. Therefore, the comparison of time taken by participants were made only for those who finished the task. If there had not been any issues faced, then the comparison could have been made across all participants whether they completed the task or not.

6 Conclusion

The goal of this project was to help users to create/debug programs by creating a prototype having two features ‘Analyze’ and ‘Trace’. A laboratory experiment was conducted to see how effective these features are for the end-users in assisting them with creating and debugging programs in LondonTube visual programming tool. Participants having access to the features were faster, had fewer questions about how to use LondonTube, and had higher satisfaction, compared to those with the baseline tool.

Future research could extend the tool developed in this project.

To understand the status of a running program immediately, it would be helpful for the users if a message is shown to indicate the overall status of the executing program i.e. whether the program has errors or not. This could be achieved by showing a list of all errors in the program or a one-line summary of the status.

A clear distinction between runtime and static errors can be shown with the use of different types of icons. These icons can be color-coded based on error severity.

When a node doesn’t execute, it would be helpful to display the nodes that previously executed (similar to stack trace, bottom up tree) which will help in backtracking and fixing the bug.

In order to partially run the program for debugging, ability to set breakpoints and step through the components in the programs can be added. It will also be helpful to show what data gets passed on from one node to another during execution or watch the value received/sent by/from a component.

The ability to detect some issues during static analysis can be added. For example, divide by zero, array index out of bounds, usage of nested loops without timer.

When there are more number of components in the program i.e. when the program is complex and there are some performance issues, it may help programmers if there is a tool that does performance analysis and shows them the performance of their application indicating places where code can be improved.

The questions asked during the study suggests that users have questions on the program components and its usage. It will be helpful to add a Tutorial/Help/FAQ Documentation to the tool that the users may refer to.

Some difficulty was faced around finding widgets on palette. This issue was reported during previous study as well [9]. Most users found it easier to use search bar provided for the webpage. It would be helpful if the tool itself has a search ability for the widgets. Also the widgets can be shown along with icon so that it will be easier to relate.

Overall, these opportunities for improvement would enhance the usability of the tool developed in this project. This could make it easier for health scientists to apply the tool in their everyday research.

7 Bibliography

[1] "Debugging in Visual Studio," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/sc65sadd.aspx.

[2] "How to debug a program in IntelliJ IDEA," 2016. [Online]. Available: https://www.jetbrains.com/help/idea/2016.2/debugging.html.

[3] "Debugging tools in LabVIEW," 2016. [Online]. Available: https://www.ni.com/getting-started/labview-basics/debug.

[4] C. Scaffidi, J. Brandt, M. Burnett, A. Dove and B. & Myers, "SIG: end-user programming," in ACM CHI, Extended Abstracts,, 2012.

[5] C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang and M. B. Rosson, "Supporting end-user debugging: What do users want to know? In Advanced Visual Interfaces, 135–142," 2006.

[6] "Display the relationships between formulas and cells," 2016. [Online]. Available: https://support.office.com/en-us/article/Display-the-relationships-between-formulas-and-cells-a59bef2b-3701-46bf-8ff1-d3518771d507.

[7] "WYSIWYT," [Online]. Available: http://eusesconsortium.org/wysiwyt.php.

[8] "The Whyline," [Online]. Available: https://www.cs.cmu.edu/~NatProg/whyline.html.

[9] C. Scaffidi, A. Dove and T. Nabi, "LondonTube: Overcoming Hidden Dependencies in Cloud-Mobile-Web Programming," in Proceedings of 2016 CHI Conference on Human Factors in Computing Systems, Pages 3498-3508, 2016.

[10] K. Rector, L. Beckwith, C. Kissinger, J. Kaplan, M. Burnett, D. Inman and N. Subrahmaniyan, "Explaining Debugging Strategies to End-User Programmers," in vol. 00, no. , pp. 127-136, 2007, doi:10.1109/VLHCC.2007.18, 2007.

[11] S. K. Kuttal, A. Sarma and G. Rothermel, "Debugging support for end user mashup programming," in In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13), 2013.

[12] "Advanced LabVIEW Debugging: Profiling VI Execution With the LabVIEW Desktop Execution Trace Toolkit," 2012. [Online]. Available: http://www.ni.com/white-paper/8083/en/.

[13] "NI LabVIEW VI Analyzer Toolkit Overview," 2012. [Online]. Available: http://www.ni.com/white-paper/3588/en/.

[14] B. Boe, C. Hill, G. D. M. Len, P. Conrad and D. Franklin, "Hairball: Lint-inspired static analysis of scratch projects.," in In Proceeding of the 44th ACM technical symposium on Computer science education, pages 215–220. ACM, 2013..

[15] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh and K. Stephens, "Improving your software using static analysis to find bugs.," in In Proc. Of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pages 673-674, 2006.

[16] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix and W. Pugh, "Using Static Analysis to Find Bugs.," in IEEE Software, 25(5), Sept 2008..

[17] I. Darwin, Checking C Programs with lint., O'Reilly, 1st edition,, 1988..

[18] N. Truong, P. Roe and P. Bancroft, Static analysis of students' Java programs., In ACE '04: Proceedings of the sixth conference on Australasian

computing education, pages 317–325, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc..

[19] N. Rutar, C. B. Almazan and J. S. Foster, " A comparison of bug finding tools for java.," in In ISSRE ’04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE’04), pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[20] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in ICSE 2005. Proceedings. 27th International Conference on, may 2005, pp. 580 – 586. (Paper=119, Status=F, Phase=3), 2005.

[21] "CDT's Code Analysis (Codan)," [Online]. Available: http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.ptp.pldt.doc.user%2Fhtml%2Fcodan.html.

[22] "Cppcheck, A tool for static C/C++ code analysis," [Online]. Available: http://cppcheck.sourceforge.net.

[23] "IntelliJIDEA Static Code Analysis," [Online]. Available: https://www.jetbrains.com/idea/docs/StaticCodeAnalysis.pdf.

[24] "PyCharm Code Inspection," [Online]. Available: https://www.jetbrains.com/help/pycharm/2016.1/code-inspection.html.

[25] S. P. Reiss, " Visual representations of executing programs,," Journal of Visual Languages and Computing archive, vol. Volume 18 , no. Issue 2, pp. Pages 126-148, April, 2007.

[26] Ko, A. J, Myers and B. A., " Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior.," in In Proc. SIGCHI 2004, vol. 6 (2004)..

[27] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick and M. Burnett, "Interactive, visual fault localization support for end-user programmers," Journal of Visual Languages & Computing, 16(1–2), 3–40, 2005.

[28] K. S.-P. Chang, "Gneiss: spreadsheet programming using structured web service data," Journal of Visual Languages & Computing, http://www.sciencedirect.com/science/article/pii/S1045926X16300994, 2016.

[29] "Chrome DevTools Overview," [Online]. Available: https://developer.chrome.com/devtools.

[30] T. Nabi, C. Scaffidi, D. Piorkowski, M. Burnett and S. Flemming, " A Wiki to Help Tool Designers who Want to Put Information Foraging Theory into Practice," [Online]. Available: http://research.engr.oregonstate.edu/ift/readonly.php.

[31] "Camtasia Studio, Screen recording software," 2016. [Online]. Available: https://www.techsmith.com/camtasia.html.

[32] "Likert Scale, commonly involved in research that employs questionnaires," [Online]. Available: https://en.wikipedia.org/wiki/Likert_scale.

[33] "OBS Studio: Free, open source software for live streaming and recording," 2015. [Online]. Available: https://obsproject.com/forum/resources/categories/obs-studio.6/.

[34] "Cronbach’s Alpha, computes the Cronbach alpha statistics for a set of items that are believed to represent a latent variable (construct)," [Online]. Available: http://www.wessa.net/rwasp_cronbach.wasp#output.

[35] "Jaccard Index, Statistic used for comparing similarity and diversity of sample sets," [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index.