Design

Debugger++: Trace-based Debugging for Java Development Environments

Team 57: Robin Lai, Juntong Luo, Jane Shi, Arya Subramanyam

Client: The Reliable, Secure, and Sustainable Software Lab (ReSeSS)

CPEN 491: Capstone

Dr. Pieter Botman

April 11th, 2023

Table of Contents

List of Figures

1. Overview

In this design document, we first introduce the system context of Debugger++. This tool is based on the Java language and is integrated into one of the most popular integrated development environments, the debugger of IntelliJ IDEA. We also introduce the workflow of Slicer4J, a tool provided to us by our client. Afterward, we researched and analyzed the feasibility, capabilities, and scalability of this tool, and proposed several solutions to some shortcomings. After discussing with our client, we updated the scope of the project and proposed the final design based on the revised scope and new constraints. Next, we show the specific example and scenarios of the user's interaction and experience based on the final design. Finally, based on the final plan discussed with the client, we proposed a high-level design plan and refined the design into small components including Integration with Slicer4J, UI, debugger workflow, etc.

2. System Context

To provide context for the design of Debugger++, this section describes existing systems that Debugger++ relies on and builds upon, namely the Java Platform, IntelliJ IDEA Java Debugger, and Slicer4J. *Figure 1* is a context diagram that illustrates the relationships between these elements.

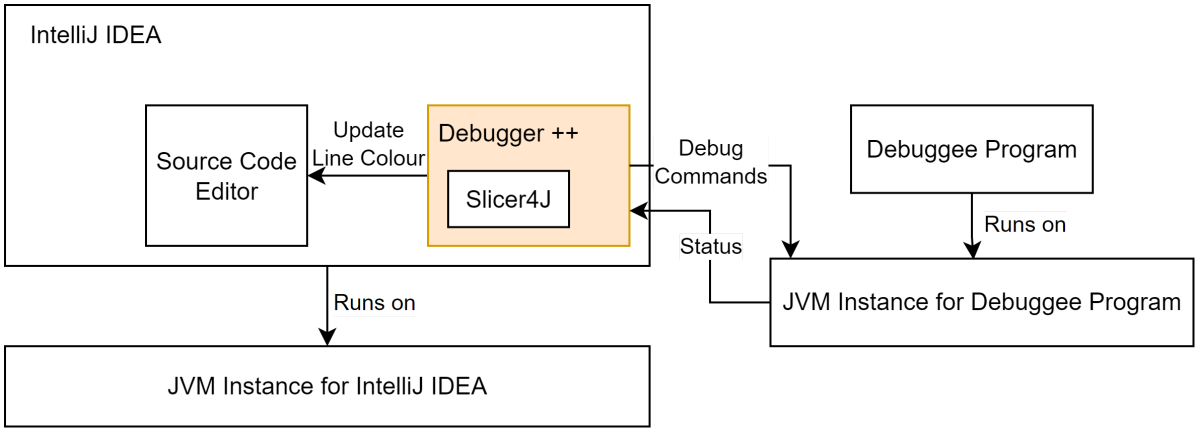

Figure 1: The system context of Debugger++

2.1 The Java Platform

Java is a cross-platform programming language and a software development platform (“Java Software”). A simplified workflow from writing a Java program to running and debugging the Java program is as follows (some of which are handled by an IDE such as IntelliJ IDEA):

1. The developer writes Java source code and saves it as *Java source code files* with the file extension “.java”.
2. A *Java compiler* compiles Java source code files to produce *Java class files*, which have the file extension “.class”. Java class files contain *Java bytecode* that can be executed on Java Virtual Machines (JVM).
    - A Java class file also contains debug information which is required by the debugger (i.e. mappings from bytecode to lines in the source code, mappings from registers to local variable names, and the name of the source file).
3. Optionally, the developer may choose to package Java class files, metadata, and other resources into a Java Archive (JAR) for distribution. JAR files have the file extension “.jar”.
4. A *Java Virtual Machine (JVM)* loads Java classes from JAR or class files and executes the bytecode. A JVM usually contains another compiler that converts Java bytecode to machine code which is executed directly on the computer.
    - The JVM also provides the *Java Virtual Machine Tool Interface (JVM TI)* API, which allows debugging tools, such as the debugger in IntelliJ IDEA, to debug the program running on the JVM. (“Java Virtual Machine Tool Interface (JVM TI)”)
    - The debugger can communicate with the JVM through a TCP socket with the *Java Debug Wire Protocol (JDWP)*. (“Java Debug Wire Protocol”)

2.2 IntelliJ IDEA

IntelliJ IDEA is a comprehensive Integrated Development Environment (IDE) developed by JetBrains for Java and other JVM-based languages (“IntelliJ IDEA – the Leading Java and Kotlin IDE”). It provides a wide range of features to help developers write, debug, and manage their code more efficiently. It is written in Java and Kotlin and runs on a JVM. It is also extensible, as it allows users to write plugins to modify/extend its behaviour.

IntelliJ IDEA includes functionalities such as code editing, code analysis, building, testing, version control, and most relevant to this project, debugging. The following steps describe how a debug command is handled. We do not include references to relevant functions in this subsection as they depend on the type of debug command and some of these functions are not open-source.

1. The user clicks a debug action button.
2. The debugger front end calls the corresponding handler in *JavaDebugProcess*, which then calls other functions to handle this command.
3. The command is sent to the debuggee JVM via the JDWP.
4. The JVM executes the command and returns the result.

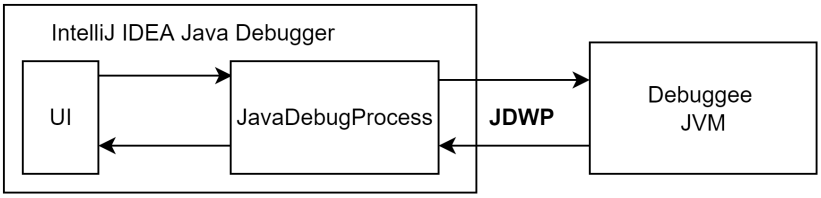

Figure 2: The communication between UI, JavaDebugProcess, and JVM

2.3 Slicer4J

Slicer4J is a dynamic slicer for Java programs (Ahmed et al. 1570). This subsection explains the process of using Slicer4J to obtain a dynamic slice given a JAR. The dynamic slice includes a list of statements and the dependencies between the statements.

1. Instruments the JAR. Slicer4J has a *JavaInstrumenter* class which takes a JAR and outputs a different JAR. It extracts and instruments class files in the JAR and rebuilds a new JAR file. The output JAR contains extra print statements between statements in the original JAR.

```java
public static int greater(int x, int y) {
    if (x >= y) {
        return 1;
    } else {
        return 0;
    }
}
```

```java
public static int greater(int x, int y) {
    DynamicSlicingLogger.println("3");
    if (x >= y) {
        DynamicSlicingLogger.println("4");
        return 1;
    } else {
        DynamicSlicingLogger.println("5");
        return 0;
    }
}
```

Figure 3: The same function before and after instrumentation. Left - the original function. Right - the decompiled code of the instrumented function.

2. Runs the instrumented JAR. The system needs to run the instrumented JAR and record its standard output, where the execution trace is printed to.
3. Slices the program with the trace from the standard output. This is done by the *Slicer* class in Slicer4J.

Figure 4 illustrates this process.

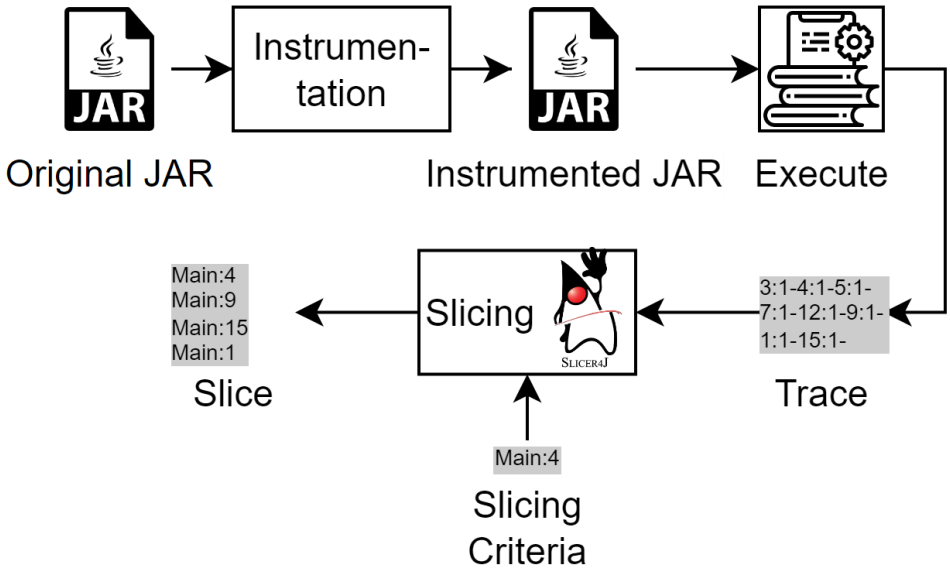

Figure 4: Slice production process within Slicer4J

2.4 Debugger++

Debugger++ is an IntelliJ IDEA plugin which extends the original Java debugger IntelliJ IDEA and provides additional functionalities, aiming to help the user debug more efficiently by

leveraging Slicer4J. It interacts with the debuggee JVM and other components in IntelliJ IDEA (e.g. Source Code Editor) to facilitate debugging.

3. User Experience and Functions

This section walks through the experience of the user utilizing Debugger++, while showcasing images of the user interface. The interaction of the user with the user interface is detailed with each image.

Step 1: Start Slicing from Line

The user starts a debugging session by right clicking on the desired slicing criterion line and selecting the “Start Slicing from Line” option from the pop-up menu.

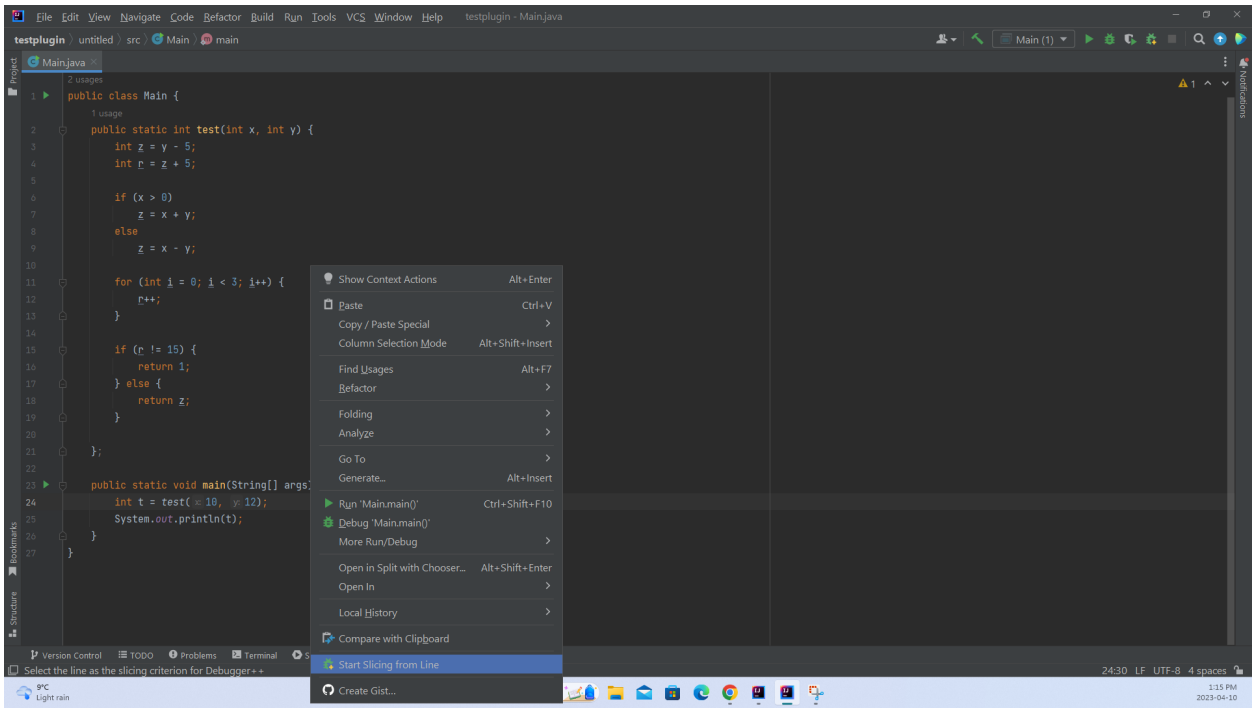

Figure 5: Menu showing the “Start Slicing from Line” Option

Step 2: Debugging Session Begins

The IDE begins executing the dynamic slicing as shown below, with the progress bar.

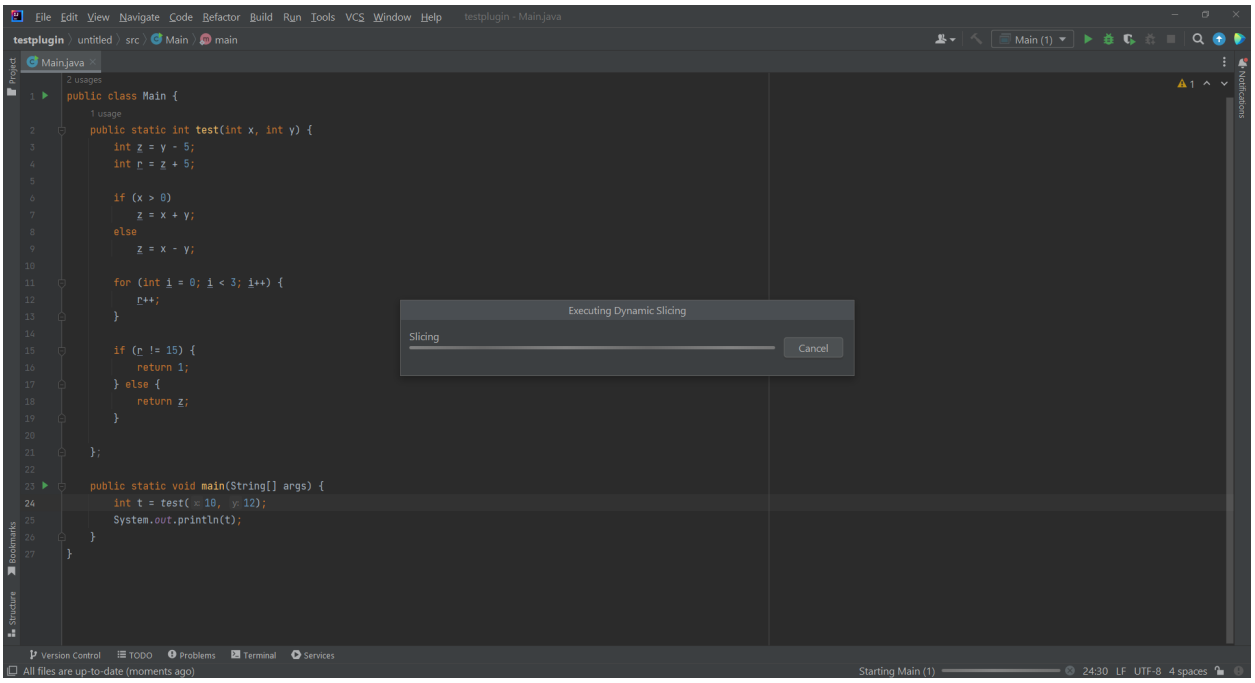

Figure 6: Window shows Dynamic Slice Executor Running

Once the program slice is computed, the IDE starts a new debugging session and grays out statements that are not in the slice, as can be seen in the source code viewer. A new Debugger++ instance opens to control the new debugging session. If there doesn't already exist a breakpoint on the first slice line, it is automatically set.

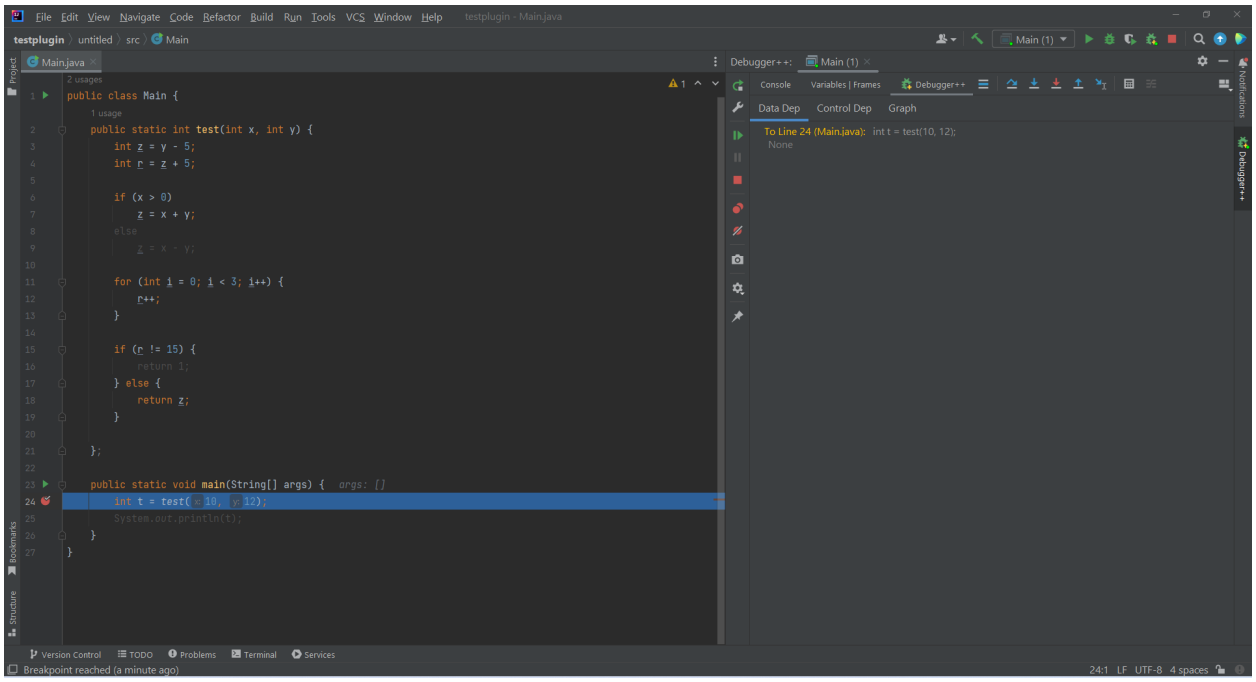

Figure 7: Line graying and new Debugger++ instance

Step 3: Use Debugger Actions

The user can press any of the Debugger++ Actions including Step Over, Step Into, Run to Cursor etc. to execute the program slice. The user will not see any non-slice lines or grayed out lines visited during the debugging session and any breakpoints on these lines will be ignored. Attempted Run To Cursor Action on a non-slice line will give a warning.

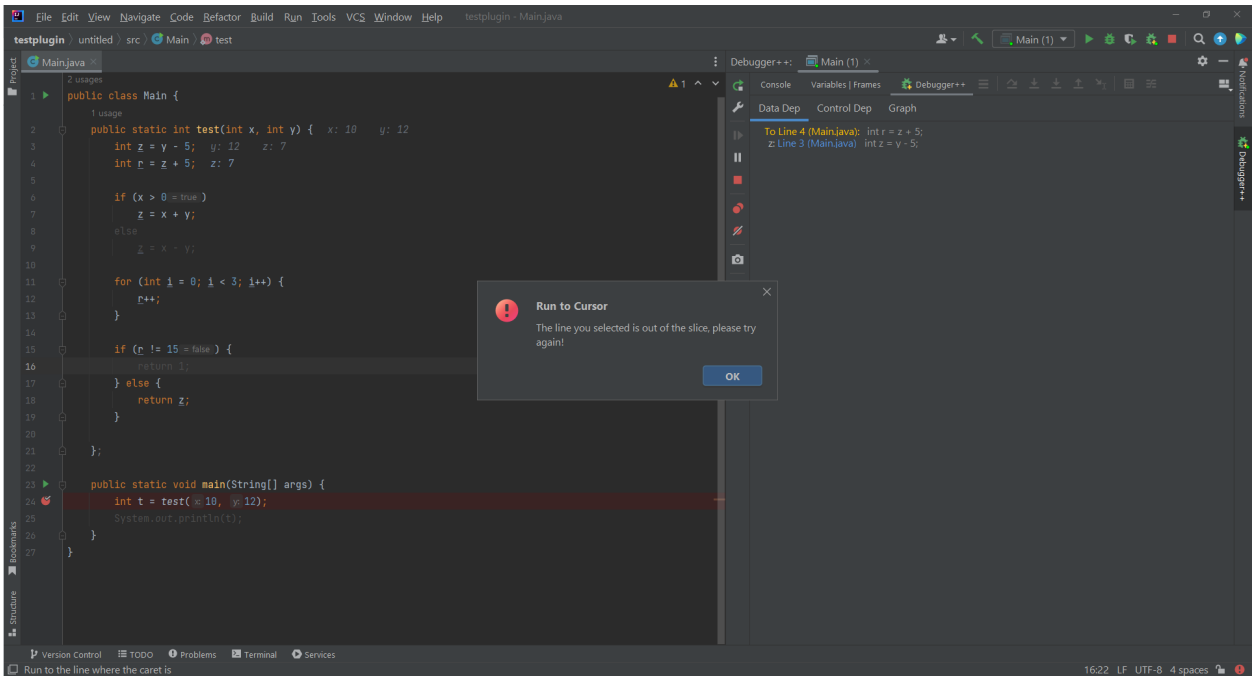

Figure 8: Run To Cursor on Non-Slice Line Warning

Step 4: View Slice Information

The Debugger++ instance has a Slicer4J Tab which includes 3 sub-tabs. The data dependencies, control dependencies, and a graph visualization of the current statement in the slice are shown within these sub-tabs, respectively.

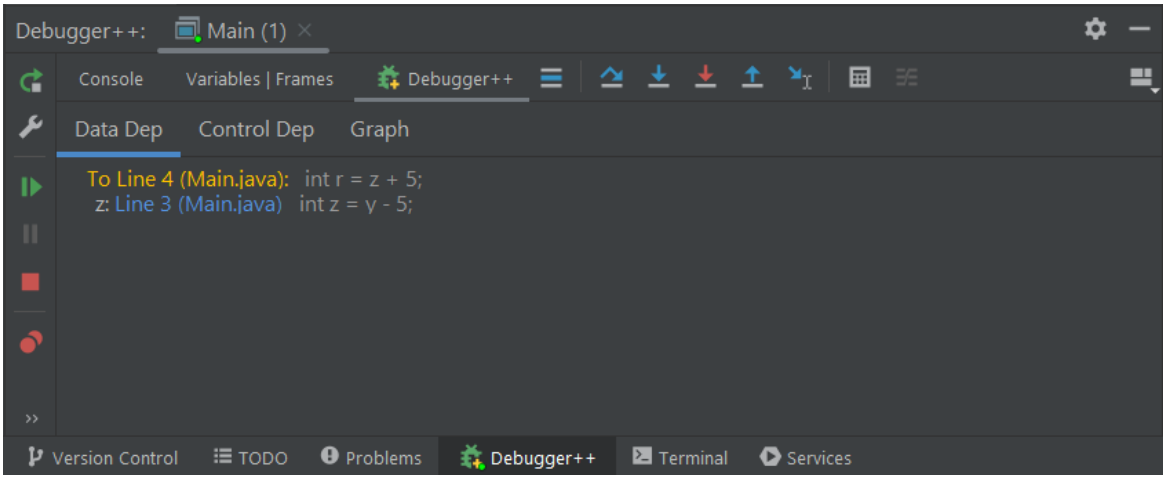

Figure 9: Data Dependencies Tab

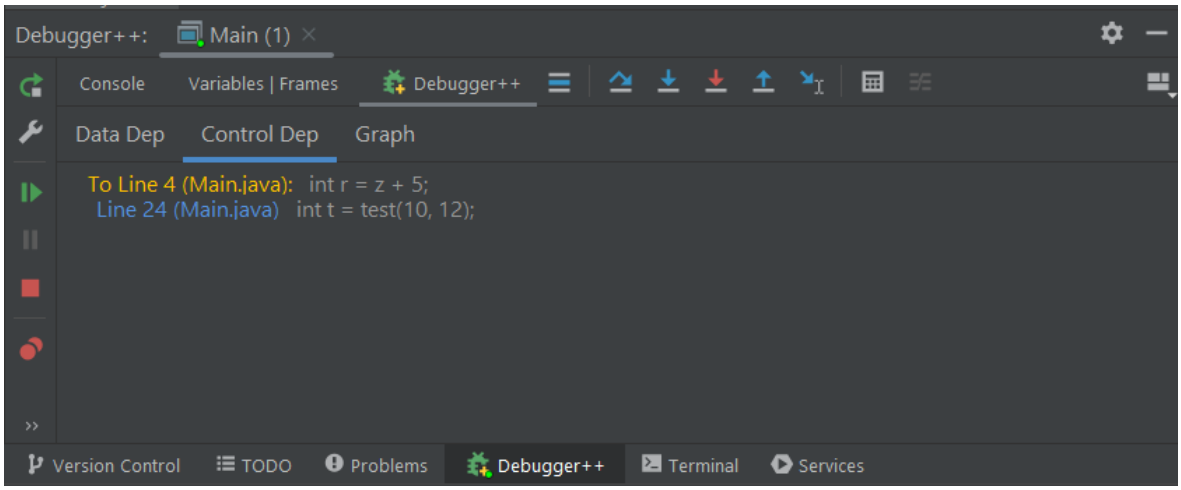

Figure 10: Control Dependencies Tab

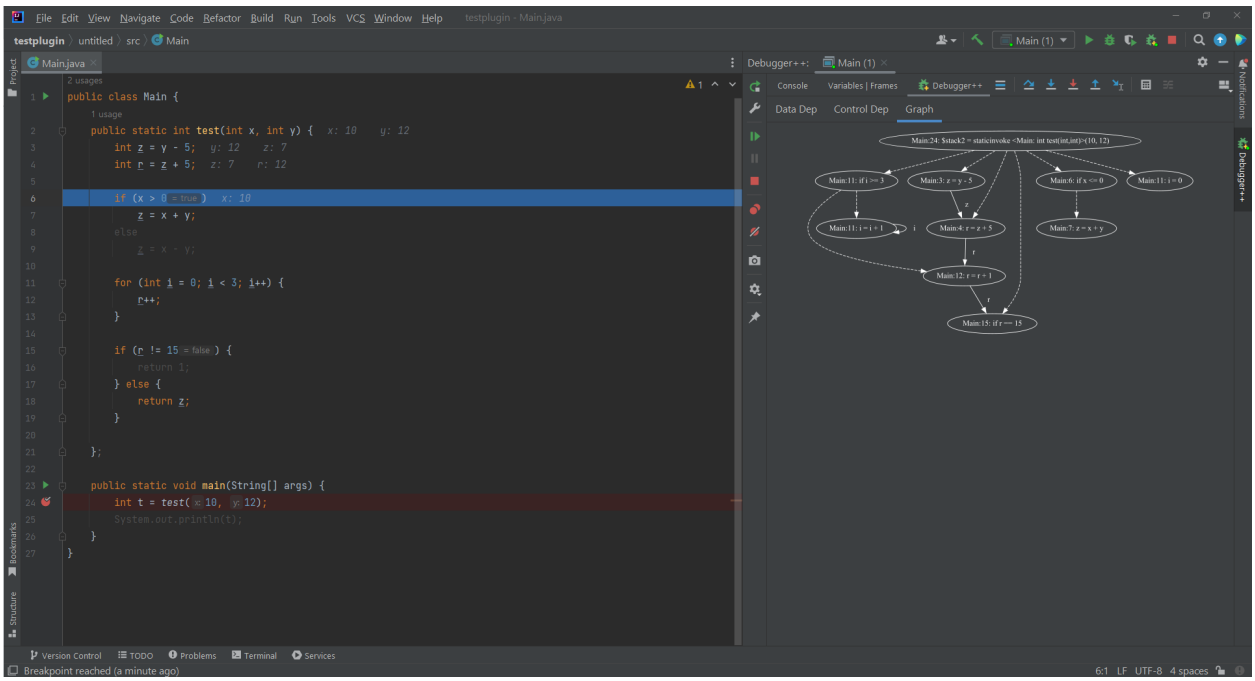

Figure 11: Dependencies Subgraph Tab

4. High-Level Design

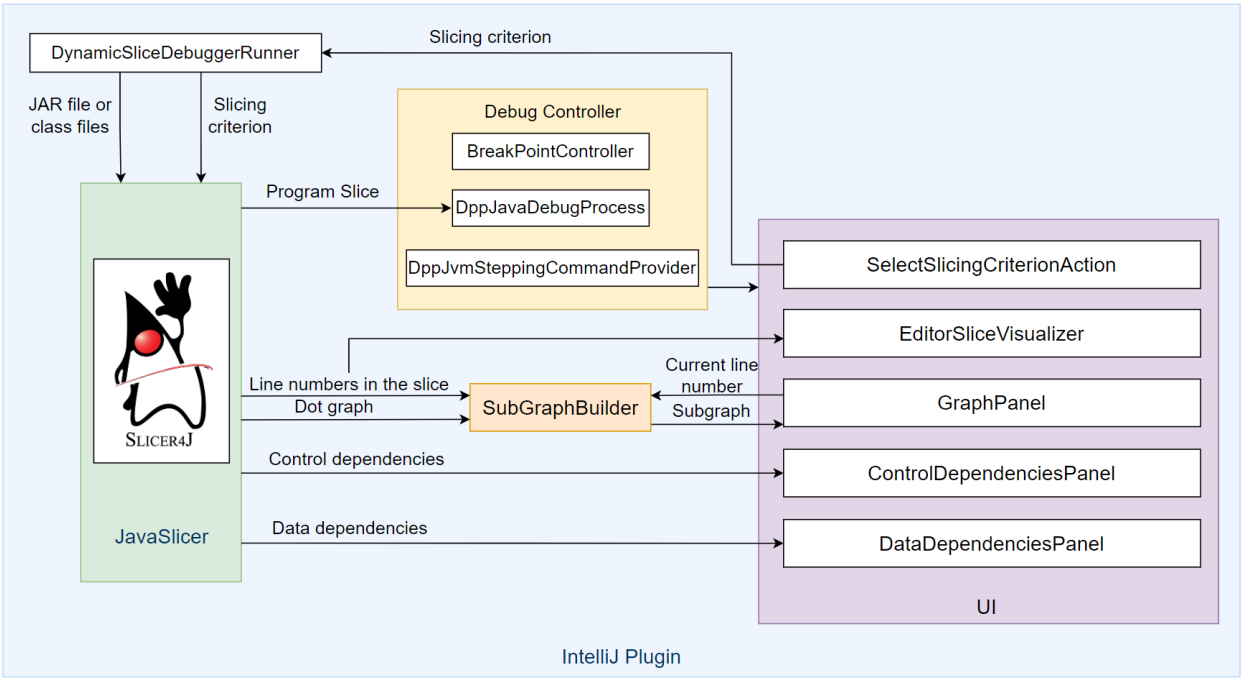

Figure 12: Architectural diagram

The Slicer4J module is provided by the client. It takes as inputs (1) a Java Archive (JAR) file of the program to slice, and (2) a slicing criterion (e.g. a statement that throws an exception) and performs dynamic slicing on the program for the selected slicing criterion. The *JavaSlicer* component acts as a wrapper of the Slicer4J module. It outputs (1) a program slice, (2) a list of line numbers in the slice in the order of execution, (3) a dot graph of the slice with nodes as statements in the slice and edges representing data and control dependencies between the statements, and (4) data and control dependencies for each line in the slice.

The *DynamicSliceDebuggerRunner* is the component that achieves slice production by calling relevant functions from *JavaSlicer* and providing the required user input, including the JAR file and slicing criterion. It also automatically starts a new debugging session after *JavaSlicer* produces the required slice.

The *DebugController* module takes the slice from *JavaSlicer* as an input and uses it to determine which lines to skip when controlling the debug actions, such as “step over” and “step

into", to prevent the user from seeing any non-slice lines being visited during their debugging session. It provides commands which define the behaviour of the stepping buttons. In particular, the *DppJavaDebugProcess* subcomponent is for handling the "Run to Position" button, the *DppJvmSteppingCommandProvider* component provides step into and step over commands that skip non-slice lines, and the *BreakPointController* subcomponent is for adding a breakpoint to a given source code location.

The *UI* module includes various submodules to control various parts of the interface. The *EditorSliceVisualizer sub*module uses the slice produced by *JavaSlicer* to gray out statements that are not in the slice in the source code viewer in IntelliJ IDEA.The *ControlDependenciesPanel* and *DataDependenciesPanel* modules take the dependencies data produced by *JavaSlicer* to display the control and data dependencies for the current line in the debugger. The *GraphPanel sub*module takes the current line number and passes it to the *SubgraphBuilder* component.

The *SubgraphBuilder* component takes the dot graph, the line numbers in the slice, and the current line number as inputs and outputs a subgraph, which is displayed by the *GraphPanel* and updated dynamically as the user steps over each line in the slice.

All the modules mentioned above are created through an IntelliJ Platform Plugin, written in Java or Kotlin.

5. Components Design

5.1 Slicer4J Wrapper

This section describes *JavaSlicer*, the wrapper for Slicer4J.

5.1.1 Slicer4J Integration

We have two options regarding how to integrate Slicer4J into Debugger++:

1. Include Slicer4J as an external dependency. This means including Slicer4J as a separate JAR in our plugin, treating it as a black box, providing inputs through command line arguments, and getting outputs through the log files.
2. Include Slicer4J as an internal dependency. This means bundling Slicer4J code directly with our project, calling functions in Slicer4J and getting outputs directly with Java/Kotlin code. This option requires rewriting Slicer4J's build system and refactoring some functions in Slicer4J to expose some of its internal functionalities.

We chose the second option for three reasons:

1. It is much more performant as we do not need to get results through disk I/Os.
2. It incurs much lower overhead in terms of the amount of code for communicating with Slicer4J.
3. It is more straightforward when we modify Slicer4J. Slicer4J is currently included as a Git subtree in our Debugger++ repository, which means we can make and commit changes to Slicer4J directly in our Debugger++ repository, avoiding the hassle of switching between multiple repositories when working on one functionality.

5.1.2 JavaSlicer

JavaSlicer is a wrapper class of Slicer4J that converts Slicer4J APIs to ones that are more friendly for other components in Debugger++. It currently has the following interfaces:

- *instrument*: takes an *ExecutionEnvironment* from the IDE, extracts the paths to the compiled executables (class files or JAR files), instruments the executables, and returns paths to the instrumented files.
- *collectTrace*: executes the instrumented files returned by *instrument* and collects the trace of execution. Returns the trace.
- *createDynamicControlFlowGraph*: takes the trace and the original program to create a *DynamicControlFlowGraph* which is a graph containing all the potential dependencies in the trace. Returns the graph.
- *locateSlicingCriteria*: locates the slicing criteria provided by the user in the *DynamicControlFlowGraph*. Returns a list of statements in the graph.
- *slice*: takes the *DynamicControlFlowGraph* and the location of the slicing criteria in the graph to slice the program. Returns the slice.

5.1.2 ProgramSlice

The *ProgramSlice* class is a wrapper of Slicer4J's output. It takes the Slicer4J's DynamicSlice and the opened Project as the input and outputs the following objects:

- *sliceLinesUnordered*: a set of source files and line numbers representing the slice for enabling efficient look-up of the lines in the slice.
- *dependencies*: a map from source file locations to their dependencies.
- *firstLine*: the source file location of the first line in the slice.

5.2 User Interface

The purpose of this component is to display UI elements through which the user interacts with Debugger++ and views the slicer information.

5.2.1 Select Slicing Criterion Action

The *SelectSlicingCriterionAction* class implements the functionality of choosing a line in the program as the slicing criterion for the dynamic slicing process. This is added to the action group with the id *EditorPopupMenu* to ensure that the option appears on right-clicking a line in the source-code editor. It selects the line that the mouse is currently on, regardless of any highlighting. On selecting the "Start Slicing from Line" action, dynamic slicing is automatically triggered.

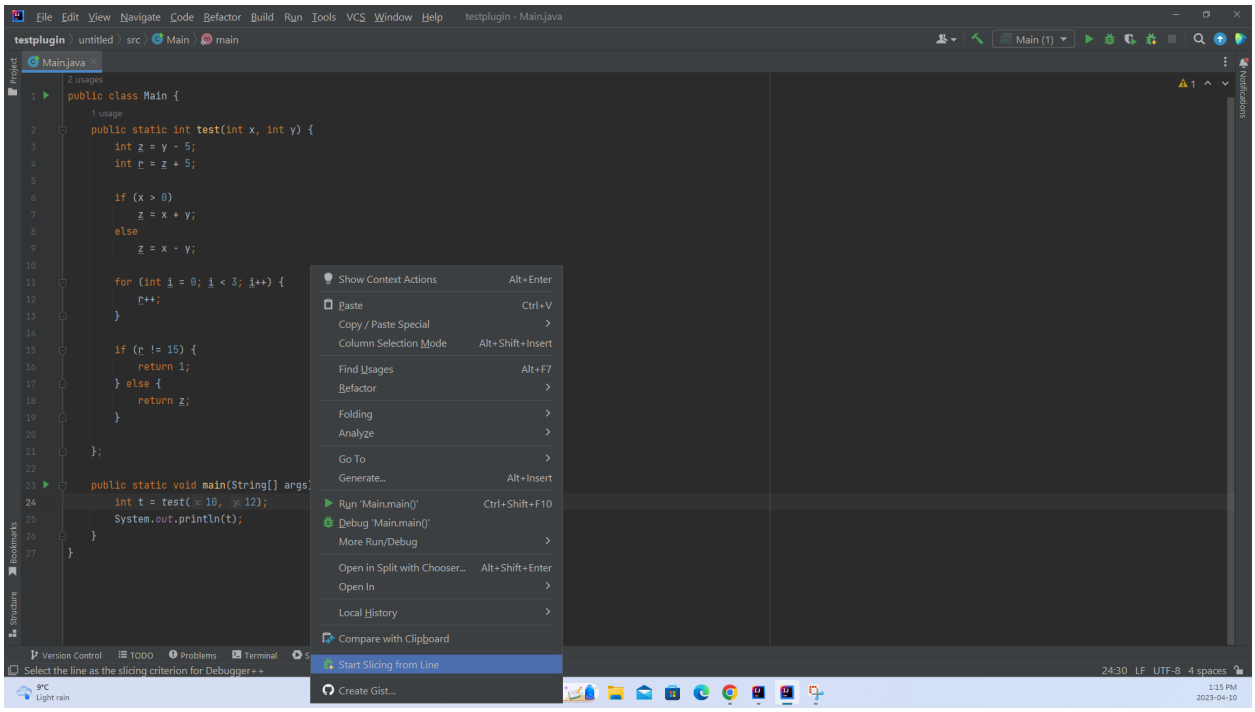

Figure 13: "Start Slicing from Line" option in EditorPopupMenu

5.2.2 Source-Code UI Manipulation

The source code of the program being debugged is manipulated to gray out all lines that are not in the program slice. Line graying is implemented in the *EditorSliceVisualizer* class after obtaining the program slice, triggered when the new Debugger++ *debugProcess* starts, as can be seen within the *DebuggerListener* class. When the debugging session ends, the original line colours are restored. Additionally, the source-code is also manipulated to ensure that no breakpoints can be set on non-slice or grayed out lines, while the debugging activity occurs.

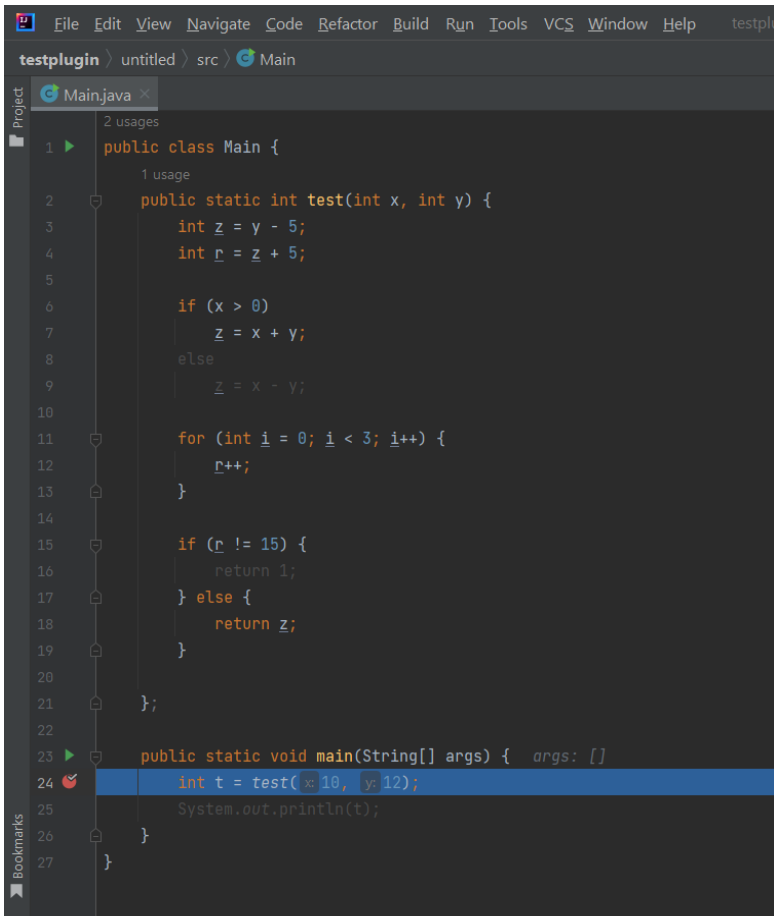

Figure 14: Non-slice lines grayed out in Source Code Editor

5.2.3 Slice Information Tab

The Slice Information Tab includes the sub-tabs of (1) Control Dependencies, (2) Data Dependencies and (3) Dependencies Graph. Thest sub-tabs are updated dynamically based on the current slice line that the debugging session has paused at (line in blue in the source-code editor). Their implementations can be found in the *ui/dependencies* folder within the *ControlDepedendenciesPanel*, *DataDependenciesPanel* and *GraphPanel* classes, respectively. They are added to the Debugger Window by adding a new parent tab, called *sliceInfoTab*, to the current Debugging session's *RunnerLayoutUi*. We use the Java Swing Package for the tab design and the Graphviz library for the Dependencies Graph.

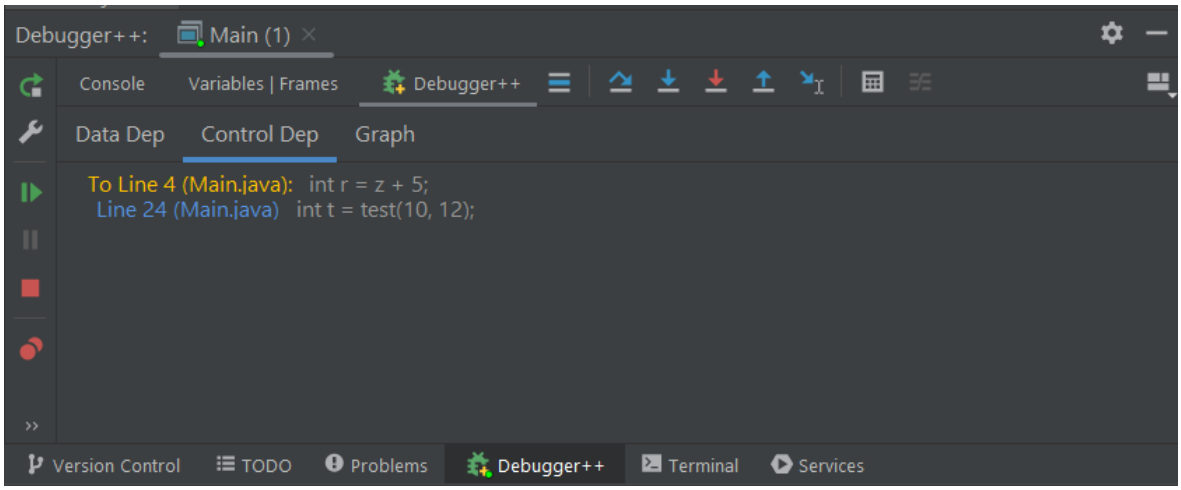

Figure 15: Debugger++ Window Slice Info Tab showing sub-tabs

5.2.4 Icons

All Icons can be accessed from the ui/*Icons* Class and are loaded from the *resources/icons* folder. We have designed Debugger++'s icons based on the IntelliJ Platform's Design Guide. All icons are used in SVG format.

Figure 16: Debugger++ Icon

5.3 Debug Controller

The Debug Controller component is designed to implement Functional Requirement 3.2: "Skip non-slice lines" by modifying the behaviour of existing debug actions. It has the *DppJavaDebugProcess* and *DppJvmSteppingCommandProvider* subcomponents for handling different debug actions, and the *BreakPointController* for managing breakpoints.

5.3.1 DppJavaDebugProcess

The *DppJavaDebugProcess* subcomponent is for handling the “Run to Position” button. It shows an error dialog when the user attempts to run to a position that is not in the slice. *Figure 17* illustrates how it is integrated into the original debugger. The *GenericDebuggerRunner* will be described in Section 5.4.

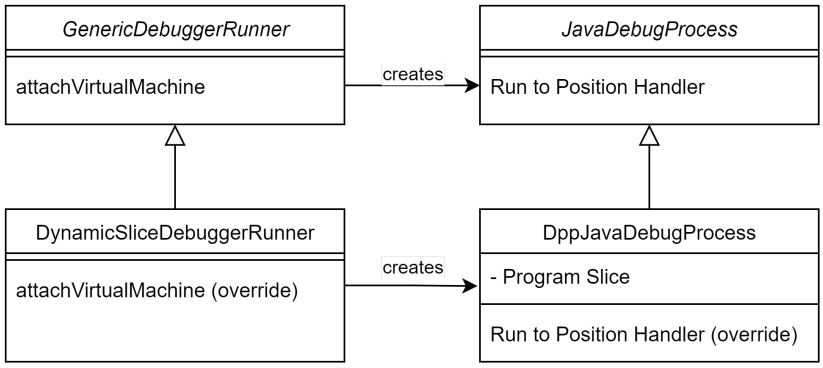

Figure 17: Design of the DppJavaDebugProcess. Only relevant fields and methods are included.

As described in Section 2.2, the *JavaDebugProcess* object defines the handlers for the debug actions in the debugger, and the *JavaDebugProcess* object is created by the attachVirtualMachine method in the *GenericDebuggerRunner*. Therefore, we create a subclass of *JavaDebugProcess* named *DppJavaDebugProcess*, and a subclass of *GenericDebuggerRunner* named *DynamicSliceDebuggerRunner*. *DppJavaDebugProcess* overrides the original debug action handlers, and *DynamicSliceDebuggerRunner* overrides the attachVirtualMachine method to create *DppJavaDebugProcess* instead of *JavaDebugProcess*.

5.3.2 DppJvmSteppingCommandProvider

The *DppJvmSteppingCommandProvider* component provides step into and step over commands that skip non-slice lines. It is integrated into the original debugger through an IntelliJ IDEA extension point. The stepping commands inherit from the original commands but with the difference that they override the *checkCurrentPosition* function. The *checkCurrentPosition* function determines the next step (step again or stop) after the target debugee process has performed a stepping and before the debugger updates the UI and reports back to the user that the stepping command has finished executing. It is originally intended for the case where a single statement spans across multiple lines, and the user wants to step over all of them at

once. In Debugger++, we provide stepping commands that instruct the debuggee to step over again until the current line is in the slice. *Figure 18* illustrates how Debugger++ leverages it to skip non-slice lines.

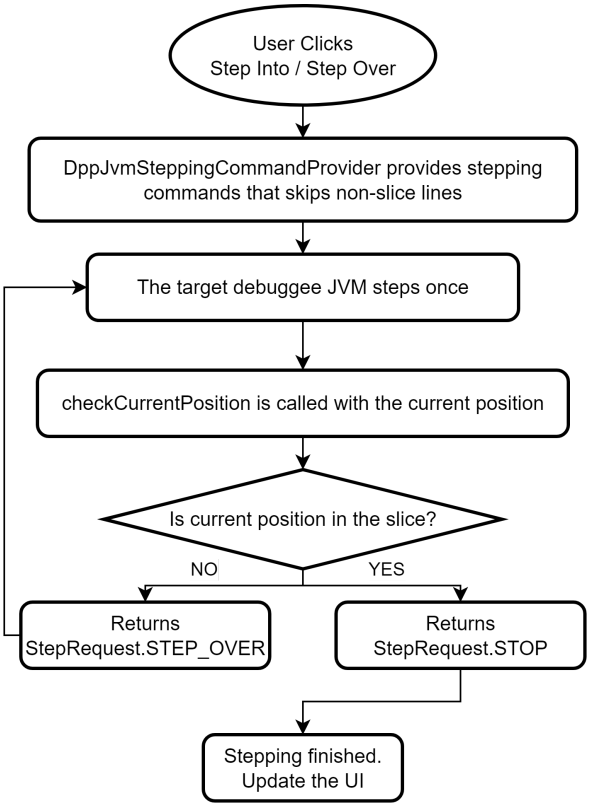

Figure 18: High-level flowchart showing how the custom stepping commands skip non-slice lines

5.3.3 BreakPointController

The *BreakPointController* subcomponent is for managing breakpoints that are specific to Debugger++'s needs. It currently has only one functionality: adding a breakpoint to a given source location.

5.4 Dynamic Slice Debugger Runner

The *DynamicSliceDebuggerRunner* component is for starting a new debugging session. It extends the *GenericDebuggerRunner* in the original debugger.

5.4.1 GenericDebuggerRunner

This subsection describes the key stages in the process of starting a debugging session, and handling a debug command from the user during a debugging session. Note that the IntelliJ IDEA Java Debugger has no documentation so the content in this subsection comes from our analysis of its source code.

The process is summarized in *Figure 19*.

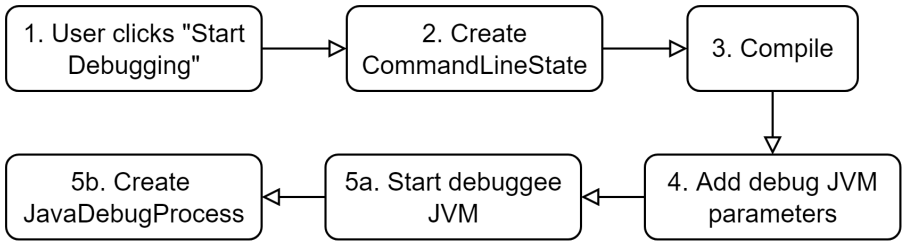

Figure 19: Summary of the process to start a debugging session

1. The user clicks the debug button after selecting a run profile which describes how the program should be run.

   Relevant function:

   com.intellij.debugger.impl.GenericDebuggerRunner#execute
2. Based on the run profile, it creates a *CommandLineState* object which describes how the target debuggee process (i.e. the process that will be debugged) will be started.

   Relevant function:

   com.intellij.execution.runners.ExecutionEnvironment#getState
3. Compiles source code to class files if needed.

   Relevant function:

   com.intellij.execution.impl.ExecutionManagerImpl#compileAndRun
4. Updates the *CommandLineState* object to add JVM parameters such that the JVM will wait for debugger connection at startup.

   Relevant function:

   com.intellij.debugger.impl.RemoteConnectionBuilder#create
5. Proceeds to starting the debuggee JVM and connecting to it.

   Relevant function:

   com.intellij.debugger.impl.GenericDebuggerRunner#attachVirtualMachine

a. Starts the debuggee JVM process.
   Relevant function:
   com.intellij.execution.configurations.CommandLineState#execute
b. Creates a *JavaDebugProcess* object which provides debugging capabilities and defines the handlers of all debug actions.
   Relevant function:
   com.intellij.debugger.engine.JavaDebugProcess#create

5.4.2 DynamicSliceDebuggerRunner

The *DynamicSliceDebuggerRunner* component is a subclass of the original *GenericDebuggerRunner* with two differences:

1. The *DynamicSliceDebuggerRunner* has an extra step for running dynamic slicing. The timing for running dynamic slicing must be after the program is compiled and before the debuggee JVM is started.
2. The *DynamicSliceDebuggerRunner* creates *DppJavaDebugProcess* instead of JavaDebugProcess. The *DppJavaDebugProcess* object defines our own handlers for handling debug commands and is described in Section 5.3.

Figure 19 summarizes the process to start a debugging session with dynamic slicing in Debugger++. The sequence diagram in *Figure 20* also shows the process of starting a debugging session with dynamic slicing.

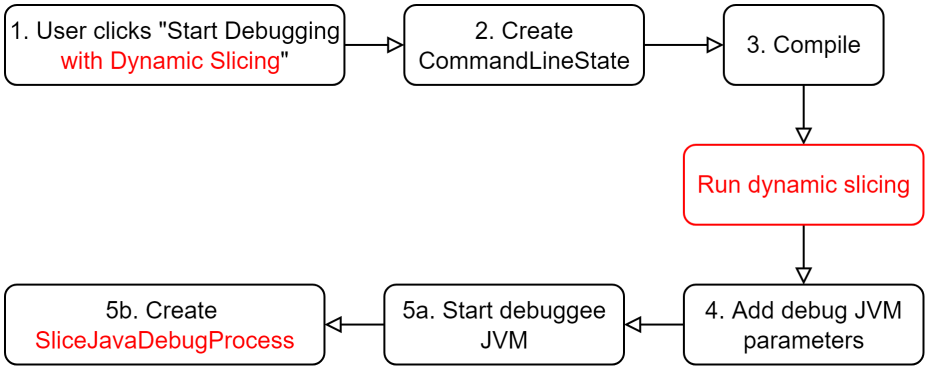

Figure 20: Summary of the process to start a debugging session with dynamic slicing. The difference with the original GenericDebuggerRunner is highlighted in red.

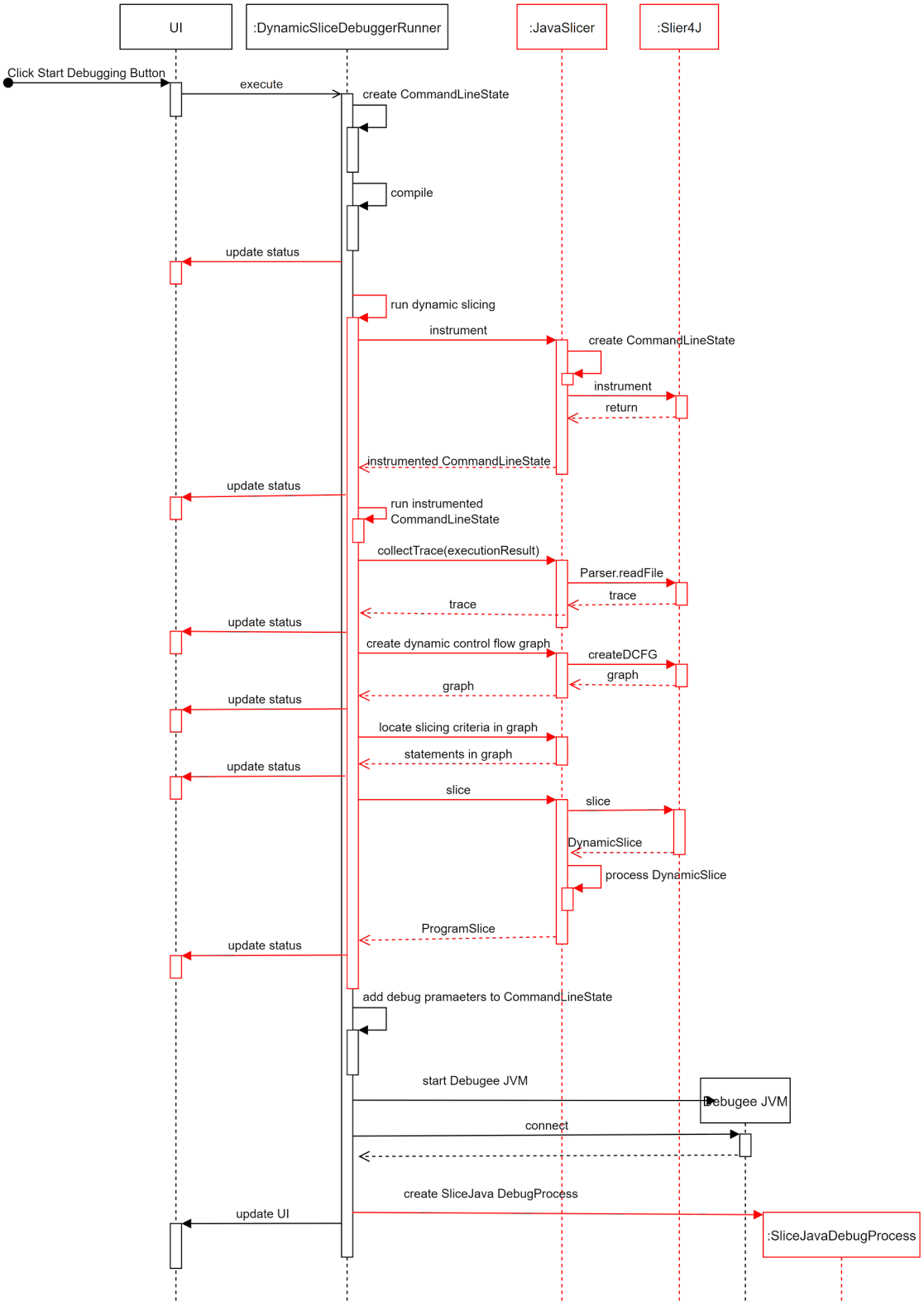

Figure 21: Sequence diagram showing how a debugging session is started. Interactions added by Debugger++ are shown in red.

5.5 Subgraph Builder

The *SubgraphBuilder* component is used for generating a subgraph containing all statements that have been executed and their direct dependencies in a debugging session. The subgraph is dynamically generated as the user steps over each line in the slice. The *SubgraphBuilder* component takes as inputs (1) the dot graph containing control and data dependencies of the whole slice generated by Slicer4J, (2) a list of line numbers in the slice in the execution order, and (3) the current line number of the program that Debugger++ is stopped on, and outputs a subgraph containing nodes which represents lines that are executed in the debugging session as well as their direct dependencies represented as labeled links in the graph.

The subgraph is generated through identifying all line numbers that were executed in the debugging session (which are all lines that appear before the current line number in the slice.log file generated by Slicer4J) and only keeping those nodes with their corresponding links. The *SubgraphBuilder* component is also responsible for reversing the direction of every link in the dependencies graph, since the link direction in the graph produced by Slicer4J is not intuitive. The implementation of the *SubgraphBuilder* component can be found inside the *trace* folder, within the *SubGraphBuilder* class.

6. Future Development Ideas

Based on the result of our Usability Study and discussion with our client, we have identified some ideas for future development that would help enhance Debugger++ and increase its usability. Below is a list of descriptions of our ideas.

1. Change the Dependencies Graph to be clickable to allow a user to jump back to the debugging session state at which the graph node was added
2. Change the Dependencies Graph to have toggle-able dropdowns for each level, and optionally remove the Control and Data dependencies tabs which only showcase direct and not transitive dependencies (1st level of the graph)

3. Map the Jimple statements to the source code and show nodes at the source code level in the graph to make the text in the Dependencies Graph more user friendly
4. Get dependencies (control/data/graph etc.) from the statement instance level, using the current state of execution
5. Do instrumentation at the start of the IDE; do slicing on starting the debugging session to increase performance of the tool

Bibliography

Ahmed, Khaled. “Slicer4J: a dynamic slicer for Java.” *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1570-1574.

“IntelliJ IDEA – the Leading Java and Kotlin IDE.” *JetBrains*, https://www.jetbrains.com/idea/. Accessed 27 November 2022.

“Java Debug Wire Protocol.” *Java Documentation*, Oracle, https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html. Accessed 27 November 2022.

“Java Software.” *Oracle*, https://www.oracle.com/ca-en/java/. Accessed 27 November 2022.

“Java Virtual Machine Tool Interface (JVM TI).” *Java Documentation*, Oracle, https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/. Accessed 27 November 2022.

Appendix A: Design Rationale

The following section describes the process through which our team, in conjunction with the client, arrived at our final project specifications and functional requirements. For information solely on the final solution, please look to Section 3. This section follows a chronological order of tasks and updates to the project. It serves as a record of the design decisions and changes to scope from the beginning of our project.

A.1 Initial Functional Requirements

Based on the needs of the client, our team layed out multiple functional requirements and constraints. For convenience, we will address these as Draft Functional Requirements (DFR), and they will only be referred to in this section. Please see the Requirements Document for the final Functional Requirements and Constraints.

- DFR1: Allow the user to select a slicing criteria and generate a slice based on it.
- DFR2: Highlight slice or gray-out non-slice statements in the source-code.
- DFR3: Allow the user to (1) step backward and (2) step forward through slice statements.
- DFR4: Allows the user to select statements in the slice and present its (1) control dependencies, (2) data dependencies, and (3) variable values read and written.
- DFR5: (1) Presents graph visualizations of the dependencies between statements in the slice, (2) highlighting the node representing the selected statement.
- DFR6: Can be used seamlessly along with the traditional IntelliJ debugger.
- DFR7: Any program, including both non deterministic and deterministic programs should be supported

A.2 Functional Requirements Feasibility Issues

Through our initial investigation of Slicer4J, we identified technical limitations that would impact the feasibility of some Functional Requirements.

A.2.1 Slicer4J does not support line-by-line or 'online' trace collection

Slicer4J requires the entire program to be executed to produce the slice.

Impact on DFR: This impacts DFR6, preventing the user from obtaining the slice mid-debugging session. They must complete the entire execution of the program, and hence, have completed their debugging activity to obtain slice information.

Proposed Solution: Modify Slicer4J implementation to support 'online' trace collection. This would be achieved by redesigning Slicer4J's Instrumentation module, *JavaInstrumenter*, using the Java Virtual Machine (JVM) API to create our own custom Instrumentation.

A.2.2 Slicer4J does not output Variable Values Data for slice-lines

Slicer4J does not collect variable values for slice-lines. Additionally, they can only be collected mid-execution since they are discarded from the run-time stack after execution. Unfortunately, Slicer4J requires an entire execution of the program to produce the slice, and the values are discarded at the slice production time.

Impact on DFR: Cannot achieve DFR4.3: "[present] variable values read and written [by a statement]".

Proposed Solution: If we choose to reimplement the Instrumentation of Slicer4J, as per the Proposed Solution for A.2.1, we can store the variable values during this process.

A.2.3 For non-deterministic programs, each new execution will produce a different slice

With non-deterministic programs, for instance, programs requiring user input or using the "rand()" function, the program may follow a new execution path each time. This may result in a different slice produced with every new execution or debugging session. The issue arises in meeting DFR2, DFR3, DFR4 and DFR5, all of which require all of the slice information prior to the beginning of the debugging session. However, running the slicer *prior* to the debugging session may produce a slice based on a different execution path compared to the one that actually occurs *during* the Debugging Session. Therefore, the user may be presented with completely incorrect slice information compared to the actual execution of the program during

debugging. Unfortunately, the execution path of a non-determinstic program cannot be predicted prior. Re-tracing or re-slicing during the debugging session cannot be achieved due to the limitation *A.2.1 Slicer4J does not support line-by-line or 'online' trace collection*.

Impact on DFR: This impacts almost all DFRs and any slice information used during debugging, as the slice numbers may be different before and after the debugging session. Slice information produced prior to the debugging session may be different from the one observed while executing the program during debugging.

Proposed Solution: Only support deterministic programs. Another solution proposed was the creation of a Slice History Tab feature where a user can view past slice information, during the same debugging activity.

A.3 Implementation Options I

These problems were extensively discussed with the client. Based on the discussion, 3 possible implementation options were identified -

A.3.1 "Mimic" the 1st execution trace in all next executions

The program is executed and its slice is produced prior to the debugging session. The program execution path is saved, recording the chronological trace of program line numbers executed from start to finish. Any subsequent 'executions' during a debugging session from the user's perspective are a clone of the saved trace, by replaying the steps from start to finish to achieve "mimicking". Keep in mind, that a user cannot obtain a slice mid-debugging session due to the limitation *A.2.1 Slicer4J does not support line-by-line or 'online' trace collection*.

Impact on A.2.1: Resolved, as a complete execution will be done prior to the debugging session.

Impact on A.2.2: Unresolved.

Impact on A.2.3 Resolved, as all subsequent executions "mimic" the very first one.

Additional Considerations: As the debugger must be modified to only used the ‘mimicking’ information stored (eg. ‘step next’ will visit the next line in our saved execution), all debugger functions will need to be reimplemented.

A.3.2 Only deal with deterministic programs

Build the solution assuming we will only run it on deterministic programs. This means that Debugger++ will only support programs which will follow the exact same execution path or trace, on any run.

Impact on A.2.1: Unresolved.

Impact on A.2.2: Unresolved.

Impact on A.2.3: Resolved, as we will not support non-deterministic programs.

Additional Considerations: This may not be as useful to developers as they will not be able to use our solution on any non-deterministic programs.

A.3.3 Alter the instrumentation of Slicer4J to support online trace collection

Now Slicer4J will be able to produce a slice mid program execution. The instrumentation module will be reimplemented to support online trace collection.

Impact on A.2.1: Resolved, using our proposed solution.

Impact on A.2.2: Resolved, using our proposed solution.

Impact on A.2.3: Resolved, since we do not need to produce the slice prior to the Debugging Session.

After evaluating our options, our team decided to move forward with *A.3.3: Alter the instrumentation of Slicer4J to support online trace collection*. This is due to the fact that it resolved all of the problems and provides a seamless experience to users for using slicer information along with debugger functions.

A.4 Project Scope Updates I

We extensively designed the solution and outlined user behaviour, following which we presented the solution *A.3.3: Alter the instrumentation of Slicer4J to support online trace collection* to our client. The client then informed us that they would like to make some modifications to the constraints motivated by the ease-of-use for the user and their own internal decisions regarding expansion or modification of Slicer4J. DFR6 was modified, and a new constraint was introduced -

- (modified) DFR6: Can be used seamlessly along with the traditional IntelliJ debugger by ensuring that existing debugger actions only deal with slice lines. Eg. ‘Step Into’ would function as normal, except the user will never step-into a non-slice line.

Now, all debugger actions need to be manipulated to skip non-slice lines.

- (new) DFR8: Avoid modifying or updating Slicer4J.

Now, reimplementing the instrumentation module to allow the actualization of proposed solution *A.3.3: Alter the instrumentation of Slicer4J to support online trace collection* is no longer possible.

A.5 Implementation Options II

Based on these new constraints, and since proposed solution *A.3.3: Alter the instrumentation of Slicer4J to support online trace collection* could no longer be pursued, our team identified two new possible implementation options -

A.5.1 “Mimicking” the existing debugger using the Slicer4J trace

Same as the “mimicking” option previously mentioned, *A.3.1 “Mimic” the 1st execution trace in all next executions*.

Impact on DFRs: DFR6 is impacted by this solution. Program outputs (e.g. printing to console) do not work under the debugging mode. A user cannot view large and nested Java objects. Furthermore, since reimplementation of debugger actions is needed to achieve this solution,

some existing debugger actions such as Evaluate Expression, Watch, Thread Dump, Set Value cannot be supported.

A.5.2 Modifying the existing debugger to skip non-slice lines

The user will step through each line using the original debugger as usual, with the difference that lines not in the slice are automatically skipped by the debugger. This is implemented by modifying the existing debugger's source-code.

Impact on DFRs: DFR7 is impacted as only deterministic can be supported due to the fact that trace collected initially may be different from the current debugging execution.

For Milestone 3, the decision made by the client was to use *Solution A.5.2: Modifying the existing debugger to skip non-slice lines*. Additionally, the client confirmed their approval of the trade-off of only supporting deterministic programs through this solution.

A.6 Project Scope Updates II

On discussion with the client, our solution underwent another review based on their preferences and needs.

- (deleted) DFR3 ("Step Forward" and "Step Backward" buttons): Will be removed and no further implementation will take place.
- (modified) DFR5: (1) Presents graph visualizations of the dependencies between statements in the slice, (2) Building the graph incrementally based on the state of the debugging session, showing only the subgraph for the statements that have been executed.