Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Ein Framework und IDE-Plugin für die vereinheitlichte Verarbeitung von Software Fehlerlokalisierungsergebnissen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Jost Triller
geboren am:	11.7.1998
geboren in:	Berlin
Gutachter/innen:	Prof. Grunske Prof. Kehrer

eingereicht am: .... verteidigt am: ....

Abstract

Localizing bugs in software can be difficult and time-consuming. To make this task easier, software fault localization (SFL) tools are developed and used to support developers in finding the causes of bugs. Today, most SFL tools use their own output format and possibly their own tools for presenting results. This makes the use of multiple SLF tools and integration of new SFL tools in development environments difficult. It also means that for each new SFL tools being developed, it is often required to also create some way to display the results in a readable way.

With this work I will try to lessen some of these issues by implementing a framework that provides a generalized interface for SFL results and an IDE plugin that allows extensions to process and present these SFL results.

Contents

1 Introduction

Fixing bugs in software uses a significant share of the working time of programmers[7, 6]. Reducing the time spent finding and correcting software faults thus will reduce the economical cost of software. Additionally, fixing bugs that appear in production quickly can save significant amounts of money[8]. To reduce the time spent on bug fixing, a number of tools have been developed, such as debuggers that can trace and change the flow of computer programs (e.g., GDB[2]), static code analysis tools (e.g, the Clang Static Analyzer[1]), runtime sanitizer, or software fault localization (SFL) tools. Creating a universal framework for the latter will be the focus of this thesis.

1.1 Software fault localization techniques

SFL tools are intended to help the programmer finding the location of code that causes faulty behavior, which can otherwise be a lengthy process, especially in bigger code bases. Usually these tools use a combination of failing and successful test cases to infer likely causes. The exact way this is done depends on the concrete algorithm used. To give an idea how these work and what kind of in- and output they work with (especially this is relevant for this thesis), I will provide a quick overview over a few of these techniques.

1.1.1 Spectrum-based fault localization

Spectrum-based fault localization methods assign a suspiciousness score to elements of the program (ines or functions, for example). The score for a program element is calculated based on how often this element happens to be executed in failed or successful tests[10]. Typically, the more often this program element is involved in faulty tests compared to its involvement in successful tests, the higher its suspiciousness score is. The expected output of SFL tools based on this technique is a list of program elements (lines, functions, etc.) each with a score indicating the suspiciousness of it.

1.1.2 Program slicing

A program slice is a set of statements that have an effect on some kind of criterion. In the context of debugging, such a criterion could, for example, be the execution of a program statement, that (we know) will cause the program to be in a faulty state (e.g., “`assert x > 0`” with “`x <= 0`”). The idea is that the cause for the faulty behavior (i.e. the criterion) will be in one or more of the statements in the program slice. Knowing these statements may help the programmer to find the bug more quickly[9]. There are different ways on how to create a program slice. One could generate a very inclusive slice by analyzing the program statically to build a program data/control dependency graph. Slices generated like this will most likely contain statements that are irrelevant to the slice criterion in real scenarios. Another way to generate program slices is dynamic slicing, which executes the program with concrete inputs and then generates the slice based on these executions. This may reduce the number of irrelevant statements in a slice. A typical output of a slicing based SFL tool would be the slice associated with a slice criterion for failing tests (e.g., "`assert test() != FAIL`").

1.1.3 Delta debugging

The idea behind delta debugging is to compare a failed test with a minimally different successful test at different states of program execution to find the cause-effect chain that leads to the failure of the failed test[14]. Thus, the typical output of a tool based on this method would be an ordered list of statements with the relevant assignment of variables with the implicit meaning that the next element in the list was (partly) caused by the current element.

1.1.4 Learning based fault localization

There exist different machine learning based approaches. One example is to train the model with coverage data of test cases of the program that we want to test, to classify whether a test case will fail or not. Using artificial test cases that only cover very specific parts of the program, the trained model then rates the probability of this specific part of the program causing the test to fail[13]. The output could be a list of program statements linked with a likelihood for being the cause of a bug. While I don't know of any tools that do this, I can also imagine that an SFL tool using this ML method could allow a programmer to interactively invoke the classification model on a selected part of the code.

1.2 Motivation for a unified framework

Usually each SFL tool has its own specifications how the inputs are provided and in which format the results are returned. This comes with the problem that tools for presenting the results to the programmer in a helpful way (instead of potentially long lists of numbers and raw text), or for integrating the SFL tool in a development environment, need to be developed from scratch for each new SFL tool. This makes the development of SFL tools unnecessarily expensive. Additionally, this will make it difficult for the user to integrate multiple SFL tools into a single development environment. When introducing new SFL tools, potentially existing frameworks for SLF tools can't be reused, resulting in additional configuration work.

To deal with some of these issues, Simon Heiden proposes to create a language that could be used by a large range of SLF tools to describe the results of the automatic fault localization processes. Having such a universal format, SFL tool developers can focus on targeting that, while tools using the SFL results (e.g., visualizing the result, letting the programmer explore the results interactively in the IDE, or describing them in natural language) can be developed and used independently of the SFL tool. Integrating different SFL tools that implement this proposed language will be also much easier, as processing the results would be nearly identical no matter the SFL tool that produced the result.

1.3 Fault describing transition systems

The fundamental idea of how to encode numerous possible bug descriptions is to use a transition system to describe classes of program flows. Such a transition system has predicates on each node and edge. These predicates describe the conditions of the program at different times, for example, "`Program is at line 5 and x == 5`". Nodes stand for singular states of a program, while edges describe any number of states that happen between two states/nodes. Thus edges use temporal logic formulas as predicates, while predicates associated with nodes use traditional non-temporal logic. Using these predicates on transitions systems, bugs then are described by creating transition systems that are as close as possible to describing the class of program flows that include a specific bug. An example of a fault describing transition system (FDTS) can be seen in Figure 1.

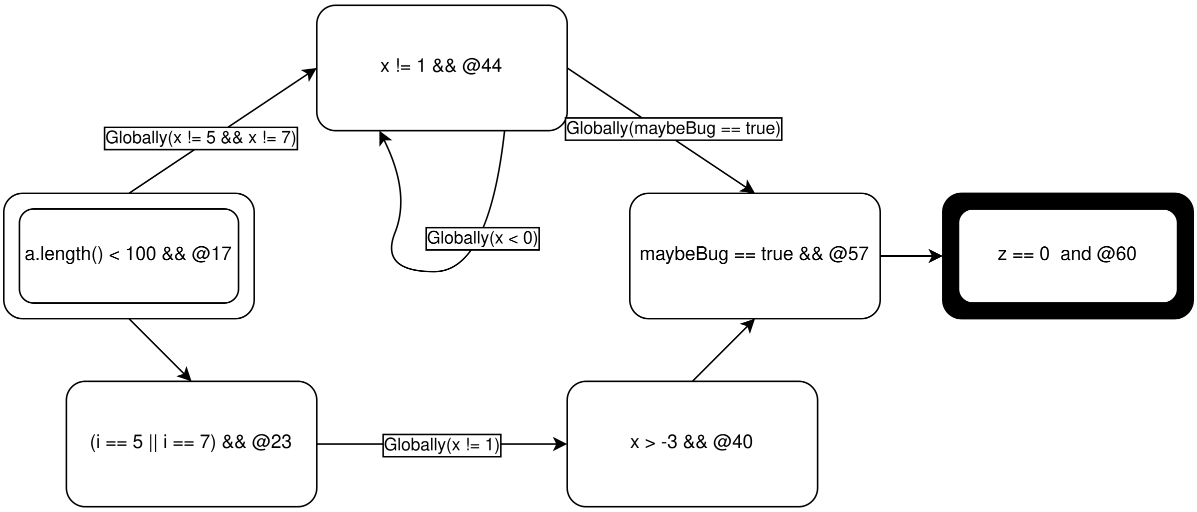


Figure 1: An exemplary FDTS. Each box stands for a node in the graph and thus for some class of program state. The very left node, for example, requires that "`a.length < 100`", and additionally "`@17`" (which stands for the program being in a state that corresponds to the source code line 17). The top left edge also has a requirement, namely that for all program states that happen during the start and end node of that edge (that is what "`Globally(...)`" stands for) `x` must be neither 5 nor 7. Because an FDTS describes a class of transition systems, it needs at least one entry and exit point. Here, these so-called "start nodes"/"end nodes" are depicted by double box border/thick box border. This FDTS describes program flows of the code example Listing 2 that end in the function "`main2()`" returning 1.

1.4 Scope

Implementing the proposed SFL result language in form of fault describing transition systems and using the resulting framework to create an IDE plugin for the JetBrains

IntelliJ family, that can be used to load SFL results and process them using additional tools/extensions, will be the work of this thesis.

2 Requirement analysis

The fundamental motivation and thus requirement for the system I will develop, is to mitigate the problems described in subsection 1.2.

Rough system overview The system should consist of a framework that provides a universal interface for loading and working with SFL results (based on FDTS), and an IDE plugin, based on the previously mentioned framework, that allows the installation and usage of extensions over a graphical interface.

User types The users of this system can be grouped into three different types:

- **End user**: Someone who wants to use the plugin together with extensions and SFL tools to help them to locate bug causes.
- **Plugin extension developer**: Someone who develops extensions for the IDE plugin, that process and possibly present the SFL results to the end user.
- **SFL tool developer**: Someone who develops an SFL tool and wants to target the FDTS framework.

Each of these types of users will have different requirements for the system, so I will consider them separately.

2.1 End user requirements

1. The end user wants to load the results of SFL tools into the plugin. This should be a very similar procedure for each kind of SFL tool.
2. The end user wants to organize loaded SFL results. This could include sorting SFL results (for example by suspiciousness score), or grouping SFL results into different tabs. The general criterion is that the lack of organization features shouldn't hinder the usage of the plugin.
3. The end user wants to easily install new 3rd party extensions. Again, this process shouldn't differ significantly across extensions.
4. The end user wants to run selected extensions on selected loaded SFL results.

The end user also has demands for potential extensions, which aren't directly requirements for the system, but they indirectly add constraints to what kind of API the system should at least provide the plugin extensions with.

5. The end user wants to view the SFL results in the internally used, raw format (FDTS).

6. The end user wants to use one or more SFL results to generate one or more new SFL results. A concrete example where this could be useful, is an extension that transforms a very complex SFL result into a smaller, easier to understand SFL result (for example, by only reasoning about program flow at function level, instead of line level).

7. The end user wants to read an SFL result described in natural language

8. The end user wants to retrace the SFL result, possibly interactively, alongside the source code in the IDE.

9. The end user wants to select elements in an extension that displays an SFL result in a raw format, and have the selected element highlighted in the extension that describes the SFL result in natural language. This rather concrete example should be seen as reason for why communication between different extension could be necessary (see also subsubsection 2.2.2).

2.2 Plugin extension developer requirements

In this case, I will differentiate between requirements for the FDTS framework (i.e. the interface for loading and working with SFL results), and the requirements for the interface that the IDE plugin should provide for extensions. Generally the FDTS framework should also be usable for usage in different environments (for example, other IDEs, such as Eclipse), to ensure that the libraries (which, for example, allow the description of FDTSs in natural language) that are used by plugin extensions are not bound to this specific IDE plugin.

2.2.1 Requirements for the framework

1. The extension developer wants to have access to the entire content of an SFL result. In the context of an FDTS this means access to the general graph properties (nodes, edges, neighbors, etc.), access to predicates at nodes and edges, being able to access and understand the meaning of predicates.

2. The extension developer wants to modify and create new SFL results.

3. The extension developer wants to use a custom implementation of the FDTS interface (for example for application specific performance reasons).

2.2.2 Requirements for the IDE plugin

1. The extension developer wants access to the IDE functions, for example, to be able to see the source code, or to have access to the debugging functions of the IDE.

2. The extension developer wants an interface to allow end user interaction with the extension. See items 4–9 in subsection 2.1.
3. As an implicit requirement of item 9 of subsection 2.1 and for similar applications, the extension developer wants an interface to communicate with other plugin extensions.

2.3 SFL tool developer requirements

The SFL tool developer wants to be able to completely and without too many expenses represent the results of their SFL tool using the FDTS language. While it is not yet clear if FDTS are powerful enough to fulfill this criterion for most existing and potential SFL tools, for this work it is assumed that the FDTS language is good enough, and the analysis in this direction will be left for future work.

The requirement for the framework is mainly to make sure that the full potential of the FDTS scheme is used. This means that the implementation of predicates at nodes and edges should be powerful enough that either all with FDTS compatible SFL tools can represent their results out-of-the-box or that it is possible to add new predicate types using framework extensions.

3 Architecture

Designing a capable architecture will be key to ensure a high extensibility of the system. The extensibility is important to allow many potential approaches for presenting SFL results to be implemented for the IDE plugin (which is directly and indirectly part of the end user and extension developer requirements).

As the plugin will be developed for the IntelliJ IDEA IDE[4], which has a Java plugin API, Java has been selected to be the language in which the system will be implemented. The IntelliJ IDE has been chosen because it is a widely used IDE family with similar plugin interfaces for IDEs for different languages (Java, Python, C++, etc.), which in the future would make it easier to make this plugin ready for other programming languages.

3.1 FDTS framework

The FDTS framework consists of four main parts: The FDTS interface, the FDTS loader, a default implementation for the FDTS interface, and a small predicate library.

3.1.1 Predicate Library

As described in subsection 1.3, an FDTS has predicates at its nodes and edges. These are supposed to describe parts of the program state, for example, by having constraints for variable values, or requiring that the program is at a state, that corresponds to a specific line in the source code. Statements like these (“`x == 1`”, “`a.length > 0`”, or “`at line 350`”) are considered to be atomics, as they don’t contain any sub-predicates. Predicates with sub-predicates are functions like `And`, `Not`, `Or` (logical operators) `Globally`, or `Future` (temporal logic operators). This way, multiple atomics can be logically combined.

The concrete implementation uses types to describe which kind a given predicate is (the Java function `instanceof` can be used to determine the type at runtime). While this will potentially require a long list of "`if(v instanceof SomePredicate)`" statements, it would be necessary in any case, as each predicate will need to be handled differently, simply because of their different semantics. Consequentially, SFL results that require additional predicates (that come not installed with the predicate library) can only be used either by programs/extensions that don't require the knowledge of the semantics of all predicates, or by programs/extensions that have knowledge of these additional predicates.

See Figure 2 for a specific explanation of the predicate class architecture.

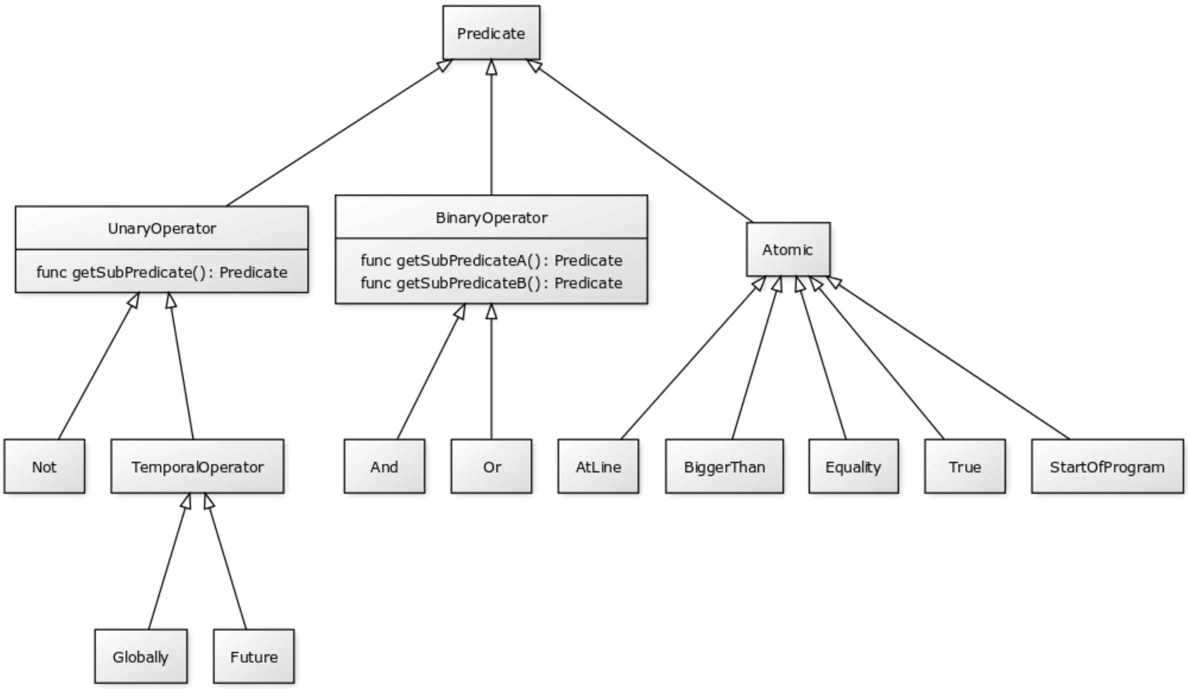


Figure 2: Every predicate implements the interface `Predicate`. Atomic types should be inheriting from `Atomic`. Temporal operators and `Not` are all unary logic operators and thus implement `UnaryOperator`, which provides a function to retrieve the single sub-predicate. Similarly, `And`, `Or`, and the temporal operators implement `BinaryOperator` which provides functionality to access the sub-predicates. The current framework comes with five default atomic predicate types: `AtLine`, `BiggerThan`, `Equality`, `True`, `StartOfProgram`. New atomic types can be easily added to the framework.

Java's generic type feature is used to provide a compile-time check, that predicates in a temporal context are temporal predicates, and predicates in a singular state (i.e. non-temporal) context are non-temporal predicates. Atomics are assumed to always be non-temporal, so they always only describe a single program state. Temporal operators (`Future`, `Globally`) are always considered to be temporal predicates. Their sub-predicates can be both temporal predicates, or non-temporal predicates. The logical operators `Not`, `And`, and `Or` can also be temporal and non-temporal, however with the condition, that their sub-predicates are also temporal exactly if they themselves are temporal, or the other way around, that they themselves are non-temporal exactly if their sub-predicates are also non-temporal.

Concretely, this can be achieved having a generic parameter for the interface `Predicate`, and then applying the mentioned restrictions in the implementations, e.g., `Atomic` only inherits from `Predicate<NonTemporal>` while `And` can inherit from both `Predicate<NonTemporal>` and from `Predicate<Temporal>`.

This guarantee only holds during compile time, and because generic type information gets erased during runtime, there also exists functionality to check during runtime, whether a predicate is temporal on non-temporal. The alternative would have been to have concrete types for each scenario, (e.g., having `PredicateTemporal` and also `PredicateNonTemporal`). However, this would introduce a lot of code duplication which lead me to decide against this approach.

3.1.2 FDTS interface

As the FDTS is fundamentally a directed graph, the FDTS interface needs the usual graph features: Getting a list of all nodes/edges, getting neighbors of nodes, getting out-/incoming edges of nodes, creating/removing nodes/edges. Nodes are indexed by a wrapper class (`NodeID`) around an integer. An implementation of the FDTS interface can use this integer in any preferred way. A single integer should generally be enough to implement any kind of technical indexing scheme. Edges are indexed using a wrapper class (`EdgeID`) around two `NodeID`s which stand for the start/end node of the given edge. This means, that a function to retrieve the start and end nodes of an edge is not needed in the FDTS interface, as one can just look directly at the `EdgeID` in question to get that information.

Furthermore, the FDTS interface has a number of functions to set and get predicates connected to nodes and edges. There are also functions to set, unset, get, and check for start/end nodes.

The FDTS interface provides a method to transform one FDTS instance into another semantically identical FDTS instance of another FDTS implementation. This will be useful in case that an extension needs to use an FDTS implementation different from the default implementation (which will be described in subsubsection 3.1.3), for example, to improve performance for a specific edge case.

A complete depiction of the FDTS interface can be seen in Figure 3.

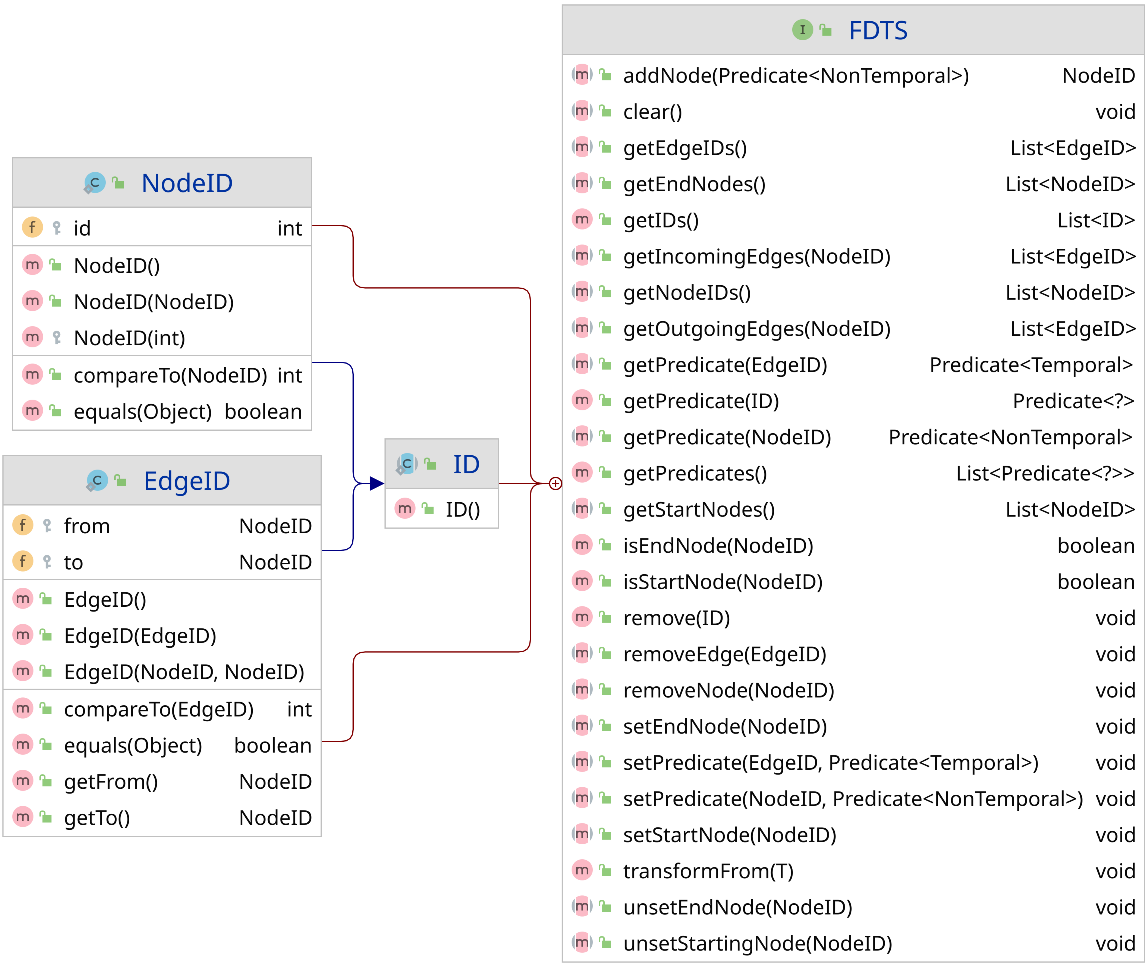

NodeID
id int
NodeID()
NodeID(NodeID)
NodeID(int)
compareTo(NodeID) int
equals(Object) boolean
EdgeID
from NodeID
to NodeID
EdgeID()
EdgeID(EdgeID)
EdgeID(NodeID, NodeID)
compareTo(EdgeID) int
equals(Object) boolean
getFrom() NodeID
getTo() NodeID
ID
ID()
FDTS
addNode(Predicate<NonTemporal>) NodeID
clear() void
getEdgeIDs() List<EdgeID>
getEndNodes() List<NodeID>
getIDs() List<ID>
getIncomingEdges(NodeID) List<EdgeID>
getNodeIDs() List<NodeID>
getOutgoingEdges(NodeID) List<EdgeID>
getPredicate(EdgeID) Predicate<Temporal>
getPredicate(ID) Predicate<?>
getPredicate(NodeID) Predicate<NonTemporal>
getPredicates() List<Predicate<?>>
getStartNodes() List<NodeID>
isEndNode(NodeID) boolean
isStartNode(NodeID) boolean
remove(ID) void
removeEdge(EdgeID) void
removeNode(NodeID) void
setEndNode(NodeID) void
setPredicate(EdgeID, Predicate<Temporal>) void
setPredicate(NodeID, Predicate<NonTemporal>) void
setStartNode(NodeID) void
transformFrom(T) void
unsetEndNode(NodeID) void
unsetStartingNode(NodeID) void

Figure 3: An overview over all methods of the FDTS interface.

3.1.3 FDTS default implementation

The provided default implementation of the FDTS interface stores nodes in an `ArrayList<Predicate<NonTemporal>` object (see Figure 4). Elements of value `null` are considered to be non-existent. The indexing integer of `NodeID` is then simply used to point at the position in this list. Edge predicates are stored in an `ArrayList<ArrayList<Predicate<Temporal>>>` object, where the edge from node `a` to node `b` is located at `edges[a][b]` (`a` and `b` stand for the internal integer value of the `NodeID`s that describe the edge). Start and end nodes are stored in two lists of `NodeID`s.

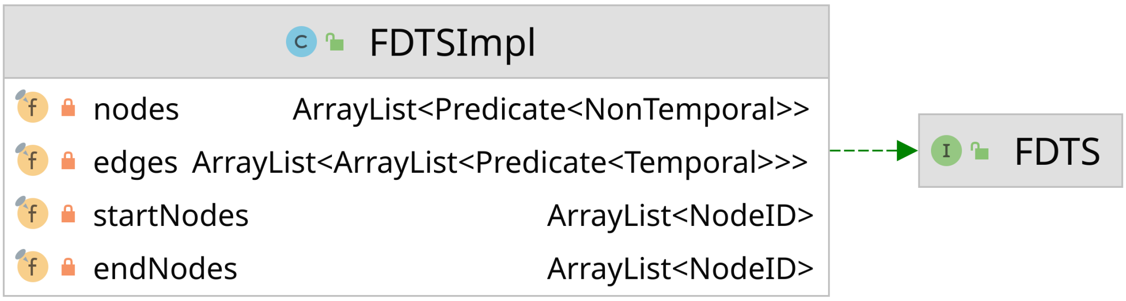

FDTSImpl
nodes ArrayList<Predicate<NonTemporal>>
edges ArrayList<ArrayList<Predicate<Temporal>>>
startNodes ArrayList<NodeID>
endNodes ArrayList<NodeID>
FDTS

Figure 4: The important fields of the default FDTS implementation.

The implementation hasn't been designed to be fast or memory efficient. The main goal was to create a simple, easy to understand data structure. This is important, as this default implementation (converted to the JSON format) is intended to be the primary communication format with SFL tools.

The framework comes additionally with an FDTS decorator class and an `FDTSMeta` implementation based on this FDTS decorator and the default implementation, with a dictionary which can be used to capture all additional information an SFL tool might want to provide. This is mainly meant to be used to communicate with SFL tools. Extensions that want to use additional meta information of an FDTS should probably convert a given `FDTSMeta` type into their own data structure, as it is quite difficult to work with a map/dictionary of strings, instead of proper statically typed structures.

3.1.4 FDTS loader

The `FDTSLoader` is a wrapper around the Gson[3] library. Gson allows (de)serialization of JSON and Java Objects. The `FDTSLoader` is necessary to manage the otherwise ambiguous subtypes of `FDTS` and `Predicate`. It allows converting a JSON formatted string into any FDTS implementation, or into a list of `FDTSMeta` instances. Restricting the loading of lists of FDTSs to only `FDTSMeta` is not intended but currently the only workaround for some technical issues.

At runtime the `FDTSLoader` can be configured to accept 3rd party FDTS and `Predicate` implementations.

FDTS JSON format Listing 1 is an example, how a JSON formatted FDTS could look like. As described previously, the FDTSLoader could load any kind of FDTS implementation, but in practice it will be likely easiest to use the provided `FDTSMeta` implementation, as the later described IDE plugin will be able to only load this type for the time being.

Listing 1: An example of an FDTS in a JSON format based on the `FDTSMeta` implementation.

```
{
  // the "info" object can hold all kinds of meta data about the FDTS
  "info": {
    "sus": 0.8164965809277261,
    "tool": "Some␣SFL␣tool",
    "test_module": "src/test/java/Testing.java"
  },
  // this is the FDTS itself, which will be translated by the
  // FDTSLoader into the FDTS implementation that is specified under
  // "type" (in this case "FDTSImpl")
  "fdts": {
    "type": "FDTSImpl",
    // list of nodes, contains the predicates at each node
    "nodes": [
      {
        "type": "Not",
        "subPredicate": {
          "type": "AtLine",
          "line": 59,
          "path": "./src/main/java/TestcaseForTracer.java"
        }
      },
      ...
    ],
    // list of edges, an edge at position [x][y] in this array
    // stands for an edge from node x to node y, contains the
    // predicates at each edge
    "edges": [
      [
        {
          "type": "Globally",
          "subPredicate": {
            "type": "True"
          }
        },
        null,
        ...
      ],
      ...
    ],
    // start and end nodes are simply an array of nodeID objects
    "startNodes": [{"id": 0}],
    "endNodes": [...]
  }
}
```

3.2 IDE plugin

The purpose of the IDE plugin is to give extensions the possibility to directly interact with the for the SFL results relevant development environment (e.g., the source code and build system), and secondly to provide the end user with a graphical interface to load SFL results and extensions, and use them together.

The user interface is implemented using the Java Swing library, and the entire management of FDTS objects and extensions is done very close to the data structures the Swing API provides.

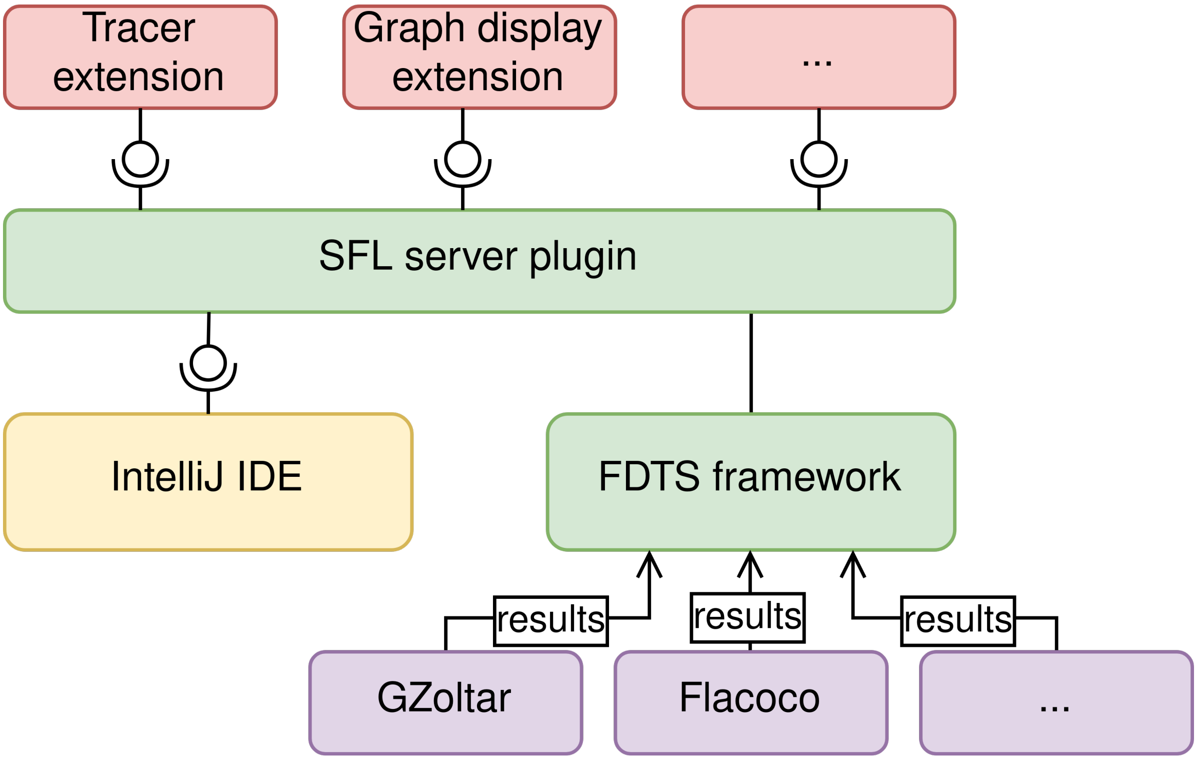


Figure 5: A rough overview of the IDE plugin. It uses the interfaces for plugins of the IntelliJ IDE. The plugin provides an API for extensions to work with SFL results, as well as functionality for reading and loading SFL results using the FDTS framework. It can be seen as a server that provides clients (extensions) with FDTSs as data.

3.2.1 API for extensions

The central part of the API for extensions are two Java interfaces: `Consumer` and `Transformer`. Extensions that want to use one or more FDTSs to present them to the user, or use them in some other way (possibly using the IDEs functionality), can implement the `Consumer` interface (see Figure 6). The single function of that interface (`void run(...)`) will be called when this `Consumer` is selected by the user to be run on a selected FDTS. Through the arguments of that function, the `Consumer` will receive the list of input FDTSs and access to the IDE (e.g., window creation, source code access). `Transformer` works the same, except that its function returns a list of (new) FDTSs. Extensions can use the `Transformer` interface for applications, that convert one or more FDTSs into one or more new FDTSs. An example would be, as previously mentioned, the intention to transform a very complex FDTS into a simpler FDTS that only reasons about program flow at the function level. In section 4 I will describe in more detail how the end user can create a new custom `Consumer` type by combining a number of transformers and consumers.

For `Consumer` types there is additionally the choice of implementation `Connectable`, which allows communication between jointly executed `Consumers`, similar to a simple observer pattern. At the moment, the only option for messages is to send/receive the currently "activated" part of the FDTS (see the requirement item 9 of subsection 2.1). In future this could be extended to be a more universal messaging system. For example instead of sending (and receiving) and FDTS and an element ID, extensions could simply communicate using `Object` and then use `instanceof` to decide how to act on a message.

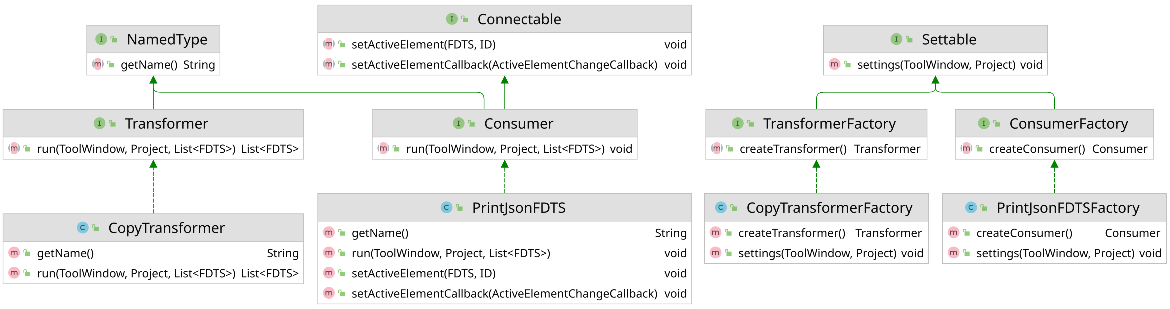

NamedType
getName() String
Connectable
setActiveElement(FDTS, ID) void
setActiveElementCallback(ActiveElementChangeCallback) void
Settable
settings(ToolWindow, Project) void
Transformer
run(ToolWindow, Project, List<FDTS>) List<FDTS>
Consumer
run(ToolWindow, Project, List<FDTS>) void
TransformerFactory
createTransformer() Transformer
ConsumerFactory
createConsumer() Consumer
CopyTransformer
getName() String
run(ToolWindow, Project, List<FDTS>) List<FDTS>
PrintJsonFDTS
getName() String
run(ToolWindow, Project, List<FDTS>) void
setActiveElement(FDTS, ID) void
setActiveElementCallback(ActiveElementChangeCallback) void
CopyTransformerFactory
createTransformer() Transformer
settings(ToolWindow, Project) void
PrintJsonFDTSFactory
createConsumer() Consumer
settings(ToolWindow, Project) void

Figure 6: The structure of the extension API. `CopyTransformer(Factory)` and `PrintJsonFDTS(Factory)` are exemplary extensions.

Extensions are loaded using a class loader, that looks for `TransformerFactory` and `ConsumerFactory` implementations among the libraries in the IDE plugin package. In future this might be extended to load classes in JARs in a specific folder. The reason for having factories for `Consumer` and `Transformer` is to make sure that the end user selecting a consumer to be run on an FDTS, shouldn't need to worry about the state of the consumer that may be influenced by previous usages of this consumer. Each time the end-user selects a consumer, in the background the plugin creates a new `Consumer` using the according factory. To allow the end user to change properties of a consumer (e.g., setting up the preferred way some graph is displayed), without being required to do this each time the consumer is newly executed, both factory types allow the implementation of a `settings` function, which provides access to the IDE, such that the extension can create a settings dialog. The plugin provides the end user with a graphical interface from which they can select to run this settings function of a selected `ConsumerFactory` or `TransformerFactory`.

3.2.2 Extensions

The plugin comes with a few extension out-of-the-box. These are mostly written with the intention to test and demonstrate the (required) capabilities of the system, and thus aren't focused on providing a good debugging experience to the user.

Tracer extension This extension takes an FDTS and, as long as this extension is active, uses it during debugging sessions (i.e. when the end user executes the program with the debugger) to stop at moments where the current program state matches a node of the FDTS under the constraints of the previously traversed program states. This can be useful for the programmer to get a better understanding of the behavior of a program that is (possibly) the reason for faulty test cases.

Each time a new debugging session is started, the extension sets breakpoints for each node, that contains an `AtLine` atomic (to be evaluated to `true`, this atomic requires the program being in a state corresponding to the given line in a source code file). Using an IDE breakpoint feature, additional constraints that the predicate at the given node expresses, are added to the breakpoints. These constraints in form of a Java expression evaluating to a boolean, must be true so that the associated breakpoints get activated.

Now, every time a breakpoint gets reached, the extension considers it and all previously encountered breakpoints, and tries to find a path from any start node of the FDTS to the node that is associated with the current breakpoint, with the restriction, that only FDTS nodes can be visited, that have already been visited previously during the program execution.

This means that at breakpoints where it can be proven, that the current state of the program cannot be part of any FDTS node (which to recapitulate describe a class of program states), are skipped. However, this leaves a number of program states that won't be skipped, but will contradict the FDTS at a later state. Additionally, predicates on edges are completely ignored. Improving on this is left for future work.

Print active FDTS element This extension displays the JSON representation of the selected FDTS graph in a new tab window. If it is run together with another extension that uses the messaging system between extensions, it will print the JSON representation of the currently selected node/edge.

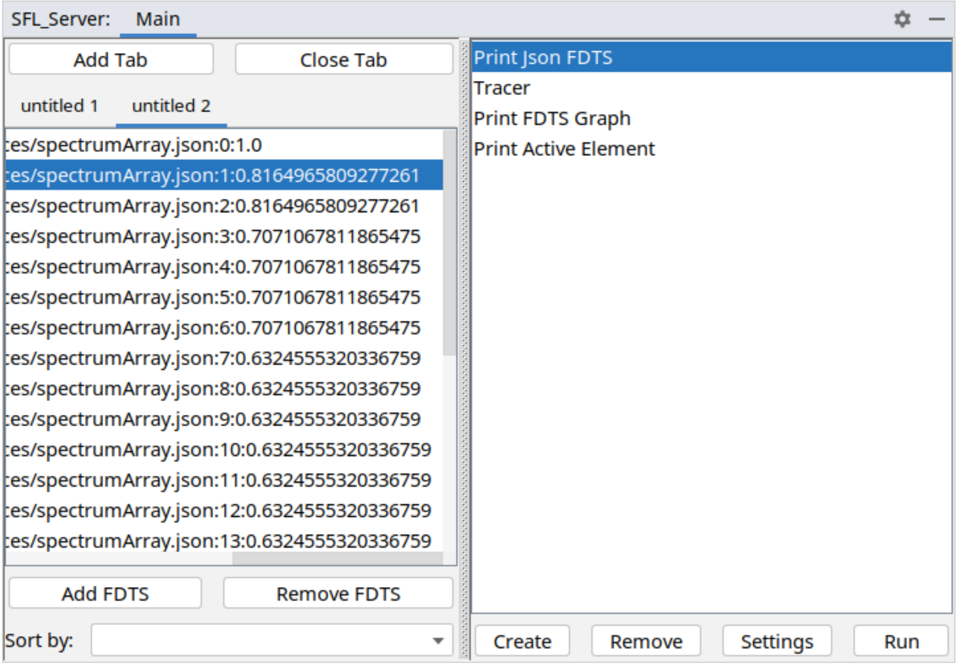
Display FDTS as graph This extension uses JGraphX, a simple graph visualization library[5] to display the FDTS graph. It allows the end user to select nodes and edges. When this happens, this extension uses the messaging system to broadcast the selected element to each extension, that has been executed together with this one.

Copy transformer The copy transformer is a `Transformer` that can be set up by the user to return any number of copies of the incoming FDTS. This has no practical application so far. It is merely supposed to demonstrate how transformers and extension settings work.

4 User interface

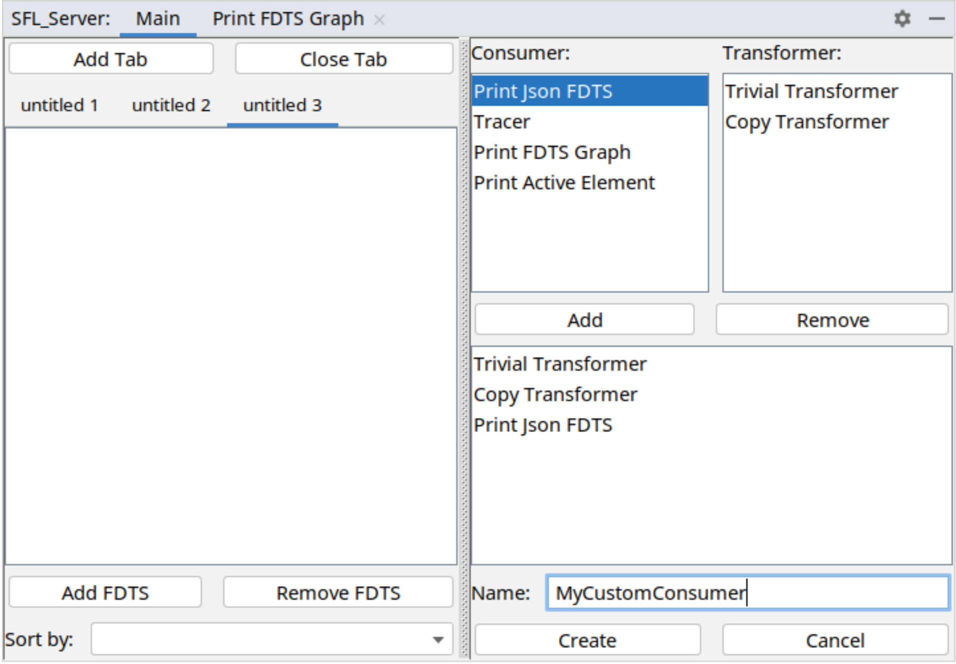
The user interface of the IDE plugin consists of two main panels: The FDTS manager window, and the consumer manager window. The FDTS manager window allows the programmer to load FDTS from JSON formatted text files and organize the loaded FDTS in different tabs. Not yet implemented is a sort function, that allows ordering FDTSs in a tab after some meta criterion, for example the suspiciousness score or filename. The consumer manager window allows the selection of extensions (in the form of consumer). The user can also create custom consumers by chaining multiple transformer together with a single consumer at the end.

Figure 7, Figure 8, and Figure 9 demonstrate the main user interface features.



```
:es/spectrumArray.json:0:1.0
:es/spectrumArray.json:1:0.8164965809277261
:es/spectrumArray.json:2:0.8164965809277261
:es/spectrumArray.json:3:0.7071067811865475
:es/spectrumArray.json:4:0.7071067811865475
:es/spectrumArray.json:5:0.7071067811865475
:es/spectrumArray.json:6:0.7071067811865475
:es/spectrumArray.json:7:0.6324555320336759
:es/spectrumArray.json:8:0.6324555320336759
:es/spectrumArray.json:9:0.6324555320336759
:es/spectrumArray.json:10:0.6324555320336759
:es/spectrumArray.json:11:0.6324555320336759
:es/spectrumArray.json:12:0.6324555320336759
:es/spectrumArray.json:13:0.6324555320336759
```

Figure 7: The FDTS manager window (left panel) can be seen with a number of FDTSs already loaded. On the right is the consumer manager window. When selecting one or more consumers in the consumer manager window and at the same time one or more FDTS in the FDTS manager window, the "Run" button in the consumer manager window can be clicked to execute the extensions. Starting multiple extension at the same time this way will let these extensions communicate using the message system.



Name: MyCustomConsumer

Figure 8: The panel on the right appears after clicking on the “Create” button in the consumer manager window. In this window the user can combine transformers and consumers to a new consumer type. This way, transformer extensions can be used.

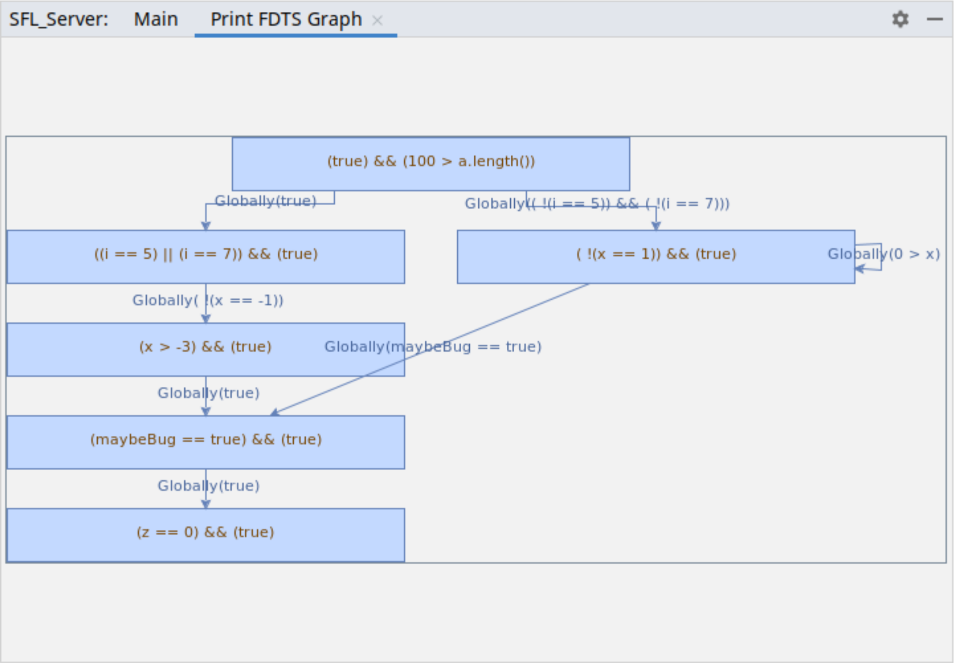


Figure 9: An example of a running consumer. It displays the previously selected FDTS using a graph visualization library.

5 Evaluation

To evaluate to what extent the FDTS framework and the IDE plugin satisfy the goal of removing barriers in SFL tool development and making the usage of SFL tools more accessible, ideally a user survey (among SLF developers and users) would need to be done. However, this validation isn't feasible for the scope of this thesis. Instead, I will systematically examine if the requirements set in section 2 are satisfied.

Additionally, as a proof-of-concept, the SFL tool FLACOCO[12] has been customized to target the plugin.

5.1 Verification: End user requirements

(see subsection 2.1)

1. ✓The end user can load any conforming SFL result.
2. (✓) The end user can organize SFL results using tabs. Sorting is not implemented, but there are no barriers doing so in the future. Whether the plugin is easy to use in this respect would need to be validated using user surveys.
3. (✓) Installing 3rd party extensions is possible. It is not yet user-friendly, as it is required to copy new extension packages directly into the IDE plugin package. However, it is possible to make this process significantly easier without any redesign, by using a different class loader to load classes from a specified directory.
4. ✓The end user can select any extension to be executed on any loaded SFL result.
5. ✓The end user can view the internal representation of an SFL result, by using the extension "Print active FDTS element" described in paragraph 3.2.2.
6. (✓) Applying suitable transformer extension, the end user can send newly created SFL results to a selected consumer. However, more complex transformers have not been implemented yet.
7. (✓) It is not clear if it is possible to describe an FDTS in natural language. However, in case that it is in principle achievable, it should be possible to implement the approach using an extension for the IDE plugin, as all information inside an FDTS is provided to extensions through the FDTS interface.
8. ✓Interactive retracing of an SFL result alongside the source code in the IDE is possible, as the "Tracer extension" concept shows.
9. (✓) Primitive communication between selected extension is possible using the `Connectable` interface, as shown with the extensions "Print active FDTS element" and "Display FDTS as graph". The implementation of a more general messaging system is left for future work. It can likely be based upon the existing implementation without many changes.

5.2 Verification: Plugin extension developer requirements

(see subsection 2.2)

5.2.1 Requirements for the framework

1. ✓The FDTS interface provides functions to access any part of an FDTS (including predicates)

2. ✓The FDTS interface allows creating and modification of nodes, edges, start/end nodes, predicates.

3. ✓Extension developers can easily transform a given FDTS into an FDTS of their own implementation.

5.2.2 Requirements for the IDE plugin

1. ✓The IDE plugin provides access to the IDE project and the IDE tool windows, which can be used to access most of the IDE functionality.

2. ✓Using the `Consumer` and `Transformer` interfaces, extensions can provide their functionality to the end user.

3. ✓As described in item 9 of subsection 5.1, the IDE plugin provides a system that extensions can use to communicate with other extensions.

5.3 Verification: SFL tool developer requirements

(see subsection 2.3)

While it is possible to extend the FDTS definition of the FDTS framework by adding predicate extensions (and thus being able to theoretically represent any kind of FDTS), it is not clear how practical this approach is. To test this, more advanced SFL tools and IDE plugin extension would need to be developed, which isn't in the scope of this work.

5.4 Using FLACOCO with the IDE plugin

To show that the FDTS framework is usable in practice, I developed a small application that translates the result of the FLACOCO[12] SFL tool into the FDTS JSON format that the FDTS framework (and thus also the IDE plugin) understands. FLACOCO is a spectrum-based tool based on the coverage Java library JaCoCo. The choice fell on FLACOCO, as, while possibly not being a very mature project, it seemed to work well with modern Java. I also considered using GZoltar[11], but there have been a few technical problems that would have made the development of the translator application unnecessarily time-consuming.

FLACOCO provides a Java API to run the spectrum-based fault localization algorithm and for then retrieving the results. These results in form of a map/dictionary from source code location to suspiciousness score can be used to create a simple FDTS in JSON format, looking, for example, like this:

```json
{
  "info": {
    "sus": 0.8164965809277261
  },
  "fdts": {
    "type": "FDTSImpl",
    "nodes": [
      {
        "type": "AtLine",
        "line": 59,
        "path": "./src/main/java/TestcaseForTracer.java"
      }
    ],
    "edges": [
      [
        null
      ]
    ],
    "startNodes": [
      {
        "id": 0
      }
    ],
    "endNodes": [
      {
        "id": 0
      }
    ]
  }
}
```

Each suggestion by FLACOCO (for where to look for a bug) consists only of a source code line with a floating point number describing how likely FLACOCO thinks that this line is connected to a bug. Thus, the FDTS only has a single node, being start and end node, that has as predicate only the requirement of being at the respective source code line.

Using a constructed code example (see Listing 2) and corresponding unit test (see Listing 3), the extended FLACOCO tool creates an array of such FDTS JSON objects. These can be loaded by the IDE plugin and successfully used with the various existing extensions such as the tracer.

6 Conclusion

In this work I described the design and implementation of an SFL result framework and an IDE plugin that serve as an interface between different SFL tools and a programmer who is trying to locate bugs. The system is constructed to allow for high flexibility in the usage of different SFL tools and additional tools that process SFL results. Multiple extensions for the IDE plugin have been developed to demonstrate the features and interfaces provided by the system.

The framework and IDE plugin satisfy a number of self-set requirements that have been deemed necessary to address the issues that plague the practical application of SFL tools, such as missing standardization. The approach has been to a certain extent validated by using the IDE plugin together with spectrum-based fault localization tool results.

6.1 Future Work

While the framework is mature enough to be targeted by SFL tools and extensions, some minor requirements are still to be fully implemented before the system can be used in a real development environment, such as the communication system between extensions in the IDE plugin. Additionally left for future work are the development of more extensions/improvement of existing ones and testing them for usefulness in a development environment, as well as extending more SFL tools to be compatible with the FDTS framework.

The current framework and IDE plugin only cover processing of the results that SFL tools produce. Still open is the question of how the setup and execution of SFL tools can be standardized and integrated into this or another system.

As mentioned in subsection 2.3, it is not clear if the FDTS language is powerful enough to be a good format for most SFL tools, including existing ones but also potentially future SFL tools. For example, representing the `x-causes-y` relationships that the delta debugging approach (subsubsection 1.1.3) produces as output, doesn't seem to be trivial without modifying the FDTS definition or adding special predicates.

References

[1] The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. `https://clang-analyzer.llvm.org/`. Accessed: 2022-01-26.

[2] GDB: The GNU project debugger. `https://www.sourceware.org/gdb/`. Accessed: 2022-01-26.

[3] A Java serialization/deserialization library to convert Java Objects into JSON and back. `https://github.com/google/gson`. Accessed: 2022-01-26.

[4] IntelliJ IDEA: IDE for JVM. https://www.jetbrains.com/idea/. Accessed: 2022-01-26.

[5] JGraphX is a Java Swing diagramming (graph visualisation) library. https://github.com/jgraph/jgraphx. Accessed: 2022-01-26.

[6] 2021 the state of software code report. https://content.rollbar.com/hubfs/State-of-Software-Code-Report.pdf. Accessed: 2022-01-26.

[7] T. Britton, L. Jeng, G. Carver, T. Katzenellenbogen, and P. Cheak. Reversible debugging software "Quantify the time and cost saved using reversible debuggers", 11 2020. Accessed at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf 2022-01-26.

[8] H. Krasner. The cost of poor software quality in the US: A 2020 report. https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf, 01 2021. Accessed: 2022-01-26.

[9] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Softw. Engg.*, 7(1):49–76, 03 2002. ISSN 1382-3256. doi: 10.1023/A:1014823126938.

[10] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39, 2003. doi: 10.1109/ASE.2003.1240292.

[11] A. Riboira and R. Abreu. The gzoltar project: A graphical debugger interface. pages 215–218, 09 2010. ISBN 978-3-642-15584-0. doi: 10.1007/978-3-642-15585-7_25.

[12] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus. FLACOCO: fault localization for Java based on industry-grade coverage. *CoRR*, abs/2111.12513, 2021. URL https://arxiv.org/abs/2111.12513.

[13] W. E. Wong and Y. Qi. Bp neural network-based effective fault localization. *Int. J. Softw. Eng. Knowl. Eng.*, 19:573–597, 2009.

[14] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, page 1–10. Association for Computing Machinery, 2002. ISBN 1581135149. doi: 10.1145/587051.587053.

7 Appendix

Listing 2: A constructed program to test the IDE plugin. It is assumed that a bug happens when the function `main2` returns 1.

```
import static java.lang.Integer.parseInt;

public class TestcaseForTracer {

    public static void main(String [] args){
        System.out.println(main2(args));
    }

    public static int main2(String [] args) {
        if(args.length < 2)
            return 0;

        var x = parseInt(args[0]);
        var a = args[1];
        int i = x + a.length();

        if(a.length() > 100)
            return 0;

        var c = false;

        for(; i>0; i/=2){
            if(i == 7)
                c = true;
            if(i == 5)
                c = true;
        }

        boolean maybeBug;

        if(c) {
            for (var ch : a.toCharArray()) {
                if (x == parseInt("" + ch)) {
                    x = -1;
                }
            }

            x = x * 3;
        }
        maybeBug = x != -3;
        if(!c){
            maybeBug = false;
            while (true){
                if(x == 1)
                    return 0;
                if(x == 0){
                    maybeBug = true;
                    break;
                }
                if(x<0)
                    x += 1;
                else
                    x -= 1;
            }
        }

        if(maybeBug)
        {
            var z = x % 2;
            if(z == 0)
                return 1;
        }

        return 0;
    }
}
```

Listing 3: The JUnit 5 test case code for Listing 2. `test3` und `test` will fail (i.e. the tested function will return 1 instead of 0).

```
public class Testing {
    @Test
    public void test0(){
        assertEquals(
            TestcaseForTracer.main2(new String[]{"5", "12"}),
            0
        );
    }
    @Test
    public void test1(){
        assertEquals(
            TestcaseForTracer.main2(new String[]{"7", "072"}),
            0
        );
    }
    @Test
    public void test2(){
        assertEquals(
            TestcaseForTracer.main2(new String[]{"12", "4472"}),
            0
        );
    }
    @Test
    public void test3(){
        assertEquals(
            TestcaseForTracer.main2(new String[]{"10", "1234"}),
            0
        );
    }
    @Test
    public void test4(){
        assertEquals(
            TestcaseForTracer.main2(new String[]{"-12", "1234"}),
            0
        );
    }
}
```

7.1 IDE plugin code

The code of the FDTS framework and the IDE plugin can be found here: `https://gitlab.com/tsoj/sfl_server`.

7.2 FLACOCO converter code

The code for the FLACOCO converter can be found here: `https://gitlab.com/tsoj/flacoco_fdts_converter`.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den February 1, 2022 ....