UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Android App Analysis
University of Paderborn
Warburger Str. 100
33102 Paderborn

Software Architecture Document

Paderborn, July 31, 2015

Authors:

Abhinav Solanki	Anand Devarajan
Arjya Shankar Mishra	Fabian Witter
Felix Pauck	Monika Wedel
Pham Thuy Sy Nguyen	Ram Kumar Karuppusamy
Sriram Parthasarathi	

Contents

1 Introduction

In this document we are going to present the architecture of our Android App Analysis tool, which we are going to develop in the next months. After giving an overview in Section 2.1 how the workflow of the tool will look like we are introducing the overall structure by presenting the different components the tool consists of (see Sec. 2.2). Furthermore, we describe in Section 2.3 how the introduced components interact for the analysis Level 1, 2a and 2b that are supported by the tool.

Section 3 deals with the Soot framework that will be used in the tool. There the features the tool uses are described in general as well as implementation details are mentioned. Afterwards, in Section 4 we are refining the structure of the components by introducing class diagrams for the important classes and describing their structure. This Section is partitioned into five subsections starting with the overall framework in Section 4.1 and then continuing with the level specific classes. The last subsection then deals with the structure of the client component (see Sec. 4.5).

After explaining the detailed structure we present sketches and the workflow of our tool's user interface (see Sec. 5). In the end we present the time schedule for the remaining development time in Section 6.

In this document we use a special formatting for `classes`, `methods`, `attributes` as well as for `components` to clearly show that the words reflect concrete classes, methods, etc.

Since this document directly follows the Target Level Agreement in the time line of submitted documents, we especially want to refer to some sections at this point, which answer questions that were still open in the Target Level Agreement.

- The way of parsing the source code will be explained in Section 3 which deals with Soot.
- Distinction between having two different Apps or two versions of the same App will be done by means of the `XMLParser` and the `AppIdentifier` which are described in Section 4.1.3 and Section 4.5.
- The parts we will use Soot for can be found looking at Section 3 as well as Section 4.1.4 and Section 4.3.2.
- As interprocedural Information Flow Control (IFC) analysis for Level 2a we decided to implement an IFC analysis using Program Dependence Graphs(PDG). Detail concerning this analysis are described in Sec. 4.3.

2 High-Level Architecture

This section presents in subsection 2.1 the general workflow of the tool by showing in which order which activities are executed. Afterwards, we introduce the components of the tool and interfaces which connect them in subsection 2.2. Finally, the communication between the components is presented in subsection 2.3.

2.1 General Workflow of the Tool

The following section is a description on general workflow of our Android App Analysis tool in the form of activity diagram. The activity diagram(see Figure 2) follows the basic step by step action performed by our Android App Analysis tool.

The first and foremost step is 'How to start our Android App Analysis tool?', since our analysis tool can be started both in command prompt and through the User Interface(UI). The user can choose one of the described possibilities based upon his/her convenience. After starting the application the user has to select the Level(1, 2a and 2b) of analysis he/she want to perform (see Figure 1). The analysis performed in each level differ between them, which will be described in detailed bellow. Then the user must select any one of the mode(SUMMARY or COMPARISON) and provide input(s) to perform the analysis. The input(s) should always be an `.apk` file or a previous analysis result and the number of input varies for each level and the mode the user selects.

In Level 1 and Level 2a SUMMARY mode the user should provide one `.apk` file as input, but in Level 1 and Level 2a COMPARISON mode the first input will be one `.apk` and the second input will be a previous analysis result.

The Level 1 is used to find out different groups of permissions(REQUIRED, MAYBE_-REQUIRED, UNUSED, MISSING, MAYBE_MISSING) in App, Component, Method, and Class of an application. If the user is performing the Level 1 SUMMARY, the input `.apk` is passed to the `XMLParser`. The `XMLParser` unzips the `.apk` file and extracts the manifest permissions in one end. In the other end, the same apk is passed into soot and generate a jimple model, Simultaneously the tool gets the API's information which will be stored in `DataStorage`, the API's defines the permissions of many versions of Android library. This information is used to link the layers and map the permissions that are used. In the next step the tool tracks the explicit intent that are used in the statement and links them to generate a graph. Then as the final step of the analysis, the tool will compare the manifest permission which is obtained previously with the generated graph and categorise the permissions into five different group(REQUIRED, MAYBE_REQUIRED, UNUSED, MISSING, MAYBE_MISSING).

Level 2a is used to find out the data flow between use of the permissions, Level 2a SUMMARY mode performs the same steps as Level 1 (see Figure 3)until the tool builds a program dependence graph by analyzing explicit intents and modelling control dependencies and data flow. For this, an information flow control technique is used. By means of the graph and the permissions declared in the manifest file, the tool finds the tool detects which statements are sources and sinks and analyzes the program dependence graph to detect paths from sources to sinks. Finally the result is generated for different detail levels (Control Flow, Statement Flow and Resource to Resource).

When the user is performing the Level 1 and Level 2a COMPARISON mode, then the same procedure similar to SUMMARY mode is followed for the first `.apk` input and a result is

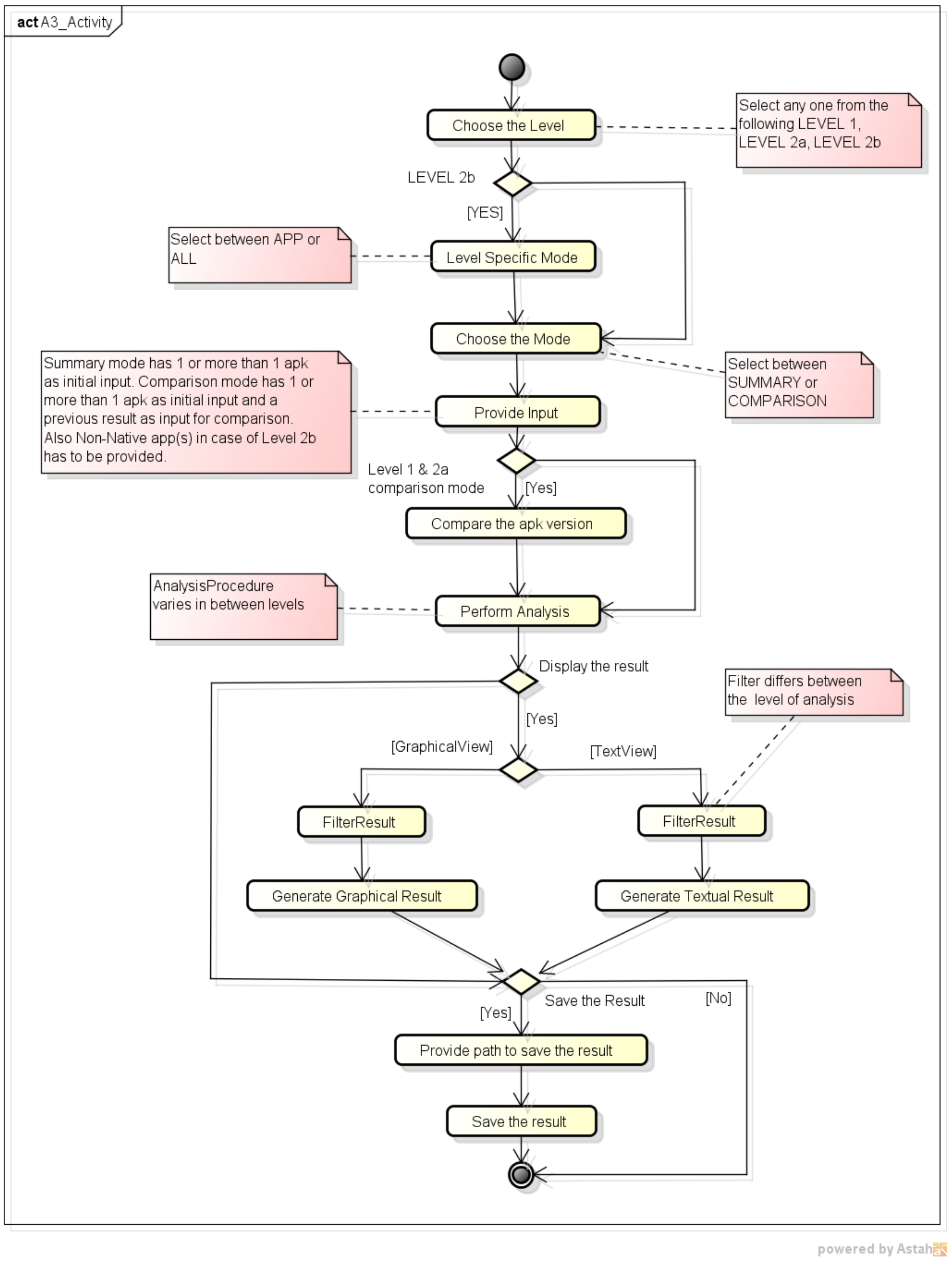
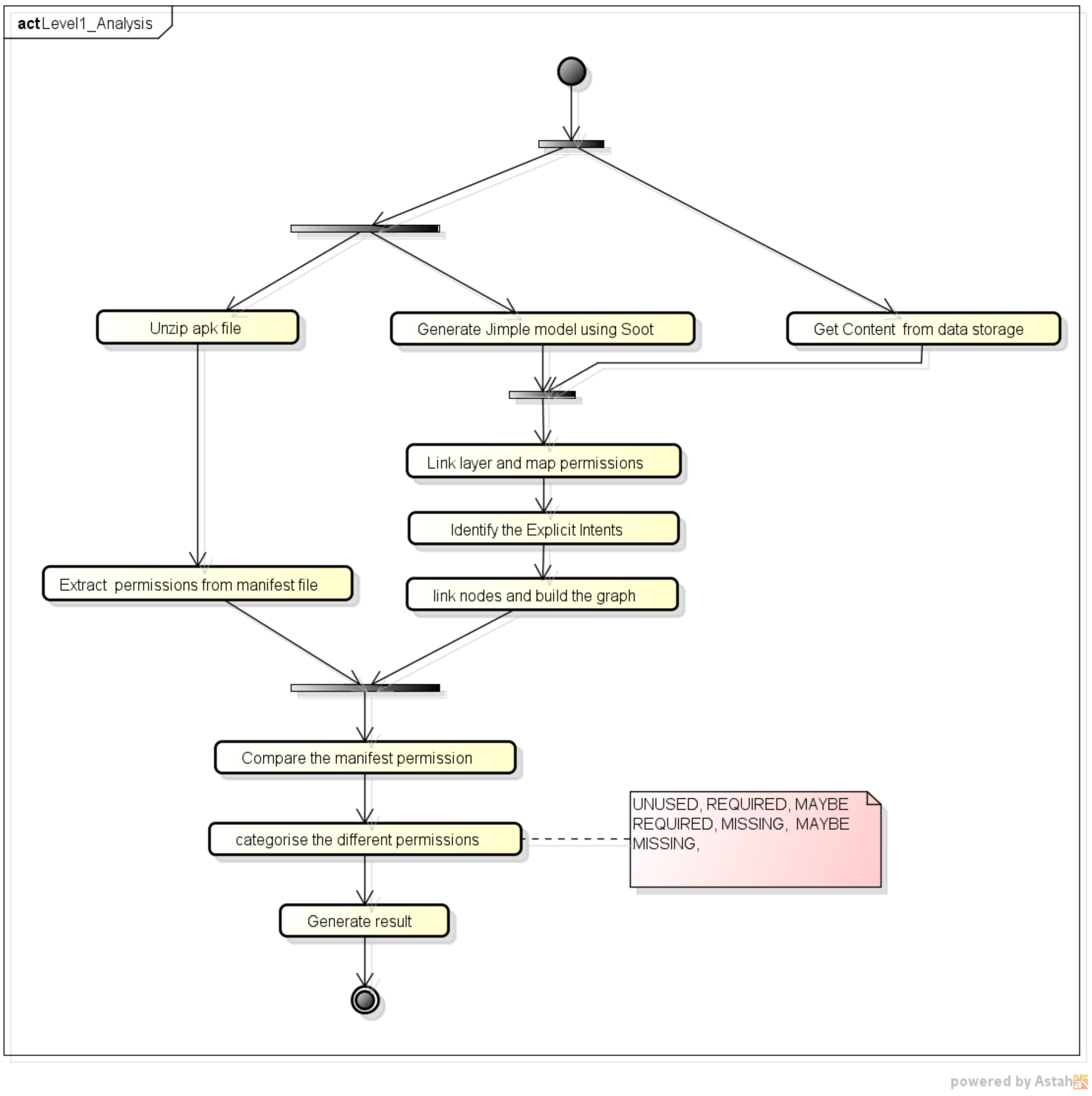

act A3_Activity
Choose the Level
Select any one from the following LEVEL 1, LEVEL 2a, LEVEL 2b
LEVEL 2b
[YES]
Select between APP or ALL
Level Specific Mode
Choose the Mode
Select between SUMMARY or COMPARISON
Summary mode has 1 or more than 1 apk as initial input. Comparison mode has 1 or more than 1 apk as initial input and a previous result as input for comparison. Also Non-Native app(s) in case of Level 2b has to be provided.
Provide Input
Level 1 & 2a comparison mode
[Yes]
Compare the apk version
AnalysisProcedure varies in between levels
Perform Analysis
Display the result
[Yes]
Filter differs between the level of analysis
[GraphicalView]
[TextView]
FilterResult
FilterResult
Generate Graphical Result
Generate Textual Result
Save the Result
[Yes]
[No]
Provide path to save the result
Save the result
powered by Astah

Figure 1: Activity Diagram for General Workflow



actLevel1_Analysis
Unzip apk file
Generate Jimple model using Soot
Get Content from data storage
Link layer and map permissions
Identify the Explicit Intents
Extract permissions from manifest file
link nodes and build the graph
Compare the manifest permission
categorise the different permissions
UNUSED, REQUIRED, MAYBE REQUIRED, MISSING, MAYBE MISSING,
Generate result
powered by Astah

Figure 2: Activity Diagram for Level 1 Analysis

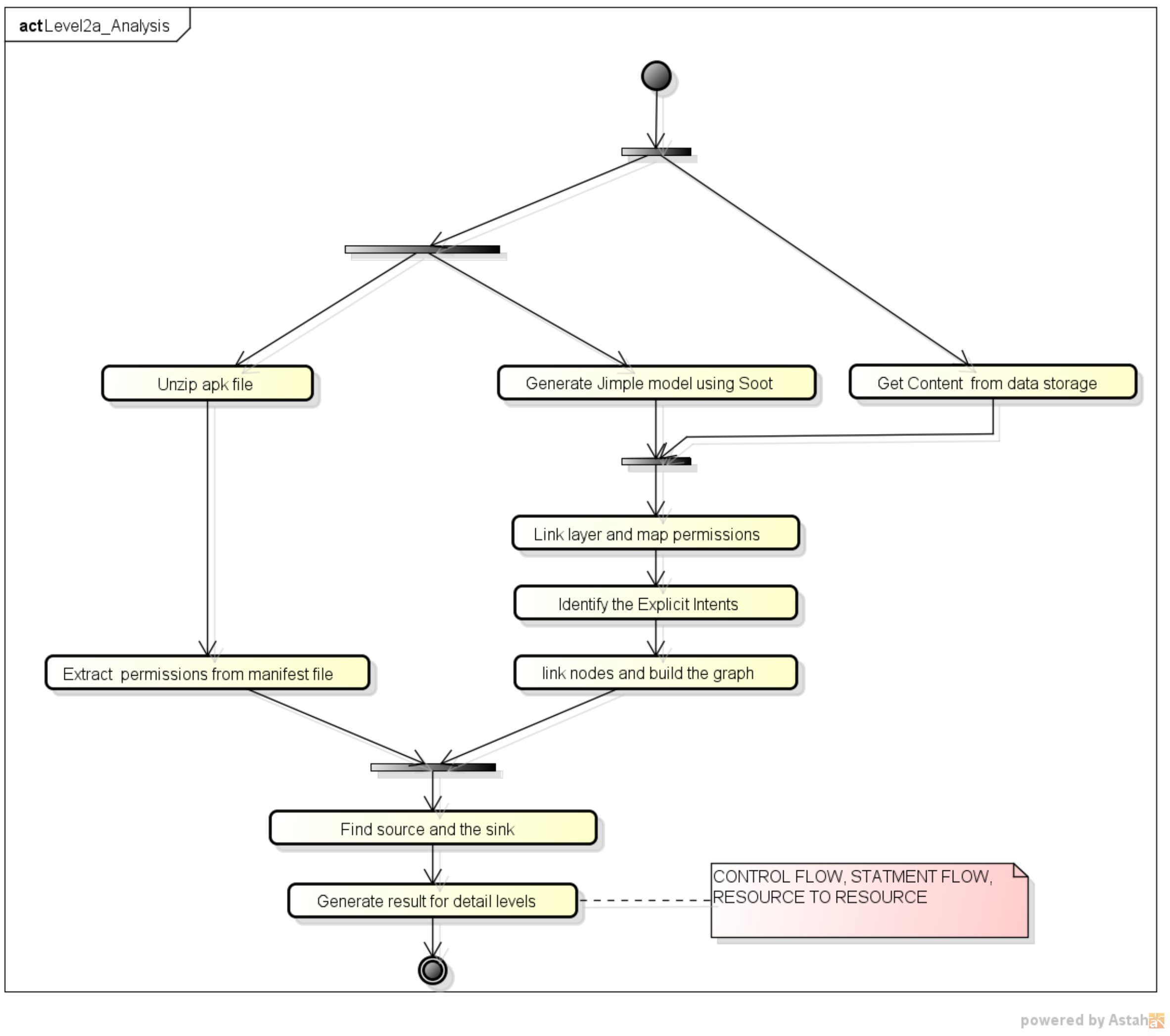

act Level2a_Analysis
Unzip apk file
Generate Jimple model using Soot
Get Content from data storage
Link layer and map permissions
Identify the Explicit Intents
Extract permissions from manifest file
link nodes and build the graph
Find source and the sink
Generate result for detail levels
CONTROL FLOW, STATMENT FLOW, RESOURCE TO RESOURCE
powered by Astah

Figure 3: Activity Diagram for Level 2a Analysis

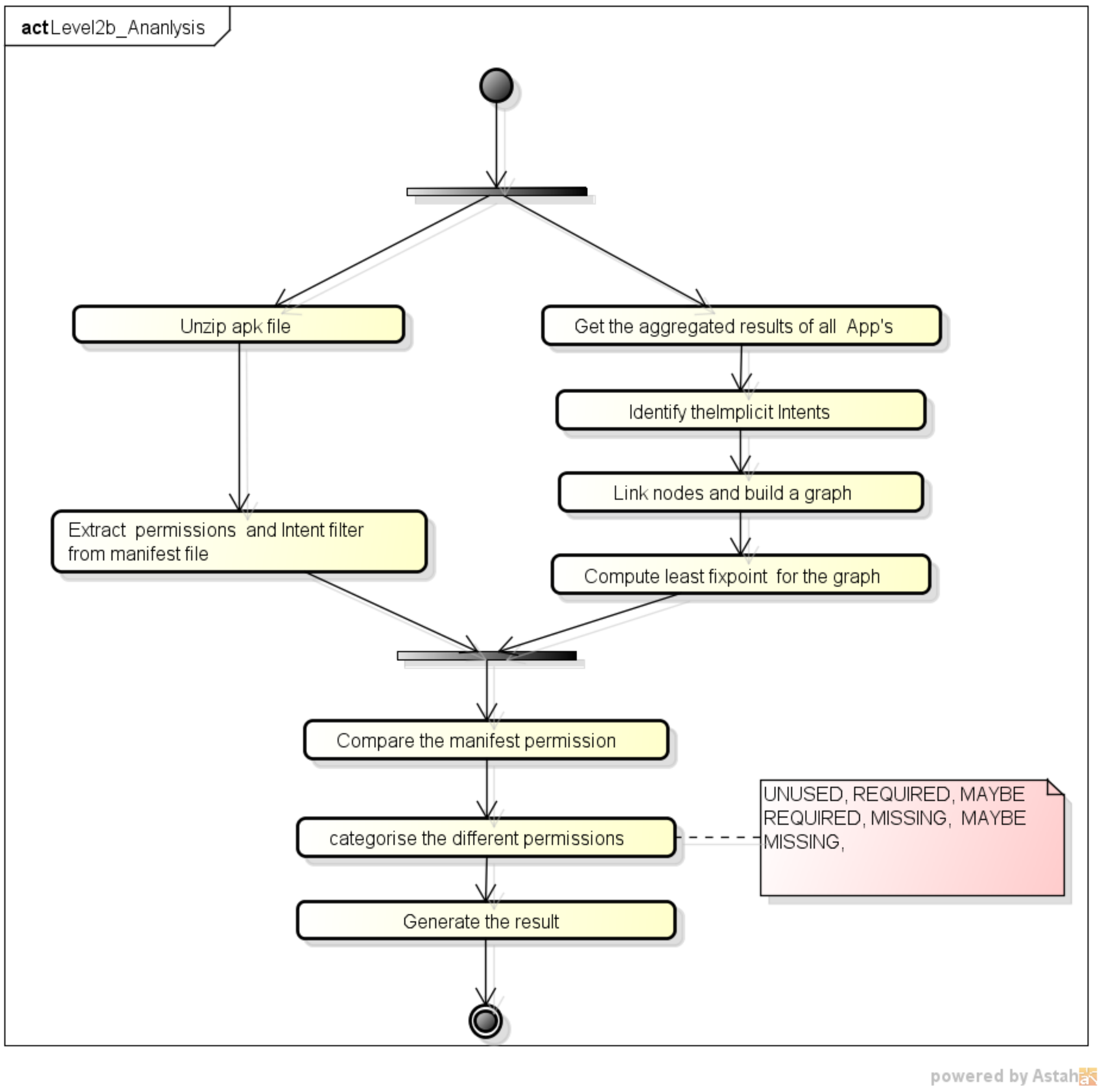

act Level2b_Ananlysis
Unzip apk file
Get the aggregated results of all App's
Identify theImplicit Intents
Link nodes and build a graph
Extract permissions and Intent filter from manifest file
Compute least fixpoint for the graph
Compare the manifest permission
categorise the different permissions
UNUSED, REQUIRED, MAYBE REQUIRED, MISSING, MAYBE MISSING,
Generate the result
powered by Astah

Figure 4: Activity Diagram Level 2b Analysis

generated and the previous result that was specified as input is compared with the just generated result. Then the final compared result is generated.

Level 2b analysis is the extension of Level 1, so in addition Level 2b will compute the resource used by an App via other Apps, which can be done by tracking implicit intent. During Level 2b analysis the user has to select one more additional level specific mode (APP and ALL) with already existing mode (SUMMARY and COMPARISON). In APP mode the tool will compute the resource usage of an App including the indirect resource usages via system App and other non native Apps(Apps that are not supported by android system environment). In ALL mode the tool will consider all Apps of the input set as starting point. So the user has to select a Cartesian product over two set of modes, which are SUMMARY,COMPARISON and APP,ALL. So the range of inputs varies between modes, which can be categorised in the following cases.

1. **SUMMARY and APP**: one `.apk` and non- native `.apk(s)`.
2. **SUMMARY and ALL**: more than one `.apks` and non-native `.apk(s)`.
3. **COMPARISON and APP**: one `.apk`, non-native `.apk(s)` and the previous Level 2b analysis result.
4. **COMPARISON and ALL**: more than one `.apks`, non-native `.apk(s)` and the previous Level 2b analysis result.

During Level 2b SUMMARY mode, the tool does Level 1 analysis for each and every individual `.apks` and the results are generated. The aggregated result is once again analysed for any implicit intent(see Figure 4) between the `.apks`, to find out the resource usage of an App via other Apps. The tool links the nodes and generates another graph linking the different `.apks` if there is any implicit intent connecting them. Simultaneously the tool will extract the declared manifest permission especially from the intent filter and compare these permissions with the graph generated from the aggregation of all results to categorise the permissions into five different groups(REQUIRED, MAYBE_REQUIRED, UNUSED, MISSING, MAYBE_MISSING).

In Level 2b COMPARISON mode the first input is `apk(s)` and the second input is the previous Level 2b analysis result. These two are compared and the permission are grouped into five different groups.

Depending on the Level(1, 2a and 2b) of analysis, the result can be viewed by the user in either graphical or textual representation. The user can also use filters to have a closer look at a subset of the detected permissions, the filters vary for each Levels. If wanted the user can save the analysis result or start a new analysis without saving.

2.2 Component Architecture

In the following the high-level architecture of our tool in form of a component diagram will be introduced and described. The component diagram is given in Figure 5.

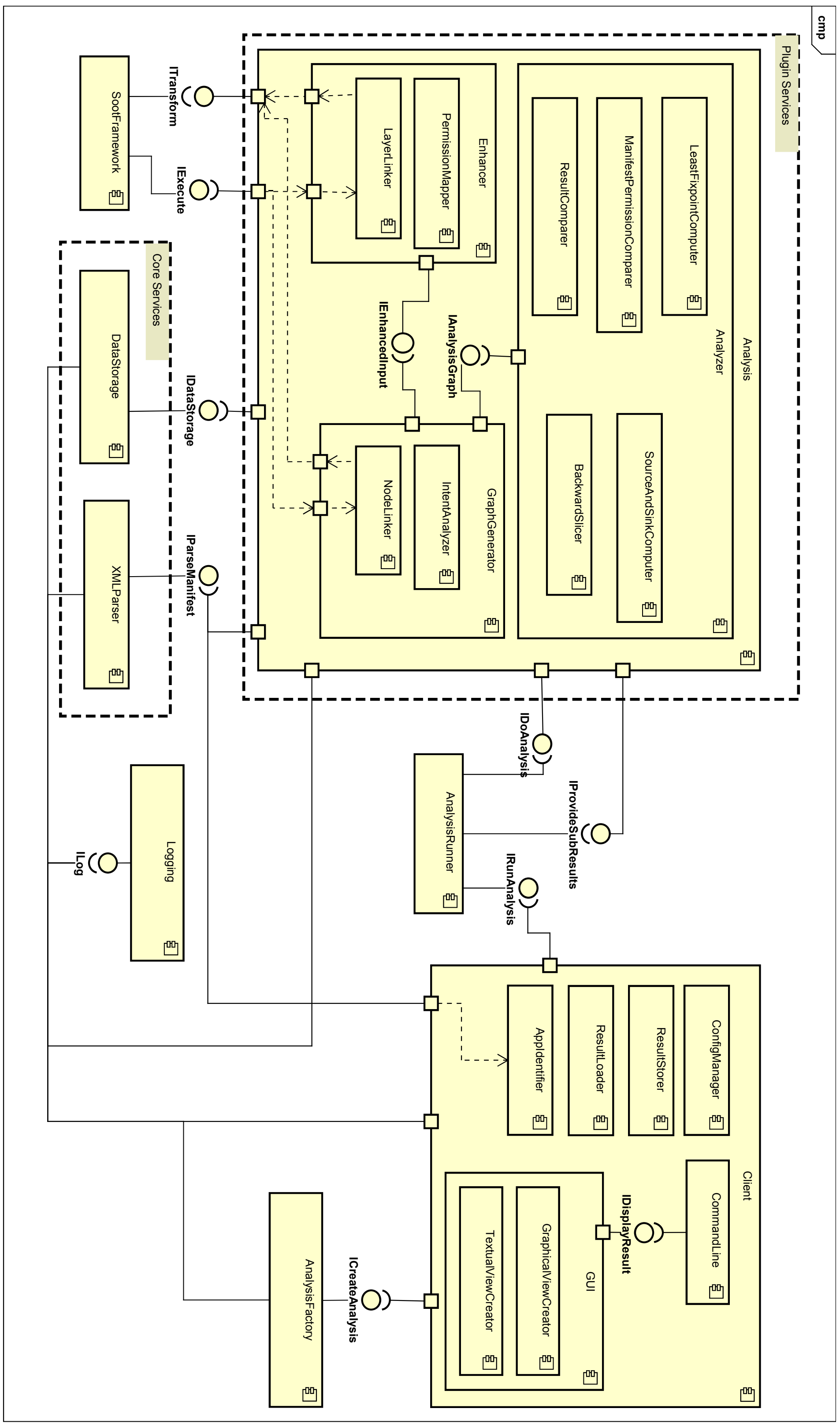


Figure 5: Component Diagram of the Android App Analysis Tool

On the left, the main analysis logic is modeled in the `Analysis` component while the user interfaces and functionality that belong to the user are contained in the `Client` component on the right. The `Analysis` is meant as a Plugin Service for our tool and, hence, will be extensible and exchangeable such that our tool can be extended with additional analyses easily in the future. Moreover, the analysis logic as a whole can be easily used by arbitrary clients and though is very portable. Our client will be a Desktop computer application but due to interchangeability the analysis logic can also be used for example within a web application.

At the center of the diagram is the central `Logging` component which provides the `ILog` interface for all components to write status logs to console or file. Furthermore, there is the component `AnalysisRunner` which will perform the analysis for the `Client`. The `Client` can trigger the analysis via the `IRunAnalysis` interface.

Another helper for the `Client` is the component `AnalysisFactory`. This will create an adequate analysis for the `Client` via the `ICreateAnalysis` interface which can then be run by the `AnalysisRunner`.

The `Analysis` component contains the whole analysis logic. It provides the two interfaces `IDoAnalysis` and `IProvideSubResults` to the `AnalysisRunner`. `IProvideSubResults` provides results from other analyses which is needed for Level 2b, that builds upon Level 1, and similar analyses. For the project group we structured each analysis in three processing steps `Enhancer`, `GraphGenerator` and `Analyzer` but in general the `Analysis` may have an arbitrary internal structure. Below the `Analysis` are the Core Services in form of the components `XMLParser` and `DataStorage`. The `XMLParser` provides access to the information of Android Manifest Files via the `IParseManifest` interface. Additionally, the `DataStorage` provides mappings from android system intents and calls to the according permissions which can be retrieved through the `IDataStorage` interface. Both Core Services may be used by the `Analysis` and its inner components. In our diagram we decided to go without the delegations of the Core Service's interfaces to the inner components to achieve a clean and comprehensible overview. Details on the usage of the Core Services will be provided later on in Section 4.

Next to the Core Services lies the `SootFramework` component. This component represents the plain Soot Framework which our tool is based on. Parts of the `Analysis` will be extensions to Soot and therefore Soot will be executed via the interface `IExecute` within the `Analysis`. The extensions provide the `ITransform` interface for Soot to work with.

Inside the `Analysis` the `Enhancer` component is the first step in our analyses. First, the subcomponent `LayerLinker` extracts a hierarchical representation of the source code of an Android App with the help of Soot. Next, the subcomponent `PermissionMapper` then adds permissions to statements in code where such are used and propagates them to the higher elements in the hierarchy, too.

The `Enhancer` passes its intermediate result to the `GraphGenerator` via the `IEnhancedInput` interface. This component will add new directed transitions between code elements, e.g. flow edges or call edges. Therefore, the subcomponent `IntentAnalyzer` checks the call of other Android components via intent and the subcomponent `NodeLinker` adds the kinds of edges discovered by the `IntentAnalyzer` and optionally by the Soot framework.

As the last step the `Analyzer` receives the computed graph through the `IAnalysis-Graph` interface from the `GraphGenerator`. The `Analyzer` contains a rather huge set of components for computing the final result. This is a collection of components needed for all three analysis levels that will be implemented by us and not all of them will be needed in each level. The only exception is the component `ResultComparer` which will compare a previous result with the newly computed result in `COMPARISON` mode for all levels. The other components are used as follows:

- For Level 1 the component `ManifestPermissionComparer` will compare the usage permissions that are specified in the Android manifest with those computed by the `Enhancer`
- For Level 2a the components `BackwardSlicer` and `SourceAndSinkComputer` will determine the information sources and sinks in the set of used permissions and then perform the backward-slicing algorithm to compute information flow from sources to sinks
- For Level 2b the component `LeastFixpointComputer` will compute the resource usage between multiple Apps with the least-fixpoint algorithm

Finally, the details of the `Client` component will be described. The `Client` provides two user interfaces. First, the component `GUI` allows the user to configure and trigger analyses and view their results in a graphical user interface. The subcomponents `TextualViewCreator` and `GraphicalViewCreator` process the result from the analysis into a textual and graphical representation that is easily comprehensible for the user. In addition, the user can trigger analyses via the command line which is specified in the `CommandLine` component. The `CommandLine` can trigger the `GUI` via its `IDisplayResult` interface to graphic result representations.

The `Client` also contains some solely functional components. The component `Con-figManager` is responsible for loading analysis configurations from files and merging them with possible configurations from the user interface. Moreover, the `ResultLoader` and `ResultStorer` components provide the ability to load previous analysis results from files and also to store those results to files. In the end, the component `AppIdentifier` identifies an Android App with the help of the `XMLParser` due to its fingerprint. This is necessary for `COMPARISON` mode since only different versions of the same App should be compared.

Like above for the `Analysis` we decided not to integrate most intra-component interfaces for the `Client` for a better overview. How user interface and functionality components interact will be explained in Section 4.

2.3 Interaction between the Components

The following subsections illustrate how the Components introduced in Sec 2.2 interact while doing the supported analysis level. The sequence diagrams especially show how the components interact with the components of `Analysis`. The way the `Analysis` communicates with its inner components depends on the level of analysis to be executed and it will be described later in this document (see Sec. 4).

Since the communication on this level of abstraction is almost the same for level 1 and 2a and different for level 2b,in the following two sections we have provided the descriptions for handling these two cases.

2.3.1 Level 1 and 2a

During Level 1, the analysis will focus on the usage of permissions in a single App and its comparison with a previous analysis result. Here, the tool will need one `.apk` file as input and a previous analysis result in case if COMPARISION mode is selected. The result presented to the User will subdivide all detected permissions into five classes as specified in the Requirement Specification Document: `REQUIRED, MAYBE REQUIRED, UNUSED, MAYBE MISSING` and `MISSING` and optionally compare it with a previous analysis result.

The Level 1 analysis can be performed in two ways, either we can perform a fresh analysis of an`.apk` file or we can also compare one `.apk` file with the previous saved result. So, it basically comprises two modes SUMMARY and COMPARISON.

The analysis of this level is described generally in a high level sequence diagrams in Figure 6 and then for each application mode (SUMMARY or COMPARISON), the detail processing is represented accordingly in high level sequence diagrams in Figure 7 and in Figure 8. These diagrams shows the sequence of actions taking place, how the task is performed, which steps it needs to undertake a complete execution, In the next paragraphs, the detailed functioning of the sequence diagram with respect to both modes is explained.

The general sequence diagram in Figure 6 describes a possible interaction of User and the Android App Analysis tool as well as the interaction between components inside the App. The User interacts to the Android App Analysis tool through the `Client` component. The sequence activities are listed step-by-step below:

- **Step1**: In first step, the User configures the analysis. The User selects Level 1 or 2a, SUMMARY or COMPARISON mode on GUI or specifies them by option parameters in CMD-Line. Despite using the App in GUI or CMD-Line mode, the User still mainly interacts to the `Client` component.
- **Step2**: The User specifies the necessary input (`.apk` file) and a previous analysis result(optional) in case of COMPARISON mode through the `Client` component.

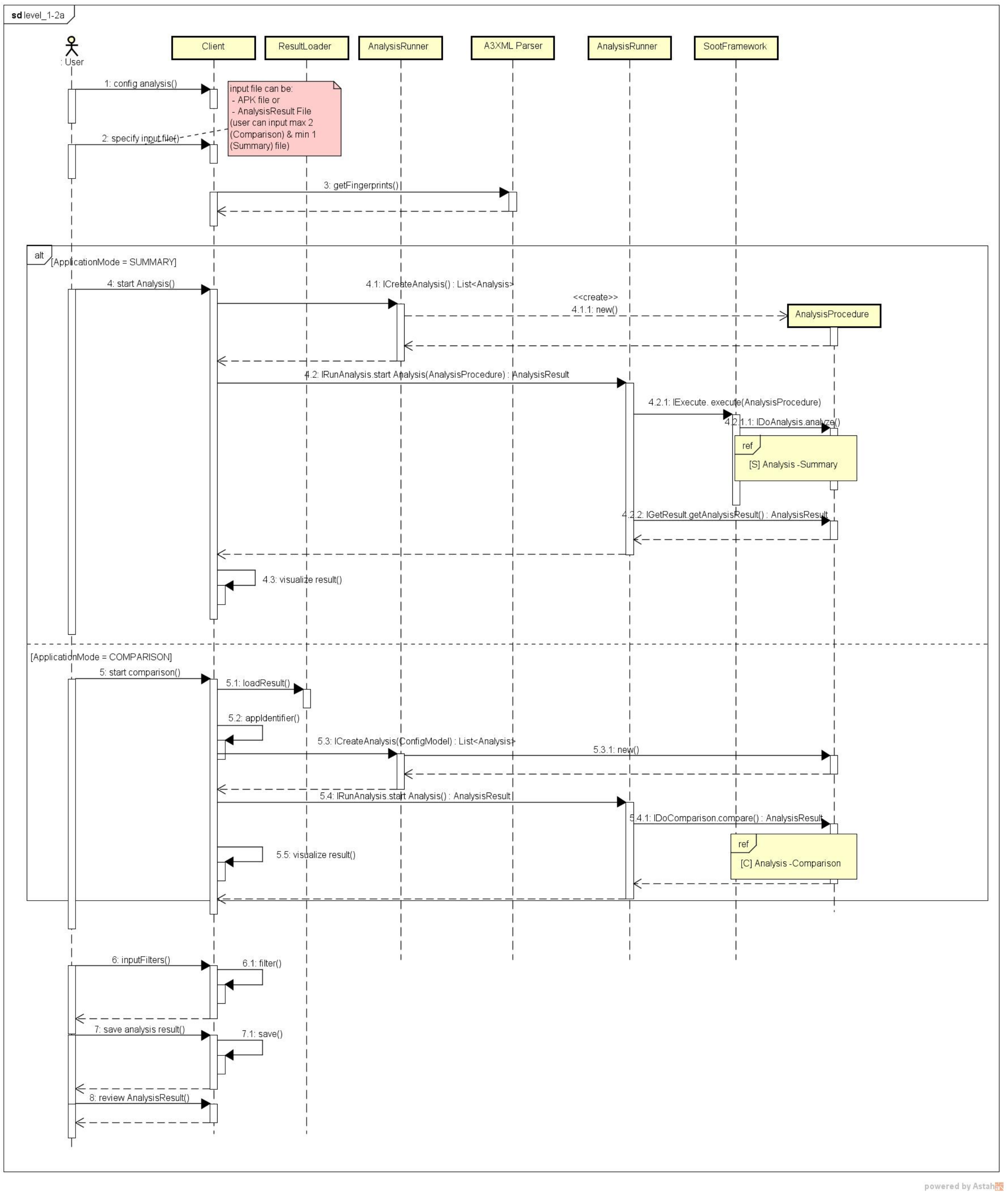


Figure 6: High Level Sequence Diagram for Level 1 and 2a Analysis

- **Step3**: The User then performs the analysis with the specified configuration and the provided input by invoking appropriate functions supported by the `Client` component. For different modes (SUMMARY and COMPARISON) the analysis is also different. It is described clearly in sections **SUMMARY Mode** and **COMPARISON Mode** below. After the analysis is done, the result will be visualized by the `Client` component.
- **Step4**: In case of using GUI mode, the User can filter the result based on defined criteria offered by `Client` component.
- **Step5**: The User can also choose saving the result(for future use) or not as an additional option. If the User wants to save it, a location should be specified by the User.

  The `User` also can load a previous analysis result for reviewing by specifying the file to the `Client` component. The component will parse the file and visualize the result to the User.

SUMMARY Mode The analysis performed during this phase is explained in Figure 7 on a high level. This mode will start with an `.apk` file as input to the tool, and then the analysis mechanism will start.

1. First of all, the `Client` in turn interacts with the `AnalysisFactory` using the `ICreateAnalysis` interface to get an appropriate `Analysis` to analyze the input files.
2. After that, `A3XMLParser` is used to extract fingerprints of the App owner from the `.apk` file. Thereafter `AppIdentifier` is called to check whether the`.apk` file is from the same owner or from different owner, whether they are same, versioning and so on.
3. Next, the `Client` passes the input `.apk` file to the `AnalysisRunner` by using the interface `IRunAnalysis` and invoke the analysis function.
4. The `AnalysisRunner` then uses the interface `IExecute` supported by the `Soot-Framework` to continue the analysis.
5. The `SootFramework` component continues the analysis by calling provided functions in interface `IDoAnalysis` of component `AnalysisProcedure`.
6. The component `AnalysisProcedure` perform the analysis with these actions below:

   The component interacts with the `DataStorage` component to get all APIs, defined permissions of many versions of Android library. Those information will be used for mapping permissions and linking layers in the `Enhancer`.

   The component also interacts with the `A3XMLParser` to get all permissions defined in the manifest file of the input `.apk` files to build a permission model.

   Inside the component `Analysis`, first the `Enhancer` disassembles the `.apk` file and creates a data model, then it collects all the permissions, the information got from

DataStorage to map and link all of them together. Here layers might be Android Components, classes, methods ot statements. Next, the Enhancer provided all linked objects for the GraphGenerator to create a graph. The GraphGenerator also extracts intents from those linked objects by the IntentAnalyzer and uses the NodeLiker to build a graph for intents. The explicit graphs are passed to the Analyzer through the interface IAnalysisGraph provided by the GraphGenerator to process the last steps of the analysis procedure. Then the ManifestPermissionComparer is called, which compares the permissions provided by the .apk file with that of collected in analysis phase. After that , the result is segregated into defined subcategories as mentioned above.

After the analysis finishes, the AnalysisRunner uses the interface IProvideSub-Results to get the analysis result from the Analysis and returns it to the client. Here in the Client, the analysis result is visualized for the User's view.

COMPARISON Mode As described above that Level 1 Analyses has two modes, this mode is used to compare analyzed result from .apk file and a previous saved result. At this mode, after the User sets up the analysis in **Step1**, in the next step, User needs to give one .apk file and one previous analysis result file as input to the Client.

In addition, the User can also continues the analysis with COMPARISON mode after SUMMARY mode finishes. For this situation, the User just specifies the second input. They are specified through the Client component. Before the User perform the comparison analysis, the Client has to validate that all the input files belongs to the same application or not. It calls the AppIdentifier component for the same purpose. If the inputs belong to different applications, the Client will throw warning messages to the User. Otherwise, the User can continue the comparison analysis.

1. Similarily as in SUMMARY mode, the Client also uses the ICreateAnalysis interface (provided by the AnalysisFactory) to get appropriate Analysis for COMPARISON mode. Beside the Analysis for the mode, the AnalysisFactory also generates other Objects to analyze the input files if the first or the second input is .apk file.
2. Secondly, the Client component will analyze the input .apk file in SUMMARY mode to get the analysis result (see the section **SUMMARY Mode** for more detail). If the input is previous analysis result, the Client uses the ResultLoader to load the result in files. This step makes sure that before comparison analysis step, all the inputs are already analyzed to get the result.
3. After the AnalysisRunner collects all required result of the inputs, using the procedure got from AnalysisFactory it invokes the comparison function in the interface IDoComparisonAnalysis(provided by the AnalysisProcedure component) to compare the results.

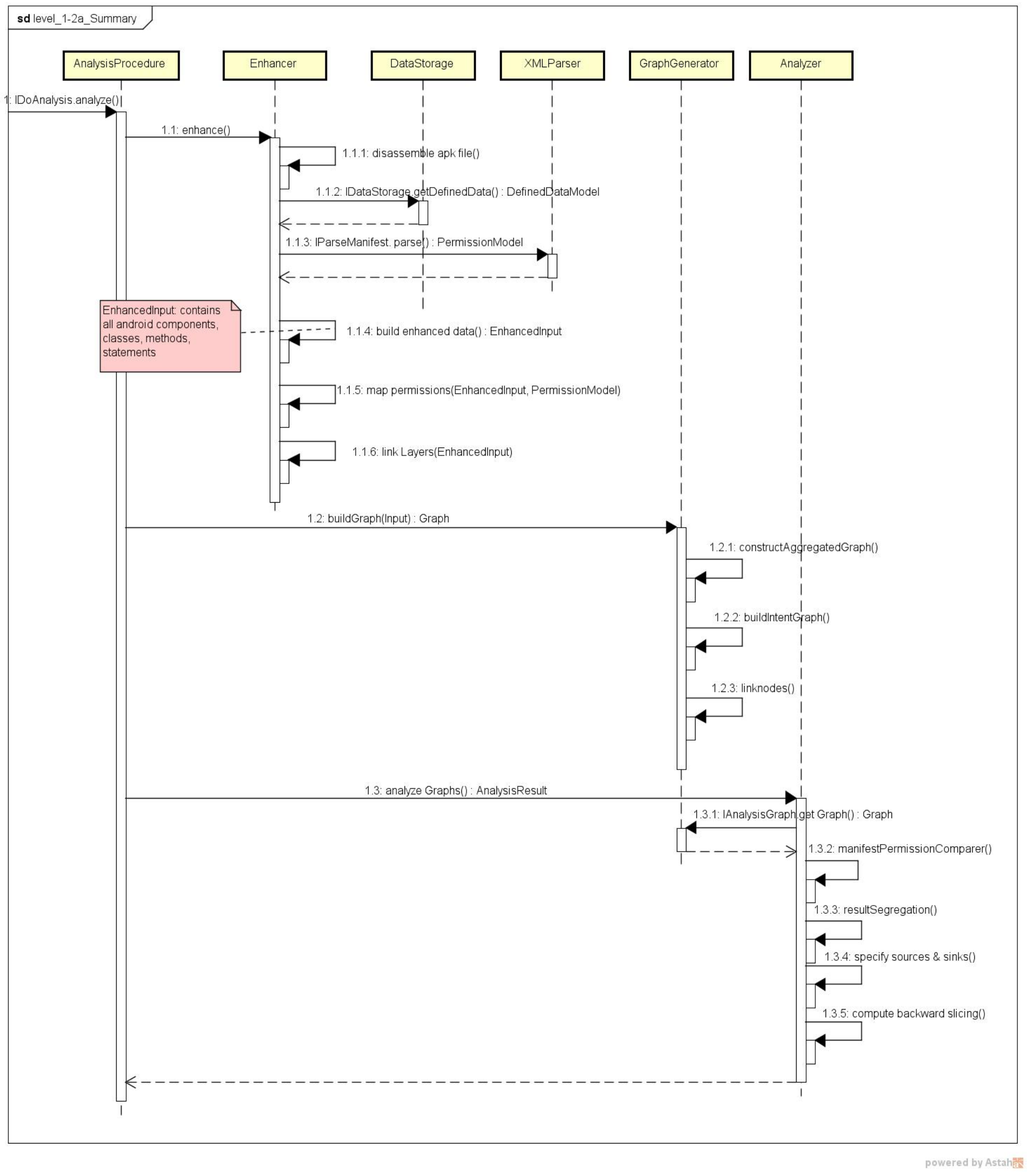

sd level_1-2a_Summary
AnalysisProcedure
Enhancer
DataStorage
XMLParser
GraphGenerator
Analyzer
1: IDoAnalysis.analyze()
1.1: enhance()
1.1.1: disassemble apk file()
1.1.2: IDataStorage.getDefinedData() : DefinedDataModel
1.1.3: IParseManifest. parse() : PermissionModel
EnhancedInput: contains all android components, classes, methods, statements
1.1.4: build enhanced data() : EnhancedInput
1.1.5: map permissions(EnhancedInput, PermissionModel)
1.1.6: link Layers(EnhancedInput)
1.2: buildGraph(Input) : Graph
1.2.1: constructAggregatedGraph()
1.2.2: buildIntentGraph()
1.2.3: linknodes()
1.3: analyze Graphs() : AnalysisResult
1.3.1: IAnalysisGraph.get Graph() : Graph
1.3.2: manifestPermissionComparer()
1.3.3: resultSegregation()
1.3.4: specify sources & sinks()
1.3.5: compute backward slicing()
powered by Astah

Figure 7: High Level Sequence Diagram for Level 1 and 2a Analysis - Summary

Inside the component `AnalysisProcedure`, the `ResultComparer` takes responsibility for comparing the input result and deriving the comparison result. Then the `AnalysisProcedure` returns the comparison result to `AnalysisRunner` for continuously passing it to the `Client` and visualizing it for User's view.

`User` can see the complete compared result from the two sources at client. Similar to the summary mode, user here also has the option to apply certain filters on the analyzed output result, after the filter is input, the `InputFilter` is passed and filtered result will be shown to the User. Here in the end, user will have an option to save the current analyzed result.

Level 2a Analysis

During **Level 2a** analysis, the processing focus on the interaction between components for intra-app information flow control analysis. In particular, based on the Component Diagram (Figure 5.)This level extends the functioning of Level 1 analysis. With respect to the main interactions happening between components inside the App in, the general sequence diagram in Figure 6 associates with the diagrams in Figure 7 and in Figure 8 for illustrating the processing in more detail.

The `User` interaction with the `Client` will be same as described above in level 1, the process during SUMMARY and COMPARISON modes are explained below-

SUMMARY Mode In this mode Figure 7, as soon as the User invokes analyzing function with specified configuration and provided input through the `Client` component. This mode extends the fucnctioning of the level 1 Analysis. It continues the processing of level 1 from `GraphGenerator`. The `GraphGenerator` also extracts Intents from those linked objects by the `IntentAnalyzer` and uses the `NodeLiker` to build a graph for Intents. The explicit graphs are passed to the `Analyzer` through interface `IAnalysisGraph` provided by the `GraphGenerator` to process for the last steps of analysis procedure. The `Analyzer` uses the `SourceAndSinkComputer` component to specify the sources and sinks in the graphs. Finally, the `BackwardSlicer` is applied to derive the result based on the graphs with sources and sinks. After the analysis finishes, the `AnalysisRunner` uses the interface `IGetResult` to get the analysis result from the `AnalysisProcedure` and returns it to the client. Here in the `Client`, the analysis result is visualized for the User's view.

COMPARISON Mode As described in Figure 8 above in Level 1 Analyses Comparison mode, this mode is used to compare analyzed result from `.apk file` and previous saved result.At this mode, after the User sets up the analysis in **Step1**, and in the next step, User needs to give one `.apk` file and one previous analysis result file as input to the `Client`.

In addition, the `User` can have some further comparisons between the analysed results , as we have extended our roots further in Level 2a Summary mode, so at this mode analysis upto Summary mode is carried and then the comparison is carried out between the current result and

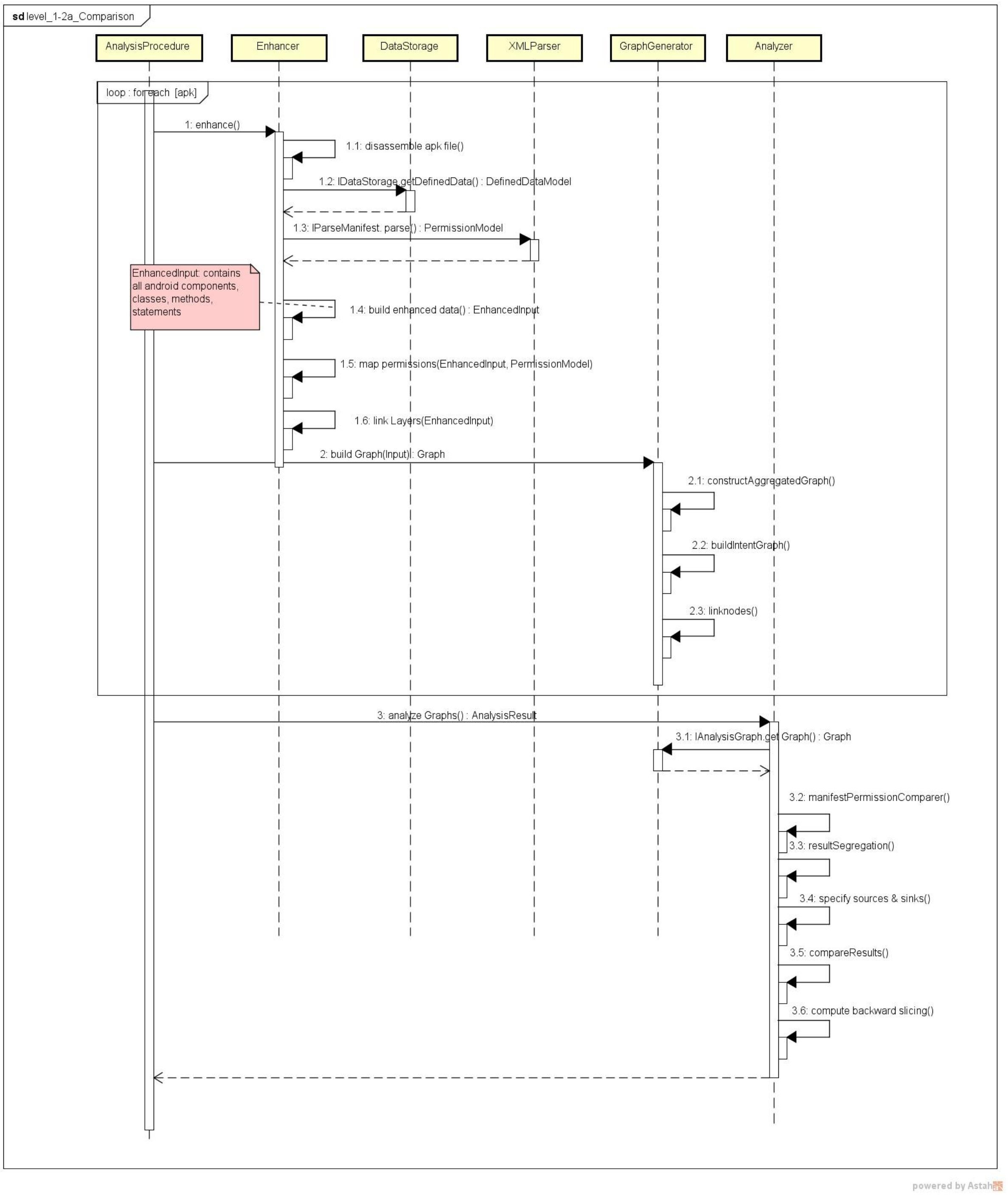

sd level_1-2a_Comparison
AnalysisProcedure
Enhancer
DataStorage
XMLParser
GraphGenerator
Analyzer
loop : for each [apk]
1: enhance()
1.1: disassemble apk file()
1.2: IDataStorage.getDefinedData() : DefinedDataModel
1.3: IParseManifest. parse() : PermissionModel
EnhancedInput: contains all android components, classes, methods, statements
1.4: build enhanced data() : EnhancedInput
1.5: map permissions(EnhancedInput, PermissionModel)
1.6: link Layers(EnhancedInput)
2: build Graph(Input) : Graph
2.1: constructAggregatedGraph()
2.2: buildIntentGraph()
2.3: linknodes()
3: analyze Graphs() : AnalysisResult
3.1: IAnalysisGraph.get Graph() : Graph
3.2: manifestPermissionComparer()
3.3: resultSegregation()
3.4: specify sources & sinks()
3.5: compareResults()
3.6: compute backward slicing()
powered by Astah

Figure 8: High Level Sequence Diagram for Level 1 and 2a Analysis - Comparison

previous saved analysed result. The brief description of the process taking place specifically during this mode is explained below-

After the `AnalysisRunner` collects all required result of the inputs, using the `procedure` got from `AnalysisFactory` it invokes the comparison function in the interface `IDoComparisonAnalysis`(provided by the `AnalysisProcedure` component) to compare the results. Inside the component `AnalysisProcedure`, the `ResultComparer` takes responsibility for comparing the input result and deriving the comparison result. Then the `AnalysisProcedure` returns the comparison result to `AnalysisRunner` for continuously passing it to the `Client` and visualizing it for User's view.

2.3.2 Level 2b

During Level 2b Analysis, the Android App Analysis tool will take Inter-App data resource usage into account. This is an extension to the functionality of Level 1. In this level our tool additionally computes the resources used by an App via other foreign Apps, meaning Apps that are not system native. Hence, it covers the possibility to use data resources indirectly by calling another App's component. As required our tool will extend the two modes SUMMARY or COMPARISON with a parameter of computation range, APP and ALL. In APP mode the tool will compute the resource usages of an App including the indirect resources usage via system Apps and other non-native Apps in addition in ALL mode to considering all Apps of the input set as starting point.

The analysis of this level is described generally in a high-level sequence diagram Figure 6 and then for each application mode (SUMMARY or COMPARISON), the detail processing is represented accordingly in high-level sequence diagram in Figure 9 and in Figure 10 showing the sequence of actions taking place, how the task is performed, which steps, it needs to undertake to forfeit complete execution. In the next Paragraphs, all the detail of the sequence diagrams with respect to both the modes is explained.

The general sequence diagram in Figure 6 describes all possible interaction of `User` and the Android App Analysis tool as well as the interaction between components inside the App. The `User` interacts with the Android App Analysis tool through `Client` , similar to the previous levels, i.e the interaction between the `User` and `Client` remains consistent through out all levels of analysis. One change here is the input to the `Client`, here input can be set of `.apk` files or previous saved results, no bar on them.

1. **Step1**: At first step, the `User` configures analysis. The `User` selects Level 2b, `SUMMARY` or `COMPARISON` mode on GUI or specifies them by option parameters in Command Line. Despite using the App in GUI or Command Line mode, the `User` still mainly interacts with the `Client` component.
2. **Step2**: Then, the `User` specifies the necessary input (`.apk` file) and previous analysis result (optional) in case of COMPARISON mode through the `Client` components.

Notable change here is that no bar on combination of `.apk` file and previous saved result as Input to `Client`.

3. **Step3**:The `User` then performs analysis with the specified configuration and the provided input by invoking appropriate functions supported by `Client` component. For different modes (SUMMARY or COMPARISON), the analysis is also different. It is described clearly in following sections SUMMARY and COMPARISON . After the analysis is done, the result will be visualized by `Client` component.
4. **Step4**: In case of using GUI mode, the `USER` can filter the result based on defined criteria offered by `Client` component.
5. **Step5**: The `User` can also choose saving the result(for future use) or not as an additional option. If the `User` want to save it, a location should be specified by the `User`.

   The `User` also can load a previous analysis result for reviewing by specifying the file to the `Client` component. The component will parse the file and visualize the result to the `User`.

SUMMARY Mode In this mode, at first the analysis of Level 1 Figure 7 is carried out. After generating the processed result from Level 1 , it is then passed to `GraphGenerator`. The `GraphGenerator` class also has aggregation relation with two other level specific classes `IntentAnalyzer` and `NodeLinker`. The `GraphGenerator` component is called to build a graph for the collection of results which is given as input.The `IntentAnalyzer` class has a constructor which passes `ResultInput` as argument. In Level 2b the `IntentAnalyzer` component is important to analyze `implicit intent`, to track the communication between two or more Apps. The function of the `IntentAnalyzer` varies between different modes. Then, based on that it will compute the least fixed pointc which passes and returns an `AnalysisGraph`. Finally, it will generate the final analysis result of the Summary mode of this level to the `AnalysisProcedure` component, which in turn will pass it to the `AnalysisRunner` component. The user can then add filters and have an option in the end to save the current result to be used in the later prospect , this process is very similar to the levels of analysis stated above (1 and 2a).

COMPARISON Mode As described in Figure 10 above , this mode is used to compare analyzed result from a set of `.apk` files with a previous saved result .In this mode, after the `User` sets up the analysis in **Step1**, and in the next step, the `User` needs to give a set consisting of `.apk` file or one previous analysis result file as input to the `Client`. In addition, the `User` can have some further comparisons between the analysis results, as we have extended our roots further in Level 2b Summary mode, which is an extension of Level 1 analysis. So Summary mode of Level 2b is carried out for each `.apk` file or the previous result is loaded from the `ResultLoader` and then the comparison is carried out. The brief description of the process taking place specifically during this mode is explained below.

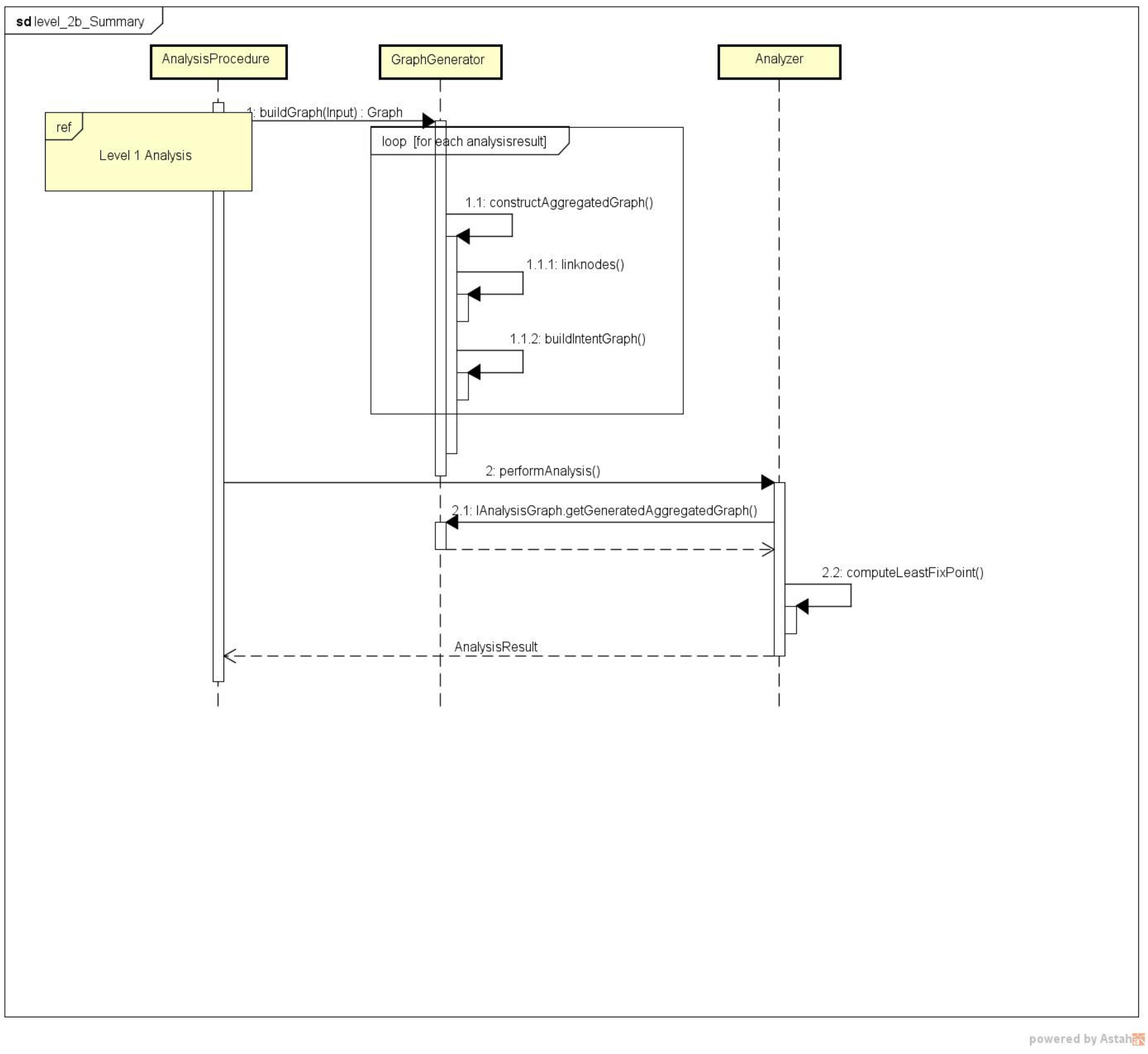

sd level_2b_Summary
AnalysisProcedure
GraphGenerator
Analyzer
ref
Level 1 Analysis
1: buildGraph(Input) : Graph
loop [for each analysisresult]
1.1: constructAggregatedGraph()
1.1.1: linknodes()
1.1.2: buildIntentGraph()
2: performAnalysis()
2.1: IAnalysisGraph.getGeneratedAggregatedGraph()
2.2: computeLeastFixPoint()
AnalysisResult
powered by Astah

Figure 9: High Level Sequence Diagram for Level 2b Analysis - Summary

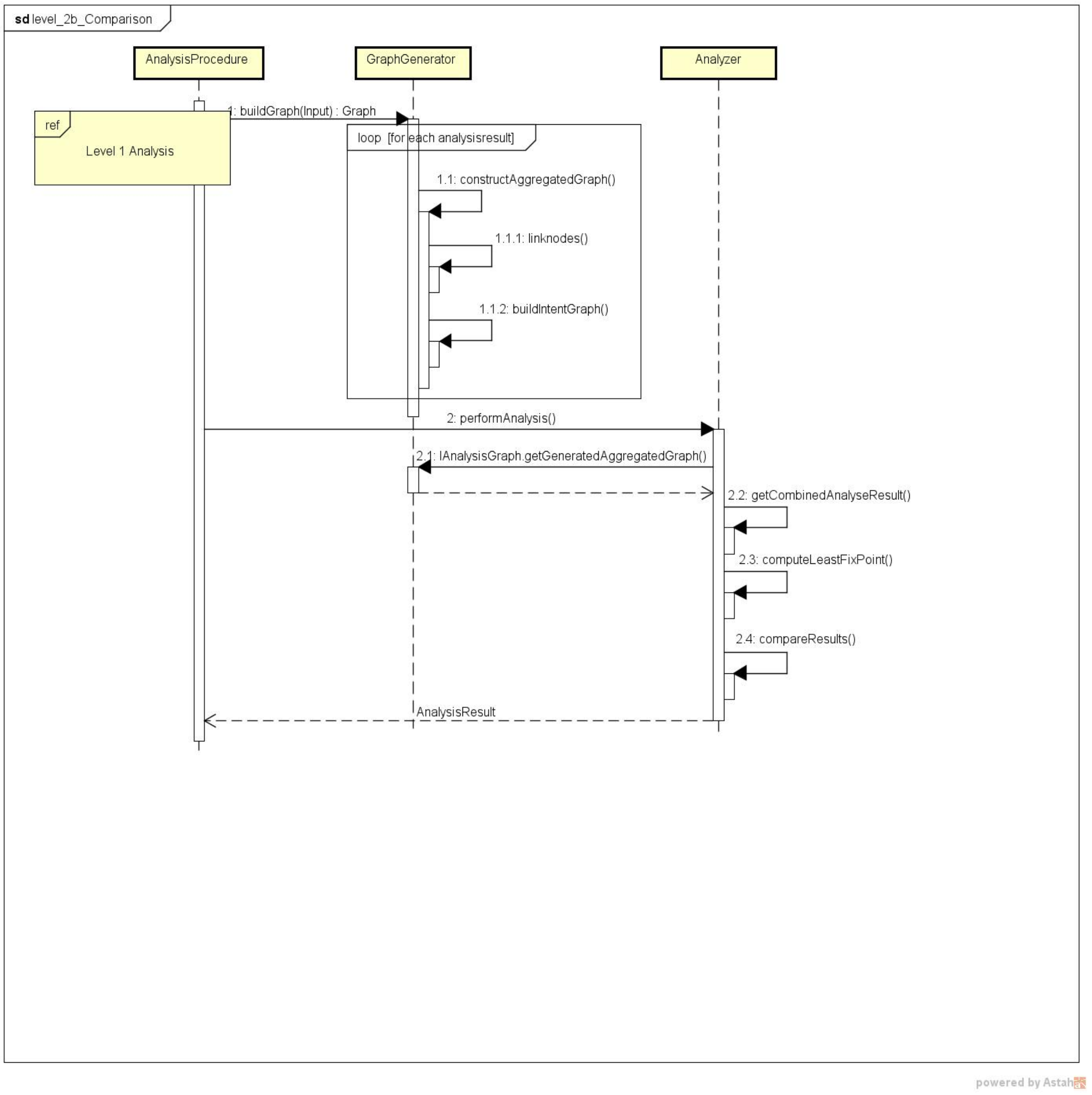

sd level_2b_Comparison
AnalysisProcedure
GraphGenerator
Analyzer
ref
Level 1 Analysis
1: buildGraph(Input) : Graph
loop [for each analysisresult]
1.1: constructAggregatedGraph()
1.1.1: linknodes()
1.1.2: buildIntentGraph()
2: performAnalysis()
2.1: IAnalysisGraph.getGeneratedAggregatedGraph()
2.2: getCombinedAnalyseResult()
2.3: computeLeastFixPoint()
2.4: compareResults()
AnalysisResult
powered by Astah

Figure 10: High Level Sequence Diagram for Level 2b Analysis - Comparison

1. Firstly, similar as in `SUMMARY` mode, the `Client` also uses the `ICreateAnalysis-Procedure` interface (provided by the `AnalysisFactory`) to get an appropriate `Analysis` Object for `COMPARISON` mode. Beside this Object, the `AnalysisFac-tory` also generates other Objects to analyze the input files if the first or the second input is `.apk` file.

2. Secondly, in the case that the input file is `.apk`, the `Client` component will analyze it in `SUMMARY` mode to get the analysis result (see the section `SUMMARY` mode for more detail). If the input is a previous analysis result, the `Client` uses the `ResultLoader` to load the result in files. This step makes sure that before comparison analysis step, all the inputs are already analyzed to get the result.

3. After the `AnalysisRunner` collects all required result of the inputs, using the `Anal-ysis` Object got from `AnalysisFactory` it invokes the comparison function in the interface `IDoAnalysis`(provided by the `AnalysisProcedure` component) to compare the results.

4. Then the `Analyzer` invokes `computeFixedPoint` and `getLeastFixedPoint-FromPreviousResult` to make it available for the `ResultComparer`

   Inside the component `AnalysisProcedure`, the `ResultComparer` takes responsibility for comparing the input result and deriving the comparison result. Then the `AnalysisProcedure` returns the comparison result to the `AnalysisProcedure` for continuously passing it to the `Client` and visualizing it for the user's view.

3 Soot Framework

Soot is a static analysis framework developed by Sable Research Group from McGill University [1]. It was initially developed for analysing, transforming and optimizing Java bytecode. Besides Java, it also supports other input languages such as SML, Eiffel and Scheme. The Soot users then, based on the intermediate representations (Jimple, Shimple, Grimp, Baf and Dava) at appropriate abstraction level, can extend or add new analyses and optimizations on bytecode. Soot constructs call graphs for a whole-program analysis or in particular for a inter-procedural analysis. Soot also provides a variety of intra-procedural analyses. We will be using a nightly-build of Soot for our project. This can be obtained from their Website[2].

The sections below will briefly describe some main features used in components of the Component diagram in Figure 5 and how Soot will be used in the project.

[1]https://github.com/Sable/soot

[2]https://ssebuild.cased.de/nightly/soot/

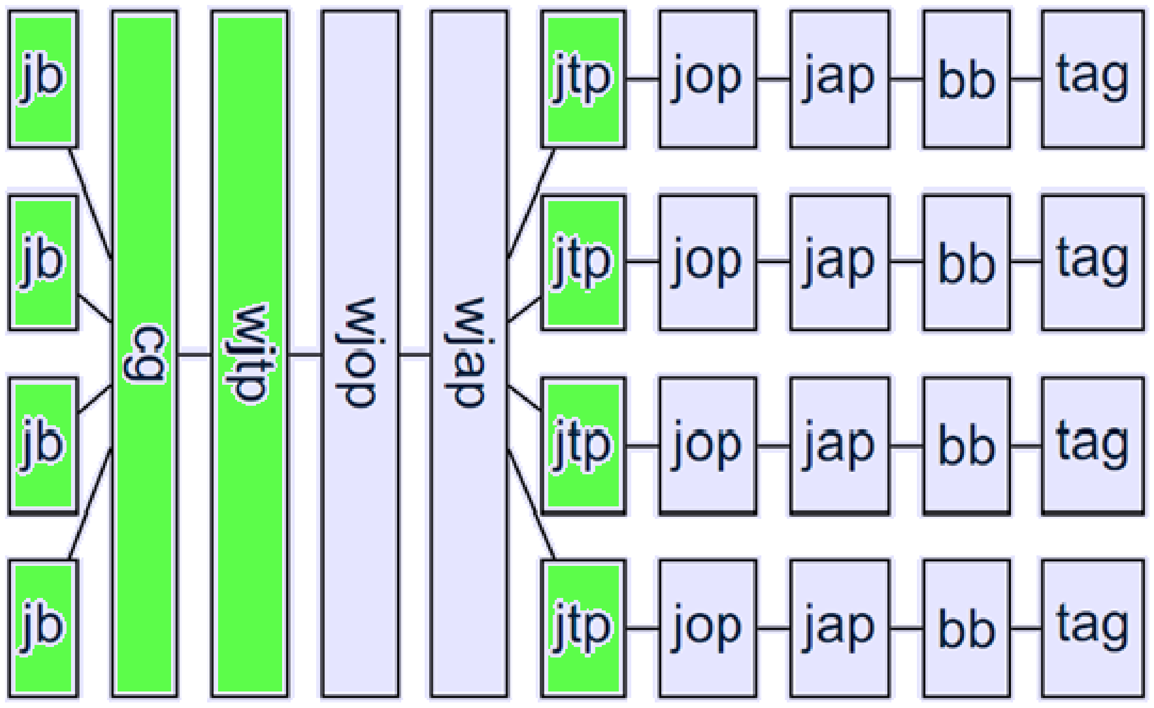


Figure 11: Phases in Soot; Figure taken from: https://github.com/Sable/soot/wiki/Packs-and-phases-in-Soot

3.1 Soot's Features

Soot's execution is divided into a number of phases. Each phase in turn comprises of several sub-phases. The behaviour of a phase can be modified using associated phase options. Within Soot, each phase is implemented using a `Pack`. Each `Pack` consists of several `Transformers` corresponding to sub-phases of the phase that is being implemented. The main Soot phases used in the project are `jb` (Jimple body creation), `cg` (Call graph generation), `wjtp` (Whole program transformation) and `jtp` (intra-procedural phase). To work on these packs of phases, Soot provides options to create and add new sub-phases to specific packs. Figure 11 provides an idea overview of the various packs implemented in Soot and their sequence of execution.

Jimple is a typed three-address intermediate representation for bytecode. The first phase of Soot (the Jimple body creation `jb`) is used to convert the input files into Jimple representation. For Android applications, Soot provides a plug-in called `Dexpler` that support Soot users to work on `Android Dalvik bytecode` besides the Java bytecode. Soot takes the `.apk` file as input and then disassembles it to `Jimple` intermediate representations. In Jimple, each class is represented as a `SootClass`, each `SootClass` contains a collections of `SootFields` and `SootMethods`. Each `SootMethod` consists of a set of `Stmt` statements that form a body for the method. The intermediate representation then is used in many phases later on of Soot's execution for analyses.[3]

Call graph supported in Soot is mainly used for inter-procedure analyses. It provides information about the `call site` (statement or method from where a call to another method is made) and all possible targets of that `call site`. It is a collection of edges representing all known method invocations. Each edge contains four elements: source method, source statement, target method and the kind of edge. The Call Graph is initialized for the whole program in the pack `cg` and in this pack different sub-phases construct call graphs using different algorithms. Soot provides four algorithms for constructing a call graph, namely RTA (Rapid Type Analysis), CHA (Class Hierarchy Analysis), VTA (Variable-Type Analysis) and SPARK (Soot Pointer Analysis Research Kit). Each algorithm is implemented by a pack of the same name. In the project Android Application Analysis, CHA and SPARK packs will be mainly used. In particular, the simplest call graph is obtained by CHA algorithm which assumes that all reference variables can point to any object of the correct type. SPARK generates the call graph and in addition, provides a point assignment graph (PAG) to support points-to analysis required for inter-procedure analysis. Soot users can specify a concrete algorithm for constructing a call graph and by default Soot always enables the phase `cg` to construct a call graph for the next phase (the Whole Jimple Transformation Pack `wjtp`) where the inter-procedural analysis will be performed. One problem associated with generating call graphs for Android applications is that Android applications do not have a `main()` method. But Soot requires a `main()` method and a list of `entry points` for the call graph. To solve this, we will be creating a dummy `main()` method and also identify a list of entry points and provide it to Soot for call graph construction. How this will be realised, will be explained in the Implementation Details Section 3.2

Inter-procedural analysis is a static analysis that requires a call-graph and pointer information. The analysis will be defined in the Whole Jimple Transformation Pack or the `wjtp` phase. The most important thing is that the call graph can only be obtained in the whole-program mode which is specified by the option `-w` to Soot. In this `wjtp` phase, Soot's users create a new transformer which mainly works on the call graph and other analysis (such as point-to analysis, side effect analysis etc.). This new transformer later on is added to the current pack of the `wjtp` phase.

Intra-procedural analysis is the key feature of Soot. The phase `jtp` is the place for user-defined intra-procedural analysis. Here, Soot users can create a data-flow analysis by specifying the abstraction and implementing a transfer function. The analysis works mainly on the control flow graph - built by `UnitGraph` where each node is a statement or an expression and each

[3]Patrice Pominville, Feng Qian, Raja Vallee-Rai, Laurie Hendren, Clark Verbrugge *A Framework for Optimizing Java Using Attributes*

edge is the control flow path between two nodes if available. The data-flow analysis associates two flow sets with each node in the unit graph, usually one in-set and one out-set. The traced information or data would be inspected in the flow set before and after each statement. Soot also supports an abstract class `FlowAnalysis` to specify the flow of analysis (forward or backward). Again, Soot users need to create a new transformer extended from Soot and add them to the current pack `jtp`.

3.2 Implementation Details

For using features of Soot, a number of options should be specified. Soot options can be broadly classified into several option groups like input options, output options, general options, phase options etc. A detailed list of the various Soot options and possible values that can be passed, can be found at the developers' website [4]. In addition, some extended implementation for analysis also need to be taken into account. Depending on the type of analysis, inter- or intra-, the implementation would be placed in the appropriate phases which are already mentioned in the previous section.

3.2.1 Running Soot

In the Android Application Analysis project, the options below must be specified for running Soot.

General options are general configurations for running Soot.

`-w` for running Soot in whole-program mode.

Input options are specifications for the input before running Soot.

`-pp` prepends the given Soot classpath to the default classpath.

`-soot-class-path` *`/android-platforms-master`* specify the path to the platform JAR files which are Android standard libraries that Soot requires for resolving types of analysed Apps. Here the `android-platforms-master` is the directory that contains all those JAR files. Another way for specifying the Android library is using the option `-android-jars` *`/android-platforms-master`*.

`-src-prec` *`apk`* sets source precedence to *apk* file because the input file for this project will alway be `.apk` file.

`-allow-phantom-refs` allows unresolved classes. This option is mainly used for classes in external library or in system library because those classes are not the targets for analysing in the project.

[4]https://ssebuild.cased.de/nightly/soot/doc/soot_options

`-no-bodies-for-excluded` to not load any method bodies of classes specified in the "exclude" package.

`-procecc-path` *`dir`* process the input `.apk` file found in this `dir`.

Output options because for this project, no output is required. The following option is specified

`-output-format` *`none`* sets Soot not to produce any output file.

Processing options specifies setting for Soot' process.

`-phase-option cg` *`cg.spark:on`* specify the SPARK algorithm for constructing the call graph in phase `cg`. Besides SPARK, there are other algorithms such as Rapid Type Analysis `RTA` (`cg.rta`), Variable Type Analysis `VTA` (`cg.vta`) and Class Hierarchy Analysis `CHA` (the default call graph constructor).

Application mode options specify the package or set up for the classes.

`-exclude` *`pkg`* excludes the package `pkg` that contains the Android libraries such as `android.*`. This option might be specified or not based on the purpose of the analysis.

After all required options are specified, Soot is executed by calling *soot.Main* available in the library *soot-trunk.jar*[5]. The various options that will be used to configure Soot are passed as a string argument to the `soot.Main` program. This will execute all the packs in the Soot in sequence. Because Soot's execution is divided into many phases, Soot also provides the option to run particular phases only. For example the method `PackManager.v().getPack("cg").apply();` is called to perform execution in this phase `cg` only. Even Soot users can call all packs executing at once by using `PackManager.v().runPacks()`. This feature allows Soot users to flexibly run their own analyses in specific phases.

3.2.2 Creating Entry Point

We have to create a dummy main method for the Android application, so that it can be used in Soot for call graph generation. Soot provides the option to create a Class from scratch and add methods and statements to the created class. We obtain the list of component classes in the Android application from the `Manifest` file. We have to compute the entry points for each component based on the Android life cycle of that component. In general, these entry points will be some of the methods declared in the component. With Soot, we will create a dummy main class using Soot's `SootClass` object. We will then create a dummy main method using

[5]'trunk' represents nightly build of Soot

SootMethod object and add this method to the dummy main class. Within the body of this dummy main method, we will add instances of the component classes and other classes used in the application. We then add Invoke Statements that use the class instances to make calls to the identified entry point methods. This dummy main method will then be set as the entry point for our analysis to generate call graph.

3.2.3 Adding own analysis to Soot

Soot provides the approach to add user-defined sub-phases without modifying Soot's structure. This is done by creating a new class that extends BodyTransformer or SceneTransformer. Both provide a way for performing analysis or make transformation on a single method body (intra-procedural) and on a whole application (inter-procedural). All extended classes from both of these transformers should override the internalTransform(...) method for different analysis purposes.

Inter-procedural analysis The SceneTransformer is used in phase wjtp. It executes once and may analyze and manipulate the entire program. Therefore for the inter-procedural analysis, Soot users create a new transformer extended from SceneTransformer and add it to the wjtp pack. When overriding the method internalTransform(...), Soot users can get a call graph which is computed in the previous phase cg. The call graph can be accessed through the environment class Scene with the method getCallGraph(). Then Soot users can query for the edges coming into a method or going out of a method or for edges going out of a particular statement (by using those methods edgesIto(method), edgesOutof(method) and edgesOutOf(statement)).

In the implementation of LayerLinker, the class LinkTransformer will extend SceneTransformer of Soot and override the internalTransform(...) method. Then using Soot's methods, the list of classes, the corresponding methods and statements can be obtained which will be the input for the addStatement(...),addMethod(...) and addClass(...) methods of the LayerLinker. For example the Soot utility getApplicationClasses() will return a list of application classes, which will then be used as input for the addClass(...) method of the LayerLinker.

Another instance, where we want to use Inter-procedural analysis in our tool, will be in the implementation of NodeLinkerLvl2a. This class will use NodeLinkTrasnformerPart1, an extension of SceneTransformer. In the overwritten internalTransform(...) method, Soot method getCallGraph() will be used to obtain the call graph for the Android application under analysis. The getGraphsFromSoot(...) method will then use the call graph from the previous step and assign it to callGraph variable in our tool.

Intra-procedural analysis The BodyTransformer is most suitable for intra-procedural analysis. This analysis is to be added to the phase jtp which means the transformer is executed on each method in a program. In particular, Soot users perform an analysis on the control flow graph supported in the three basic unit graphs - `BriefUnitGraph, ExceptionalUnitGraph` and `TrapUnitGraph`. In addition, Soot users also need to declare a class extended from `FlowAnalysis` to specify the flow of analysis. Soot provides three classes that extend the `AbstractFlowAnalysis` class. They are `ForwardFlowAnalysis, BackwardFlowAnalysis` and `BranchedFlowAnalysis`. Depending on the type of analysis, a Soot user can choose appropriate abstract class. For example, for Available Expression Analysis, the abstract class must be `ForwardFlowAnalysis` or for Live Variable Analysis, the abstract class must be `BackwardFlowAnalysis`.

In our tool, we will be using Intra-procedural analysis during the implementation of `NodeLinkerLvl2a`. `NodeLinkerLvl2a` will use the class `NodeLinkTransformerPart2` which will be an extension of Soot's BodyTransformer. Within the overwritten `internalTransform(...)` method, for each method in the Android application, an object of `ExceptionalUnitGraph` will be created. The `ExceptionUnitGraph` objects for all the methods will be added to a list. `getGraphsFromSoot(...)` method of `NodeLinkerLvl2a` will assign this list to the variable `methodFlows`.

4 Refined Architecture

This section describes the detailed class structure of our tool. Therefore, we start by describing the overall framework in Sec. 4.1. This structure then is refined for the different levels with level-specific details in Section 4.2, 4.3 and 4.4. Finally, Section 4.5 describes the detailed structure of the user interface as well as the client.

Note that whenever a method that uses parameters is mentioned in the following subsections the parameters are not mentioned in the brackets but visualized by dots. This is done since there will be no two methods with same name that differ only in the parameters. But all parameters can be found when looking up the method in the corresponding diagrams.

The class and sequence diagrams in this section will use four different colors for classes to distinguish between different roles of the classes. Green displays that the class is a soot class, whereas blue describes an interface or abstract class. The red classes are mainly used in the subsections related to the three levels (see Sec. 4.2, Sec. 4.3 and Sec. 4.4). All remaining classes are displayed in light yellow.

One more remark has to be mentioned concerning the `Logging` component (see Fig. 5). This component is not mentioned explicitly in the now following refinement to ensure easily comprehensible diagrams. Nevertheless, in our tool logging will be realized using a logging framework which will be accessed by all of the classes to log their current status while doing their tasks.

4.1 Overall Framework

In the following the overall framework of our tool will be described. This part of the tool contains the core analysis logic and serves a a basis for our 3 analysis levels. Moreover, the framework provides core services to be used by the analyses as well as the Soot framework.

4.1.1 Analysis

The class diagram of the tool framework is given in Figure 12. The framework will provide the interface `AnalysisProcedure` for performing arbitrary analyses on Android Apps. There are two kinds of analyses: An initial analysis that analyzes an Android App from scratch should be performed when the method `doInitialAnalysis(...)` is called. The second type of analysis is triggered with the method `doAggregationAnalysis(...)` and should aggregate a list of intermediate `AnalysisResults` into a new result.

For our project group we will use the class `A3AnalysisProcedure` to perform our three analysis levels. Our analyses consist of three steps as described previously in Section 2.2. The `Enhancer` class is the refinement of the `Enhancer` component. This class will be the same for all our analysis levels and, hence, can be defined directly. The concrete structure and behavior of the `Enhancer` class will be described in Section 4.1.4. The other two steps, the `GraphGenerator` component and afterward the `Analyzer` component, are level dependent. For the tool framework they are substituted by interfaces of the same name. Both interfaces provide a single method to process the information during an analysis. The concrete implementations will be passed to the `A3AnalysisProcedure` in the constructor during analysis configuration.

Every `AnalysisProcedure` will be run by an instance of the `Analysis` class. It manages the required inputs, namely the `.apk` file, the previous result `prevRes` for `COMPARISON` mode and the list of intermediate results `subResults`, which will be passed to the `AnalysisProcedure` during analysis. The type of the analysis (initial or aggregation) will be determined by the parameter `apkFile` in the constructor. If `apkFile` is `null` the type will be an aggregation analysis and an initial analysis in the other case.

The class `AnalysisRunner` provides a single interface for the `Client` to run a complete analysis which is in fact a list of `Analysis` objects that build up on each other. Such a structure is need for Level 2b which builds upon multiple Level 1 results. For Level 1 and Level 2a the provided list will only contain a single `Analysis` object since both analyses analyze only a single Android App.

To support the `Client` in configuring complex analyses, the tool framework provides the `AnalysisFactory` interface. Each analysis will have to extend this interface such that the `Client` can instantiate a factory with a set of configuration parameters and then call `createAnalysis()` to retrieve the preconfigured list of `Analysis` objects.

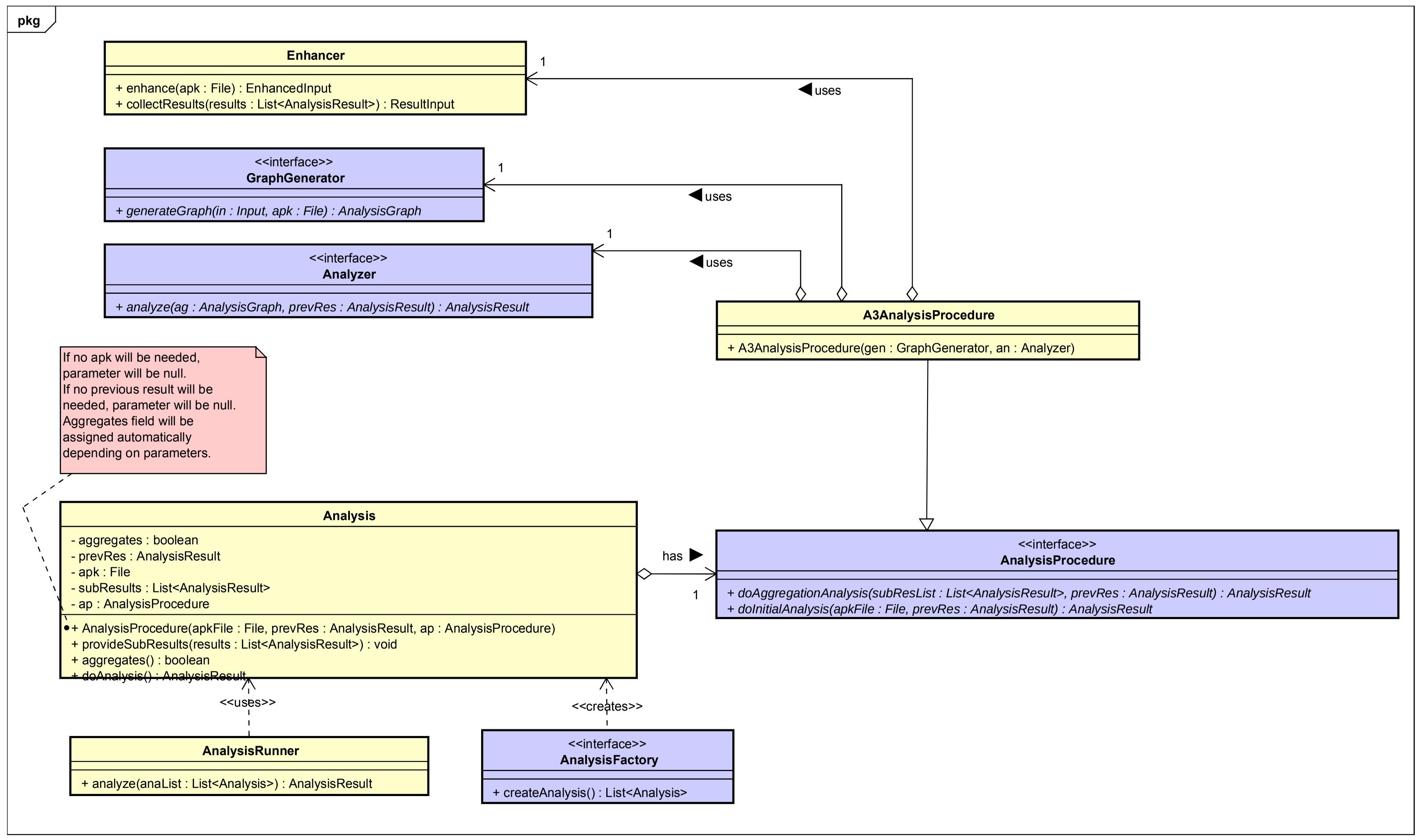

pkg
Enhancer
+ enhance(apk : File) : EnhancedInput
+ collectResults(results : List<AnalysisResult>) : ResultInput
<<interface>>
GraphGenerator
+ generateGraph(in : Input, apk : File) : AnalysisGraph
<<interface>>
Analyzer
+ analyze(ag : AnalysisGraph, prevRes : AnalysisResult) : AnalysisResult
A3AnalysisProcedure
+ A3AnalysisProcedure(gen : GraphGenerator, an : Analyzer)
uses
uses
uses
1
1
1
If no apk will be needed, parameter will be null. If no previous result will be needed, parameter will be null. Aggregates field will be assigned automatically depending on parameters.
Analysis
- aggregates : boolean
- prevRes : AnalysisResult
- apk : File
- subResults : List<AnalysisResult>
- ap : AnalysisProcedure
+ AnalysisProcedure(apkFile : File, prevRes : AnalysisResult, ap : AnalysisProcedure)
+ provideSubResults(results : List<AnalysisResult>) : void
+ aggregates() : boolean
+ doAnalysis() : AnalysisResult
has
1
<<interface>>
AnalysisProcedure
+ doAggregationAnalysis(subResList : List<AnalysisResult>, prevRes : AnalysisResult) : AnalysisResult
+ doInitialAnalysis(apkFile : File, prevRes : AnalysisResult) : AnalysisResult
<<uses>>
<<creates>>
AnalysisRunner
+ analyze(anaList : List<Analysis>) : AnalysisResult
<<interface>>
AnalysisFactory
+ createAnalysis() : List<Analysis>

Figure 12: Class Diagram of the Tool Framework

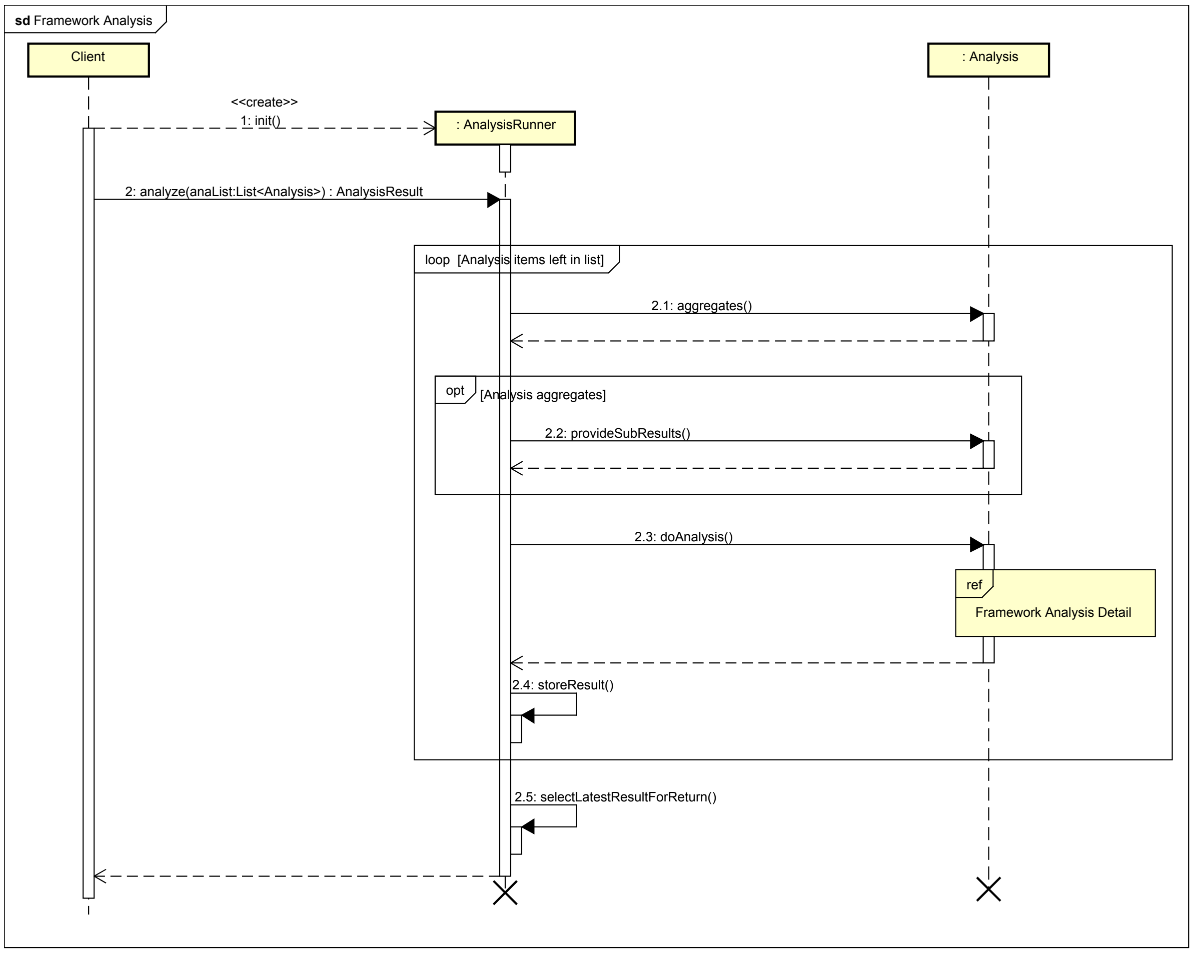

sd Framework Analysis
Client
: Analysis
<<create>>
1: init()
: AnalysisRunner
2: analyze(anaList:List<Analysis>) : AnalysisResult
loop [Analysis items left in list]
2.1: aggregates()
opt [Analysis aggregates]
2.2: provideSubResults()
2.3: doAnalysis()
ref
Framework Analysis Detail
2.4: storeResult()
2.5: selectLatestResultForReturn()

Figure 13: Sequence Diagram of the Analysis Framework

After describing the structure of the tool framework, the behavior during an analysis triggered by the `Client` will be explained. The according sequence diagram is shown in Figure 13 and Figure 14. The `Client` triggers an analysis by instantiating the `AnalysisRunner` and calling its `analyze(...)` method. The `AnalysisRunner` will process each of the `Analysis` objects in the given list.

For each `Analysis` the `AnalysisRunner` requests the type of the `Analysis` by calling `aggregates()`. If the type is aggregation, the `AnalysisRunner` provides the previously collected results to the `Analysis` via `provideSubResults(...)`. Finally, it calls the `doAnalysis()` method to start the analysis.

Depending on the analysis type (aggregation or initial) the `Analysis` will call `doAggregtionAnalysis(...)` or respectively `doInitalAnalysis(...)` on our class `A3AnalysisProcedure`. The method `doAggregtionAnalysis(...)` will call `collectResults(...)` of the `Enhancer` to get an initial data representation to do

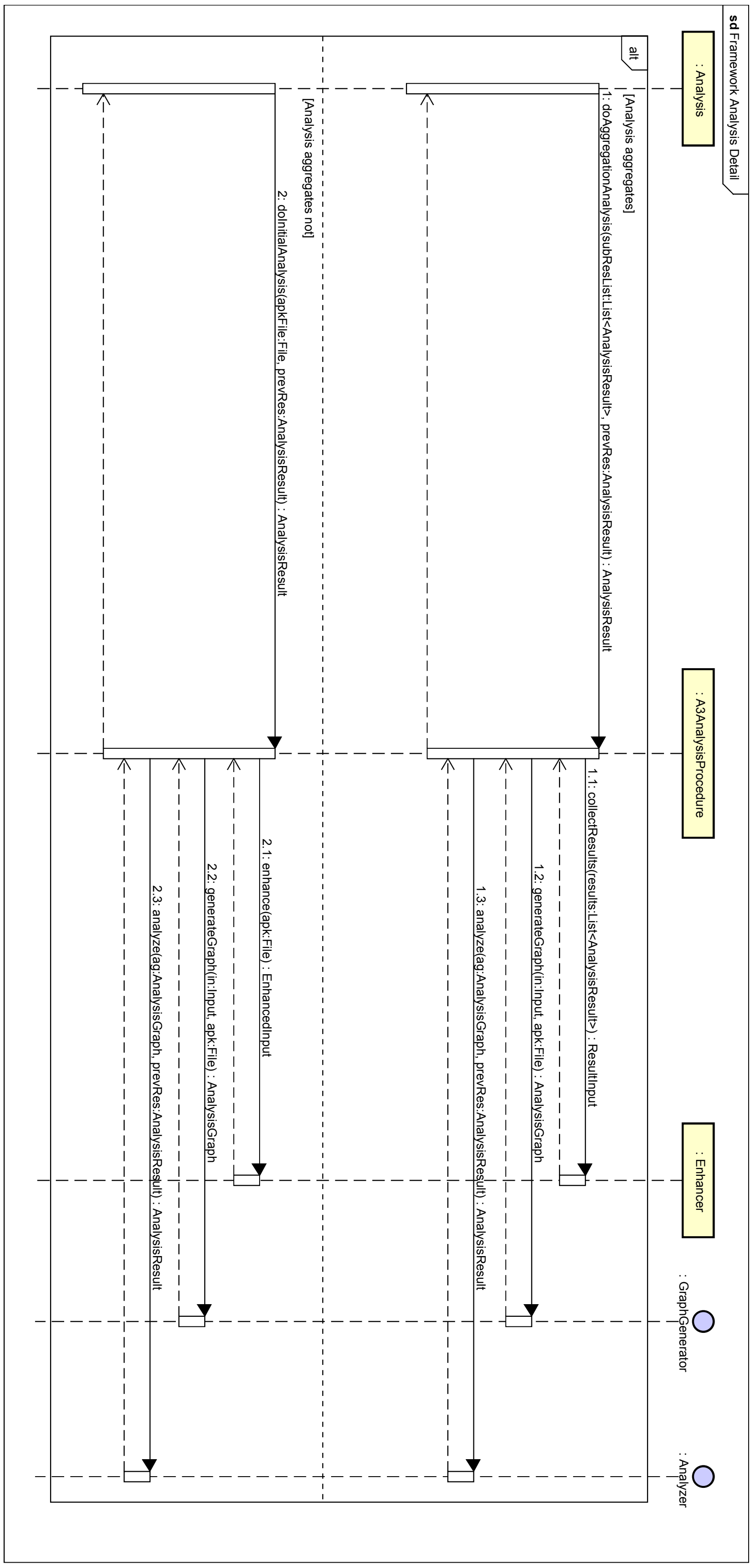

sd Framework Analysis Detail
: Analysis
: A3AnalysisProcedure
: Enhancer
: GraphGenerator
: Analyzer
alt
[Analysis aggregates]
1: doAggregationAnalysis(subResList:List<AnalysisResult>, prevRes:AnalysisResult) : AnalysisResult
1.1: collectResults(results:List<AnalysisResult>) : ResultInput
1.2: generateGraph(in:Input, apk:File) : AnalysisGraph
1.3: analyze(ag:AnalysisGraph, prevRes:AnalysisResult) : AnalysisResult
[Analysis aggregates not]
2: doInitialAnalysis(apkFile:File, prevRes:AnalysisResult) : AnalysisResult
2.1: enhance(apk:File) : EnhancedInput
2.2: generateGraph(in:Input, apk:File) : AnalysisGraph
2.3: analyze(ag:AnalysisGraph, prevRes:AnalysisResult) : AnalysisResult

Figure 14: Detail Sequence Diagram of the Analysis Framework

the further analysis with. On the other hand, `doInitialAnalysis(...)` triggers `enhance(...)` of the `Enhancer` to reach the same goal. After that the `AnalysisProcedure` calls `generateGraph(...)` of the `GraphGenerator` to continue the processing for both analysis types. The final step is to call `analyze(...)` on the `Analyzer` to retrieve the final result. This will again be done for both analysis types, aggregation and initial.

After the `AnalysisRunner` received the result, it stores the result before continuing the processing of the `Analysis` objects in the list. When all `Analysis` objects are processed, the `AnalysisRunner` will select the result stored most recently to return to the `Client`.

4.1.2 Data Structures

In this section we will describe the data structures that are used to store information during and after an analysis. The class diagram of those data structures is given in Figure 15. First, we have the interface `Input` that is used to combine the two classes `ResultInput` and `EnhancedInput`. The `ResultInput` is just a wrapper around a list of `AnalysisResults` which is needed for aggregation analyses like Level 2b. The `EnhancedInput` is more interesting in the sense that it represents the complete source code of an Android App. An `EnhancedInput` therefore contains a single instance of the class `App`. The structure for the `App` is hierarchical and organized as follows:

- An `App` contains at least one `Class`. Some of those classes may be one of the four Android `Components`. Each `App` can also be identified by its `fingerprint` which is a String
- A `Class` contains at least a `Method` or a class variable which is modeled by the class `Statement`. A `Class` may also have some subclasses of the same type which will have access to all methods and fields of the surrounding `Class`. To identify a `Class` the method `getSignature()`, that returns a String combined from package and class name, can be called. The specialization `Component` additionally contains a `type` which will be one of the four Android Component types. In case the component is a content provider it may also contain a list of of URIs for which temporary permissions are specified in the Android Manifest for this content provider. Otherwise the list will be empty.
- A `Method` contains at least a single `Statement`. It has a `returnType` and a list of parameters stored as Strings. `Methods` can also be identified via the `getSignature()` method which here returns a String build up from the surrounding `Class` signature, the method name and the parameter types.
- A `Statement` is the lowest item in the hierarchy and represents an assignment or a method call.

All classes described in the list above extend the abstract class `Element`. This class provides the ability to add a set of Android usage `Permissions` to each `Element` and also the

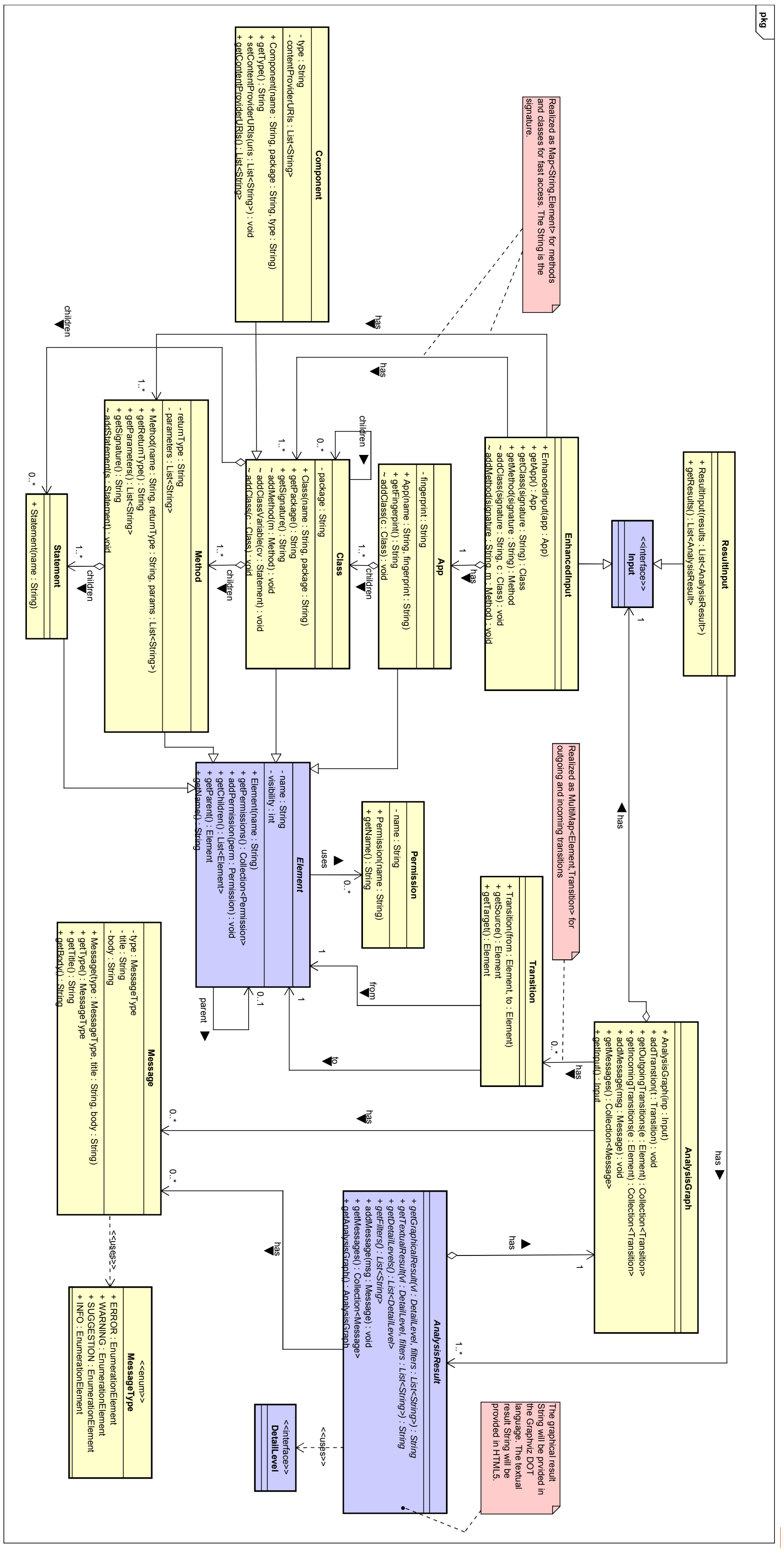

pkg
Component
App
Class
Method
Statement
Element
Permission
EnhancedInput
<<interface>> Input
ResultInput
Transition
AnalysisGraph
AnalysisResult
Message
<<enum>> MessageType
<<interface>> DetailLevel
Realized as Map<String,Element> for methods and classes for fast access. The String is the signature.
Realized as MultiMap<Element,Transition> for outgoing and incoming transitions
The graphical result String will be prvided in the Graphviz DOT language. The textual result String will be provided in HTML5.

Figure 15: Class Diagram of the Tool Data Structures

methods `getChilden()` and `getParent()` for moving up and down in the hierarchy. We will not go into more detail here, since the rest of this data structure consists of self-explanatory setter and getter methods. The `EnhancedInput` additionally contains two maps for a quick access of `Classes` and `Methods` via their signature which may speed up the processing in the `GraphGenerator`.

The `GraphGenerator` will extend the `Input` with additional directed edges. Therefore, the `Input` will be wrapped into an `AnalysisGraph`. Edges are represented by the class `Transition` which has a source and a target `Element`. Hence, `Transitions` can connect arbitrary parts of source code depending on the analysis that should be performed on that `AnalysisGraph`. Inside the `AnalysisGraph` the `Transitions` are stored inside two multimaps[6] for a fast access of incoming and outgoing `Transitions` for one `Element`.

The last step of the analysis chain will produce an instance of a specialization of the abstract class `AnalysisResult` which wraps the `AnalysisGraph`. This specialization contains the result information depending on the performed analysis and will be forwarded to the `Client`.

For gaining the ability of the analysis framework to be used by arbitrary clients, we have to specify some standards for how the `AnalysisResult` provides its information to the `Client`. For the textual result representation we want to provide preformated text, e.g. font color for the permission groups in Level 1 and 2b, and therefore we will use the **HTML5**[7] language. The same requirements have to hold for the graphical result representation. Here we chose the Graphviz **DOT**[8] language. This language describes arbitrary graphs and, hence, on the downside we have to stick to graphs for graphical representations. But this should be fine since a graph can model arbitrary relations between objects and using subgraphs we will even be able to model hierarchies. Both languages are widely used for the purposes they are made for and, hence, there exist multiple programs and tools to also process the result representations externally when saved to a file as String. For us, this is also an indicator that we are well-advised to use this languages for providing the result representations to the user.

The `AnalysisResult` also provides detail levels and filters. Those are again analysis dependent and will be provided to the `Client` through the methods `getDetailLevels()` and `getFilters()`. The `Client` can now select a `DetailLevel` and a subset of the filter Strings and request a textual or graphical result representation from the `AnalysisResult` by calling the methods `getTextualResult(...)` or `getGraphicalResult(...)`.

Moreover, the `AnalysisGraph` and the `AnalysisResult` may contain a set of `Messages`. Those `Messages` can be of different types (`MessageType`), have a title and a message body and will contain user information about special properties of the analyzed App or Apps that were discovered during graph generation and result computation. The `Client`

[6]A multimap can store multiple values under a key.

[7]For specification see: `http://www.w3.org/TR/html5/`

[8]For specification see: `http://www.graphviz.org/content/dot-language`

can access the result messages through the method `getMessages()` to provide them to the user in an appropriate form.

4.1.3 Core Services

As described in the high level structure diagram in Section 2.2 our tool framework will contain several core services that provide their functionality to the analyses and to the `Client`. In the future additional core services may be added to extend the framework features for other kinds of analyses. To be able to offer their functionality globally via a single instance, we require all core services to implement the singleton pattern. Note that this will also hold for future core services to be added to the tool framework for terms of consistency. A description of the singleton pattern can be found later on in Section 4.1.6. The two core services we provide are the `XMLParser` and the `DataStorage` which are described in the following paragraphs.

XMLParser As our tool accepts input as `.apk` file, we first need to unzip it, after unzipping we get the XML files but all these files are not in human readable form. They need to be parsed. They are present in `BXML` (Binary Extensible Markup Language), which is most popularly used format in parsing XML documents. The best feature of `BXML` of its wide use is Random Access and iterations. So for our analysis, we need to parse the `BXML` files in to standard XML format, for that we have developed `A3XML Parser` which parses several `BXML` format files present in to standard XML formats.

`A3XML Parser` uses `XML Pull Parser` to reverse the parsing scenario. `XML Pull Parser` is an interface which provides an API to transform XML files into human readable standard format. The output of this parser will then be used to extract permissions and other string constants from the XML files.

`A3XML Parser` is used at two scenes, one in the beginning and other time during the process. It basically gives us six information in separate method calls, they are described below one by one .

1. `getRequiredPermissions()`: This method will return a map containing components and the permissions associated with it from `Manifest` file.This method will be called at `Enhancer` component by Analyzer together with other methods except `getAppName()` and `getManifestInformation()`.
2. `getUsesPermission()`: This method will return a list containing all the permissions defined in the manifest file under <uses-permission> tag.
3. `getIntentFilters()`: There are some intent filters associated with activities, this function will return the mapping of the permissions with specific intent filters in activities from manifest file.

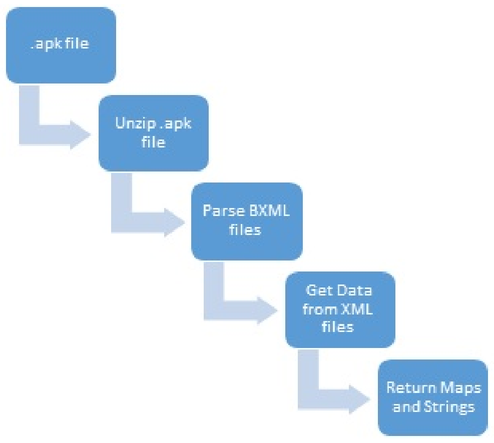


Figure 16: `A3XMLParser` Workflow

4. `getFingerprint()`: This function is be used in the very beginning of the analysis in COMPARISON mode. After the User finishes giving an input `.apk` file to the `Client`, this function will be call to check authenticity of the file. This function will return the *fingerprints*, *version name* and *version code*. These information will then be passed into `AppIdentifier` associate with the App name (got from the function `getAppName()`) to compare the `.apk` file and the previous analysis result. The *fingerprints* details are present in *META-INF* folder, while *version name* and *version code* are in manifest file.

5. `getAppName()`:This function will return the application name mentioned in the XML file, which we can say the actual name of the application as string. This will be derived from `strings.xml` file.

6. `getContentProviderURIs()`: This function will return a mapping between content providers and the respected `URIs`, whenever temporary permissions are assigned in the application.

Apart from these, `A3XML Parser` will unzip the `.apk` file into an accessible folder, which will then be carried out in further processing. Basic work flow is shown in Figure 16 and the class diagram is shown in Figure 17.

In the Target Level Agreement a warning was specified that should be provided in the case, that external libraries are used and it should also output the names of those. Actually, in contrast to our assumptions the libraries are not detectable by reading the `.xml` files. Because of that this warning will not be supported. All components are able to handle classes and methods of unknown sources like external libraries, but of course these classes and methods will not be analyzed. On the other hand they will not prohibit our tool from working.

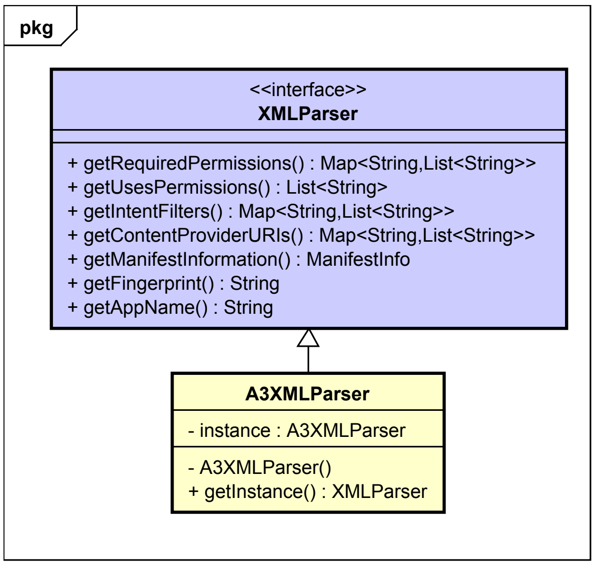

pkg
<<interface>>
XMLParser
+ getRequiredPermissions() : Map<String,List<String>>
+ getUsesPermissions() : List<String>
+ getIntentFilters() : Map<String,List<String>>
+ getContentProviderURIs() : Map<String,List<String>>
+ getManifestInformation() : ManifestInfo
+ getFingerprint() : String
+ getAppName() : String
A3XMLParser
- instance : A3XMLParser
- A3XMLParser()
+ getInstance() : XMLParser

Figure 17: `A3XMLParser` Class Diagram

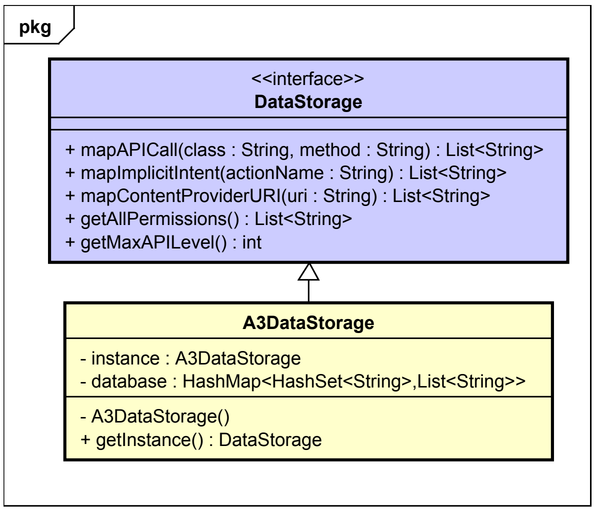

pkg
<<interface>>
DataStorage
+ mapAPICall(class : String, method : String) : List<String>
+ mapImplicitIntent(actionName : String) : List<String>
+ mapContentProviderURI(uri : String) : List<String>
+ getAllPermissions() : List<String>
+ getMaxAPILevel() : int
A3DataStorage
- instance : A3DataStorage
- database : HashMap<HashSet<String>,List<String>>
- A3DataStorage()
+ getInstance() : DataStorage

Figure 18: `A3DataStorage` Class Diagram

DataStorage The `A3DataStorage` (see Figure 18) implements the `DataStorage` interface and the singleton pattern pattern. By that, it implements the five methods of the interface:

1. `mapAPICall(...)`: This method maps a method call to a list of permissions. The method is referenced by the method name (parameter `method`) and the class the method belongs to (parameter `class`).
2. `mapImplicitIntent(...)`: An implicit intent which is calling a Android system App can be mapped to a list of permissions by this method. The intent is recognized by its action name.
3. `mapContentProviderURI(...)`: This will map a list of permissions to a URI which is used to access a content provider of the Android system. As input the URI itself is required.
4. `getAllPermissions()`: This method will simply return a list of all permissions that are provided by Android.
5. `getMaxAPILevel()`: This method will return the API level that is supported by the `DataStorage`. In case of the `A3DataStorage` that will be delivered with the tool the return value will be 22.

These five methods describe the whole functionality of any `DataStorage`.

The `A3DataStorage` holds all information in a hashmap. At the moment an analysis is started this hashmap is filled with key-value pairs. In this case this will be hashsets of Strings as keys and lists of Strings as values. The hashsets reference the methods and the lists define the permissions required by the referenced methods.

To fill the `A3DataStorage` some textfiles containing the required information will be read. These textfiles will be generated for Android API Level 22 and be delivered with the tool. In order to change the supported API Level these textfiles have to be replaced. For the creation of the files the tool PScout[9] was used. It can be reused to generate the same textfiles for another API Level. The interface also provides the possibility to exchange the whole `DataStorage`.

4.1.4 Enhancer

The `Enhacer` class will prepare the raw input for any class implementing the `GraphGenerator` interface. It will use Soot to gather information about the source code. On the other hand it will use an implementation of the interfaces `XMLParser` and `DataStorage` to get information about the uses of permissions in the provided `.apk` file. As output the `Enhancer` will create an `EnhancedInput` object. If an aggregation analysis will be run

[9]Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie. PScout: Analyzing the Android Permission Specification. In the Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012). October 2012.

the `Enhancer` only forwards the previously generated results. This happens by calling the `collectResults(...)` function. In this case the output will be a `ResultInput` object instead an `EnhancedInput` object.

Such an object has a tree structure reflecting the structure of the analyzed App. Every `EnhancedInput` object contains an arbitrary number of `Element` nodes connected by edges hierarchically from the root to the leafs. The complete data structure is described in Section 4.1.2, but here is a brief summary: The root node always is an `Element` object of the inheriting `App` class. Every root node will have at least one `Component` element as child. But it can have more `Component` or `Class` nodes as children. Every `Component` or `Class` node in turn can have an arbitrary number of `Method` nodes as children. Again these nodes can have some `Statement` nodes as children.

The structure of the `Enhancer` is shown in the class diagram in Figure 19. The sequence diagram (see Figure 20) shows a possible execution scenario for the `Enhancer`. Generating the `EnhancedInput` starts with an instantiation of the `LayerLinker` class. The `LayerLinker` will immediately instantiate an `App` and a `PermissionMapper` object. The `App` object refers to the analyzed App and contains the App's name. Therefore, the name of the App will be requested. The primary task of the `PermissionMapper` is to map permissions to all the elements contained in an `EnhancedInput` object. In order to do so, it will get a list of all permissions available on the Android system and form a map which maps Strings (permission names) to `Permission` objects. One object for each item in the list.

Then the first permissions can be mapped to the `App` object by requesting a list of permissions that contains the permissions which are defined in the 'uses' tags of the analyzed App's manifest. For example according to the following code snippet this list would contain the *android.permission.CAMERA* and the *android.permission.INTERNET* permissions:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET" />
  <application ... >
    <provider android:name="TestProvider"
      android:permission="android.permission.CAMERA"
    </provider>
    ...
```

The `PermissionMapper` needs more information from the Apps `.xml` files. It requests a map which maps permissions to Android components. According to the code snippet above this map would map the *android.permission.CAMERA* to the Content Provider *TestProvider*. Another map has to be requested by the `PermissionMapper`. This map is assigning a list of URIs to content provider components. All described requests will be responded by a class implementing the `XMLParser` interface. In the class diagram this is the `A3XMLParser` class (see Section 4.1.3).

But since the `PermissionMapper` is not only mapping permissions to `Elements` of `App` and `Component` type, it will need more information. To receive that information

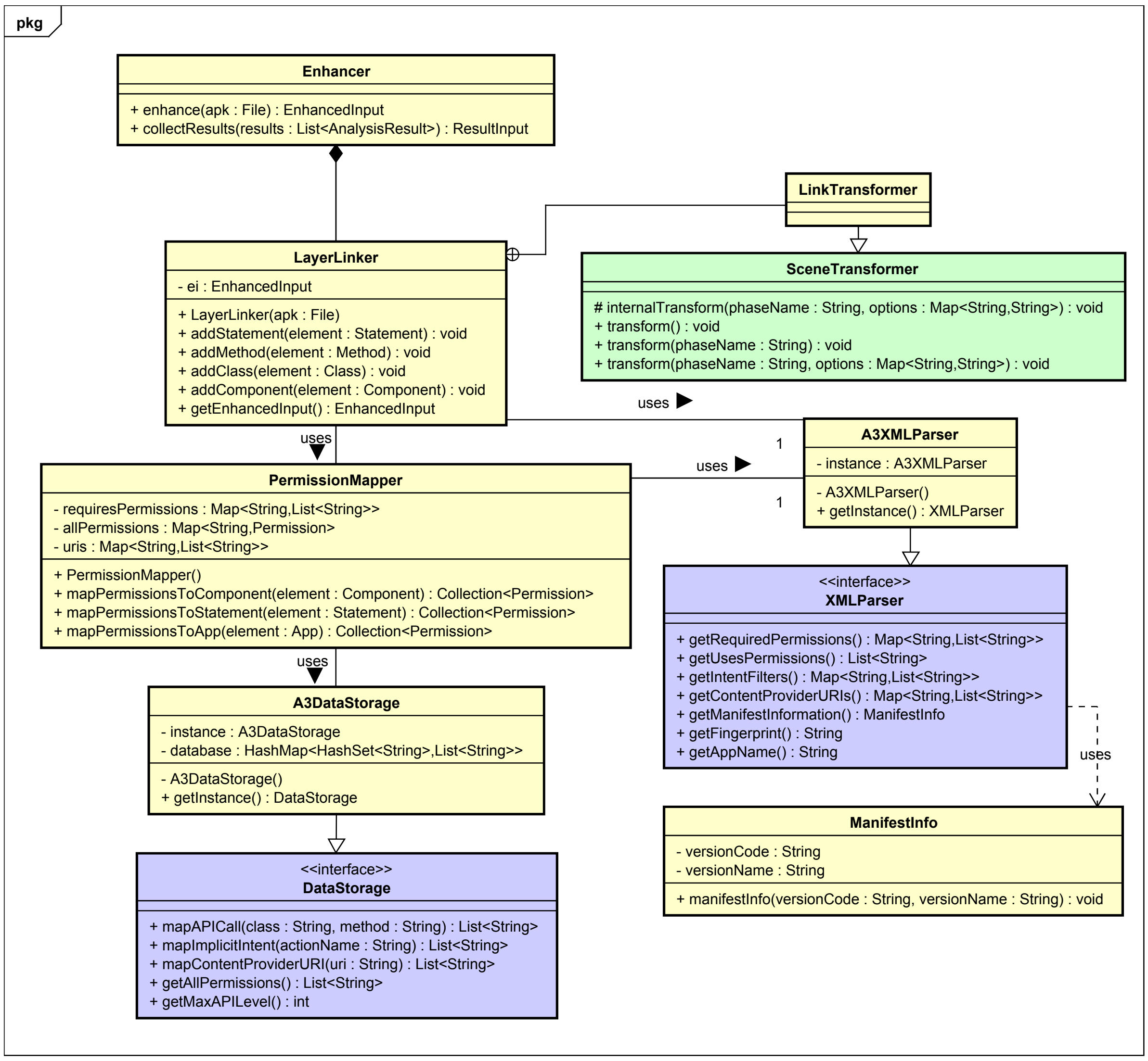

pkg
Enhancer
+ enhance(apk : File) : EnhancedInput
+ collectResults(results : List<AnalysisResult>) : ResultInput
LinkTransformer
LayerLinker
- ei : EnhancedInput
+ LayerLinker(apk : File)
+ addStatement(element : Statement) : void
+ addMethod(element : Method) : void
+ addClass(element : Class) : void
+ addComponent(element : Component) : void
+ getEnhancedInput() : EnhancedInput
SceneTransformer
# internalTransform(phaseName : String, options : Map<String,String>) : void
+ transform() : void
+ transform(phaseName : String) : void
+ transform(phaseName : String, options : Map<String,String>) : void
uses
uses
1
1
A3XMLParser
- instance : A3XMLParser
- A3XMLParser()
+ getInstance() : XMLParser
PermissionMapper
- requiresPermissions : Map<String,List<String>>
- allPermissions : Map<String,Permission>
- uris : Map<String,List<String>>
+ PermissionMapper()
+ mapPermissionsToComponent(element : Component) : Collection<Permission>
+ mapPermissionsToStatement(element : Statement) : Collection<Permission>
+ mapPermissionsToApp(element : App) : Collection<Permission>
<<interface>>
XMLParser
+ getRequiredPermissions() : Map<String,List<String>>
+ getUsesPermissions() : List<String>
+ getIntentFilters() : Map<String,List<String>>
+ getContentProviderURIs() : Map<String,List<String>>
+ getManifestInformation() : ManifestInfo
+ getFingerprint() : String
+ getAppName() : String
uses
uses
A3DataStorage
- instance : A3DataStorage
- database : HashMap<HashSet<String>,List<String>>
- A3DataStorage()
+ getInstance() : DataStorage
uses
ManifestInfo
- versionCode : String
- versionName : String
+ manifestInfo(versionCode : String, versionName : String) : void
<<interface>>
DataStorage
+ mapAPICall(class : String, method : String) : List<String>
+ mapImplicitIntent(actionName : String) : List<String>
+ mapContentProviderURI(uri : String) : List<String>
+ getAllPermissions() : List<String>
+ getMaxAPILevel() : int

Figure 19: `Enhancer` Class Diagram

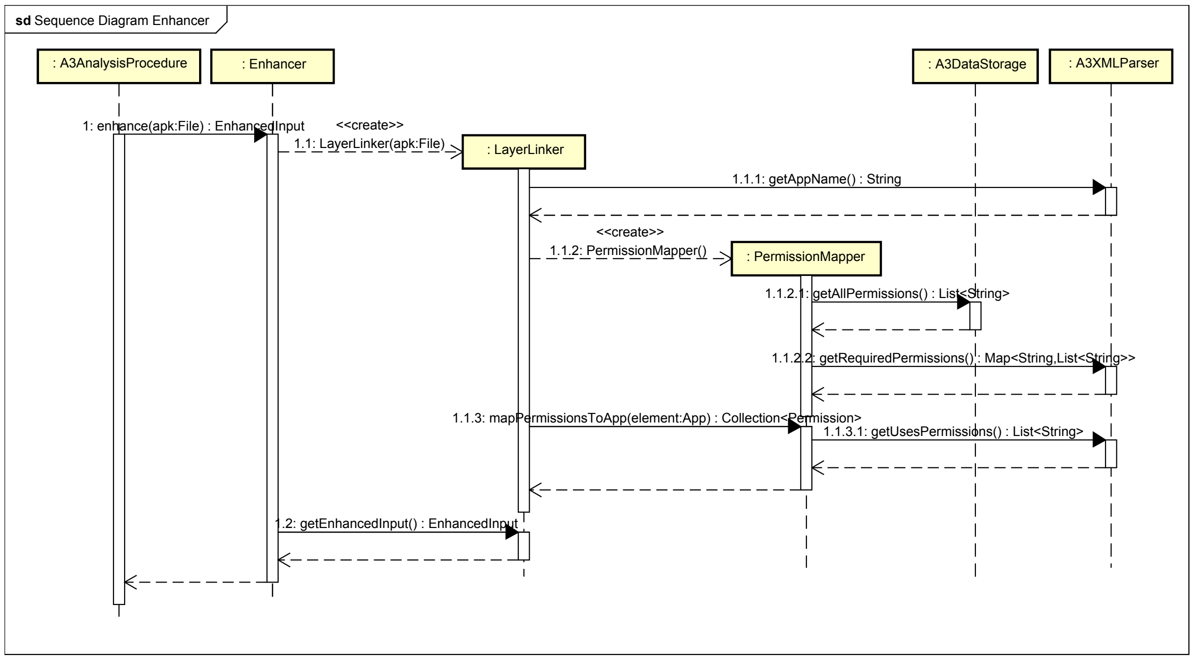

sd Sequence Diagram Enhancer
: A3AnalysisProcedure
: Enhancer
: A3DataStorage
: A3XMLParser
1: enhance(apk:File) : EnhancedInput
<<create>>
1.1: LayerLinker(apk:File)
: LayerLinker
1.1.1: getAppName() : String
<<create>>
1.1.2: PermissionMapper()
: PermissionMapper
1.1.2.1: getAllPermissions() : List<String>
1.1.2.2: getRequiredPermissions() : Map<String,List<String>>
1.1.3: mapPermissionsToApp(element:App) : Collection<Permission>
1.1.3.1: getUsesPermissions() : List<String>
1.2: getEnhancedInput() : EnhancedInput

Figure 20: `Enhancer` Sequence Diagram

a class implementing the `DataStorage` interface will come into play. Such a class is the `A3DataStorage` (see Section 4.1.3). It provides three methods to map permissions to all kinds of `Statements`: `mapAPICall(...)`, `mapImplicitIntent(...)` and `mapContentProviderURI(...)`. One of these methods will always be called if the `PermissionMapper` wants to map a set of permissions to a `Statement`, because there are three types of statements that can require a permission before execution:

1. Android API calls
2. Intent definitions referring to Android system Apps
3. Accesses to content providers of the Android system

In order to fill the `EnhancedInput` the `LinkTransformer` is needed. It is overwriting a Soot class. By that the `LinkTransformer` will call the methods `addStatement(...)`, `addMethod(...)`, `addClass(...)` and `addComponent(...)` when Soot is running with the `.apk` file of the targeted App as input. This is described in more detail in Section 3. One exemplary execution of these methods is shown in another sequence diagram (see Figure 21). Everything happening there would happen directly before Step 1.2 in the first sequence diagram. Step 1 shows, how a new `Component` is added by the `LinkTransformer`. Part of it is Step 1.1 to get the permissions which are required by that component. In Step 2 a new `Method` is added to the tree. Step 3 and Step 4 show, how to add a `Statement` to the tree. If the statement belongs to an Android API Call, the `A3DataStorage` will be called to map the involved permissions (see Step 3.1.1). The same happens if it is an implicit intent which

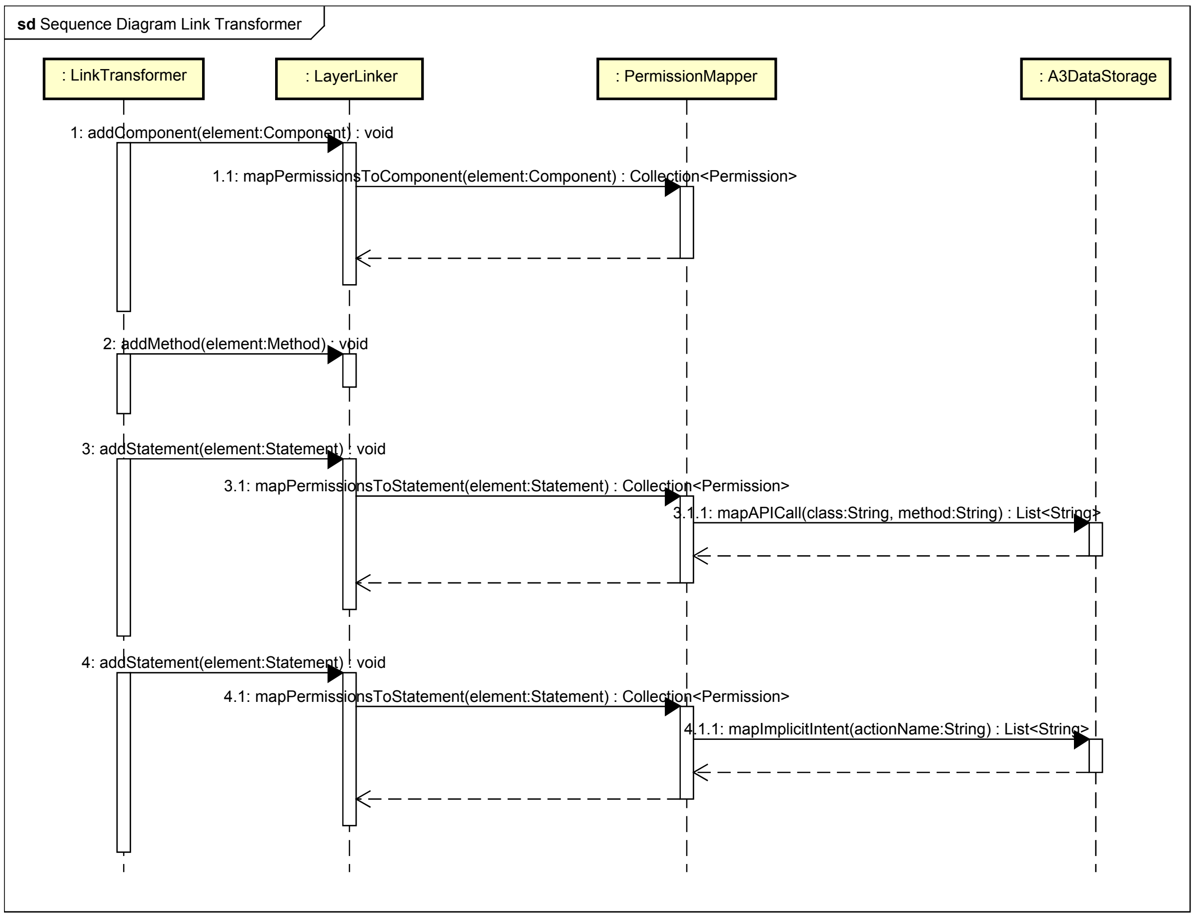
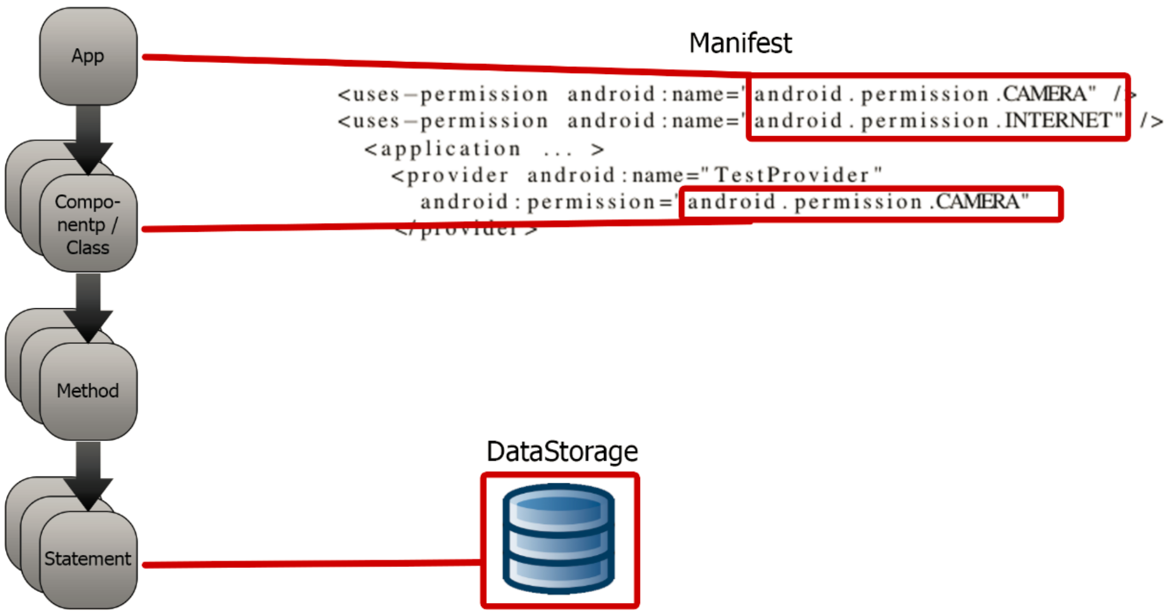

sd Sequence Diagram Link Transformer
: LinkTransformer
: LayerLinker
: PermissionMapper
: A3DataStorage
1: addComponent(element:Component) : void
1.1: mapPermissionsToComponent(element:Component) : Collection<Permission>
2: addMethod(element:Method) : void
3: addStatement(element:Statement) : void
3.1: mapPermissionsToStatement(element:Statement) : Collection<Permission>
3.1.1: mapAPICall(class:String, method:String) : List<String>
4: addStatement(element:Statement) : void
4.1: mapPermissionsToStatement(element:Statement) : Collection<Permission>
4.1.1: mapImplicitIntent(actionName:String) : List<String>

Figure 21: Sequence Diagram Showing Executions of the Methods `addStatement(...)`, `addMethod(...)`, `addClass(...)` and `addComponent(...)`



```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.INTERNET" />
  <application ... >
    <provider android:name="TestProvider"
      android:permission="android.permission.CAMERA"
    </provider>
```

Figure 22: Summary of the `Enhancer`'s Function in Form of a Sketch

calls a system App (see Step 4.1.1) or if it accesses a system content provider. To any other kind of statement no permissions are assigned.

To summarize the functionality of the `Enhancer` we can say, that it generates an `EnhancedInput` object that contains the structure of the source code down to any statement and has permissions assigned to almost all the `Elements` representing that structure as shown in Figure 22. This will not happen in case of an aggregation analysis. Then the `Enhancer` will only forward the previously collected results.

4.1.5 Usage Protocols

Next we want to provide some usage protocols for the tool framework in form of state charts. They will describe how the `Client` has to interact with the framework as well as how the `Client` can gather information from the `AnalysisResult`.

The state chart in Figure 23 models the steps the `Client` has to do to perform an analysis. We assume the `Client` has already collected all required information for analysis configuration for a certain analysis. Then the `Client` can create an `AnalysisFactory` for the chosen analysis by providing the collected configuration. After calling the `createAnalysis()` method of the `AnalysisFactory`, the `Client` receives a list of `Analysis` objects. This list will be passed to the `AnalysisRunner` via `analyze(...)` after the `Client` created an instance of the `AnalysisRunner`. Finally, the `AnalysisRunner` returns the `AnalysisResult` to the `Client`.

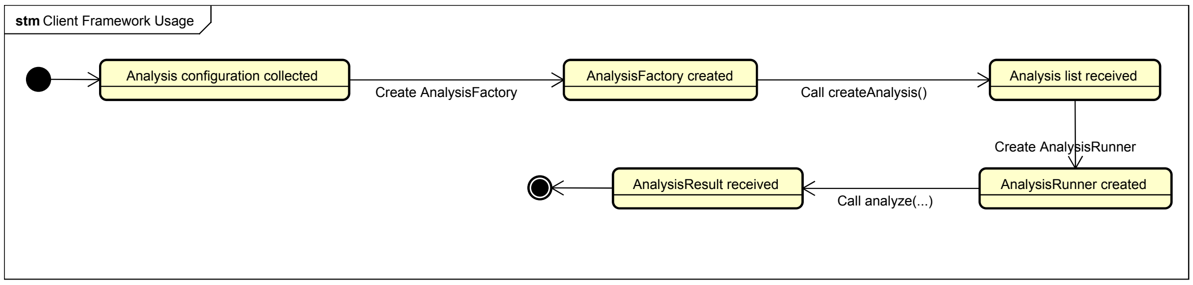


Figure 23: State Chart of the Framework Usage for the `Client`

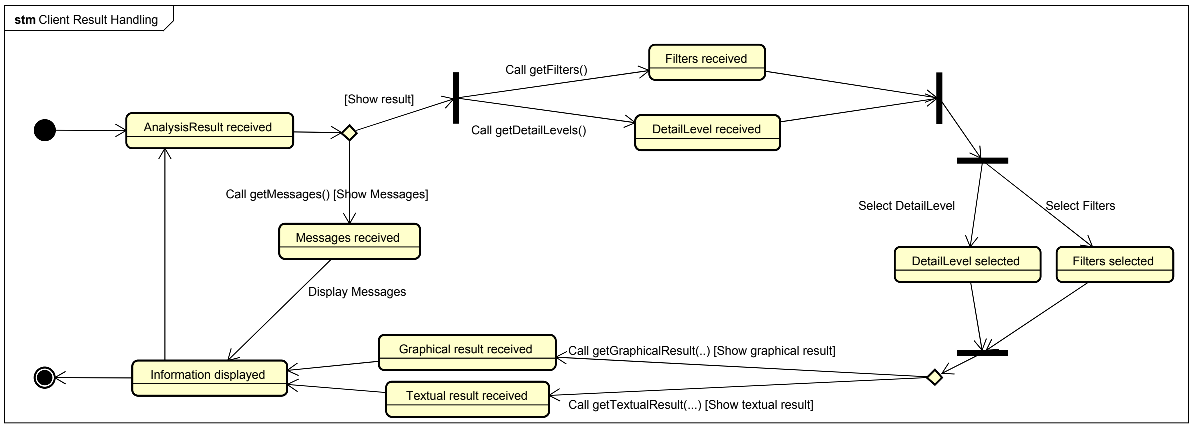


Figure 24: State Chart for Retrieving Result Representations from the `AnalysisResult`

After receiving the `AnalysisResult`, the `Client` likely wants to gather the contained information to display it to the user. This processing is described in Figure 24. If the `Client` wants to get the `Messages` inside the `AnalysisResult`, it has only to call the `getMessages()` method to get access to the `Messages`. The information contained in a `Message` can be extracted via getter methods and then displayed to the user. To get a result representation from the `AnalysisResult` is more complex since results may be very unique. At first the `Client` has to get the `DetailLevels` and the filters the `AnalysisResult` provides via the methods `getDetailLevels()` and `getFilters()`. Afterward the `Client` has to select a certain `DetailLevel` out of the provided ones and a subset of the provided filters. Depending on the kind of result representation (graphical or textual) the `Client` can now call `getGraphicalResult(...)` or respectively `getTextualResult(...)` while passing the selections as parameters to the result. The returned String will contain the information in the Graphviz DOT language or in the HTML5 language (as specified in Section 4.1.2) which can both be easily parsed and displayed to the user. After the `Client` processed the gathered information it is of course possible to request new information from the `AnalysisResult`.

4.1.6 Software Design Patterns

For the tool framework we want to provide features that reduce the workload to extend/adapt our tool to new tasks and environments. Parts of those features were already described above, using interfaces for the core services (Sec. 4.1.3), separating the analysis logic (Sec. 4.1.1) from the user interface and using external standard languages for information exchange to the UI (Sec. 4.1.2).

To perfect this first approach, we included several software design patterns[10] into the framework. In the following list all used patterns are named including a description what they are for and where they are used.

- The composite pattern is a structural pattern and is used when a part-whole hierarchy has to be modeled. The advantage of this pattern is that a functional class that this hierarchical structure uses can treat each element equally for all common properties. The core of this pattern is the abstract class `Component` which contains the common part for all objects in an hierarchy instance.

  We used a slightly adapted version of the composite pattern in the `EnhancedInput` where the `Element` class is the `Component`.

- The strategy pattern is a behavioral pattern and is used when it is necessary to select an algorithm's behavior at runtime. To do so, the algorithm's logic that should be variable is outsourced to an interface. Concrete classes will then implement that interface with different behavior. The fixed logic of the algorithm is contained in the class `Context` that owns an instance of the interface `Strategy`.

  We used the strategy pattern two times inside the `Analysis`. For the general part the `AnalysisProcedure` is the `Strategy` and the `Analysis` is the `Context`. Additionally the specific `A3AnalysisProcedure` again is a `Context` with the two `Strategy` interfaces `GraphGenerator` and `Analyzer`.

- The factory method pattern is a creational pattern and can be used when the creation of a concrete object should be decoupled from the creation process. When this pattern is used in a program, the program can be extended by new kinds of concrete objects without the need to change the creation process. The pattern consist of the interface `Factory` that has a method for creating concrete implementations of the interface `Product`.

  We used the factory method pattern in the interface `AnalysisFactory`. Its method does not return our version of a `Product` directly. This is because we had to encapsulate the `Product` which is the interface `AnalysisProcedure` into the class `Analysis` for the usage of the strategy pattern.

- The singleton pattern is also a creational pattern and can be used when only a single instance of a certain class is allowed during runtime. The singleton pattern realizes this

[10]As introduced by Gamma, Helm, Johnson and Vlissides in *Design Patterns - Elements of Reusable Object-Oriented Software, Prentice Hall*

restriction. Therefore, a singleton class has a private constructor and a static method `instance()` to offer the instance to other classes.

In our document we already mentioned in Section 4.1.3 that our core services `XML-Parser` and `DataStorage` will implement this pattern to be publicly available for all analysis classes as well as for the `Client`.

4.2 AnalysisProcedure for Level 1

The structure of the `A3AnalysisProcedure` for Level 1 is shown in the class diagram in Figure 25. Primarily it is an implementation of the `AnalysisProcedure` interface and consists out of an aggregation of three classes:

- `Enhancer` (Same class for any level of analysis.)
- `GraphGeneratorLvl1` (Implementing the `GraphGenerator` interface.)
- `AnalyzerLvl1` (Implementing the `Analyzer` interface.)

The `Enhancer` class and its function is described and explained in Section 4.1.4. The last two level specific classes (`GraphGeneratorLvl1`, `AnalyzerLvl1`) are briefly introduced in the following. Their function is to realize the Level 1 analysis. Therefore, the `GraphGeneratorLvl1` is connecting elements of the `EnhancedInput`, created by the `Enhancer`, according to the explicit intents used in the source code of the analyzed App. For instance, one edge for each explicit intent is added between the definition of the intent and the targeted component. After adding all these edges the `AnalyzerLvl1` will start the analysis. In case of a Level 1 analysis this means, that it will assign a group to any permission involved in the analyzed App. For example it will automatically assign the REQUIRED group to a permission, if there exists a 'uses' definition for this permission in the manifest and if there exists a use for this permission in the manifest or source code. All assignments to groups will be collected in an `AnalysisResultLvl1` object as result.

The next subsection will introduce the `AnalysisFactory` for this level of analysis. The two subsections after the next one will explain the structure and the function of the `GraphGeneratorLvl1` and the `AnalyzerLvl1` in more detail. In Subsection 4.2.4 the structure of the `AnalysisResultLvl1` is explained.

4.2.1 AnalysisFactoryLvl1

Figure 26 shows a class diagram of the interface `AnalysisFactory`, namely the `AnalysisFactoryLvl1`. The first parameter of the constructor belongs to the `.apk` file of App that will be analyzed, followed by a second parameter containing the previous result. The second parameter will be `null` as long as SUMMARY mode is selected.

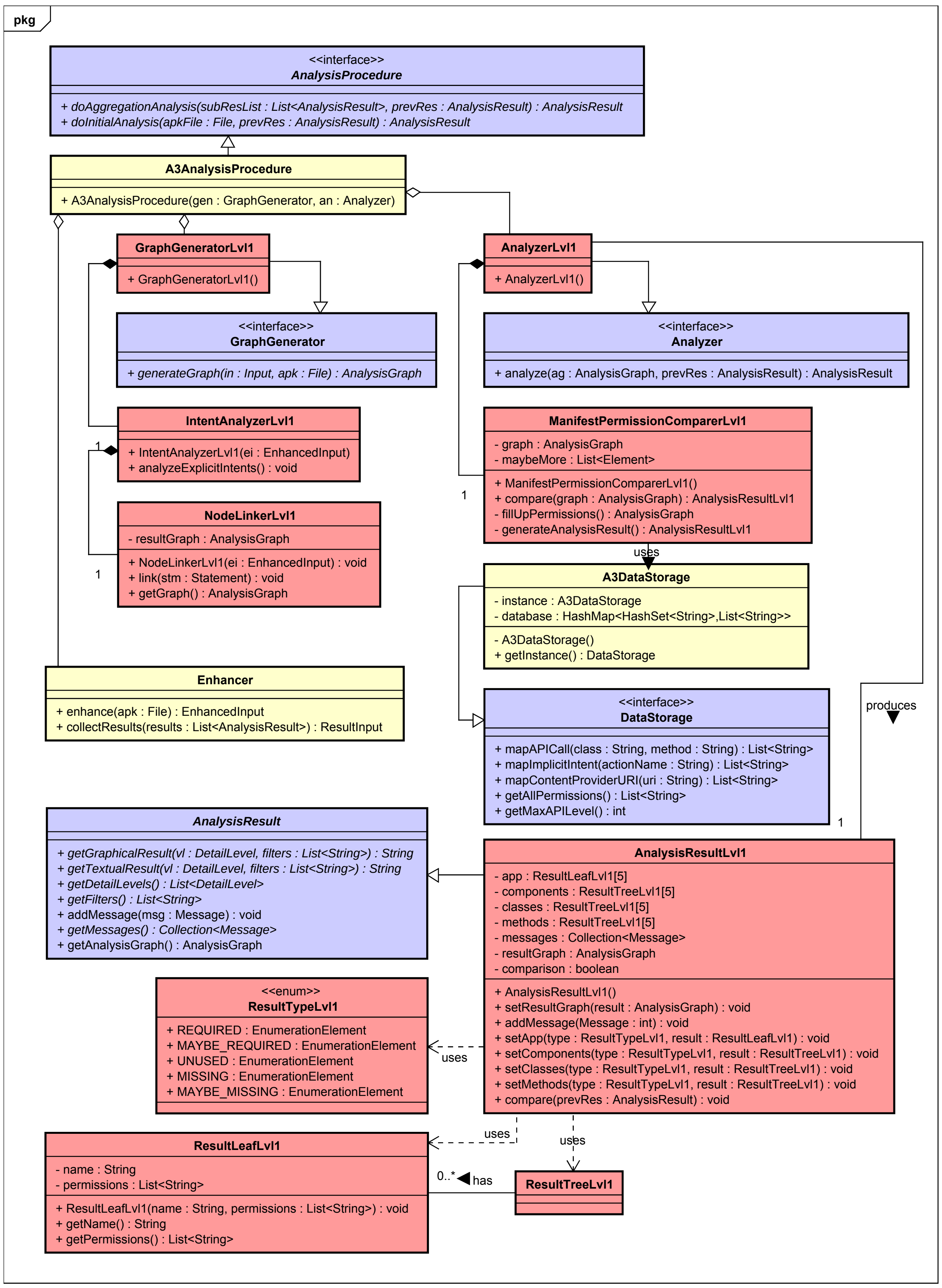

pkg
<<interface>>
AnalysisProcedure
+ doAggregationAnalysis(subResList : List<AnalysisResult>, prevRes : AnalysisResult) : AnalysisResult
+ doInitialAnalysis(apkFile : File, prevRes : AnalysisResult) : AnalysisResult
A3AnalysisProcedure
+ A3AnalysisProcedure(gen : GraphGenerator, an : Analyzer)
GraphGeneratorLvl1
+ GraphGeneratorLvl1()
AnalyzerLvl1
+ AnalyzerLvl1()
<<interface>>
GraphGenerator
+ generateGraph(in : Input, apk : File) : AnalysisGraph
<<interface>>
Analyzer
+ analyze(ag : AnalysisGraph, prevRes : AnalysisResult) : AnalysisResult
IntentAnalyzerLvl1
+ IntentAnalyzerLvl1(ei : EnhancedInput)
+ analyzeExplicitIntents() : void
1
NodeLinkerLvl1
- resultGraph : AnalysisGraph
+ NodeLinkerLvl1(ei : EnhancedInput) : void
+ link(stm : Statement) : void
+ getGraph() : AnalysisGraph
1
ManifestPermissionComparerLvl1
- graph : AnalysisGraph
- maybeMore : List<Element>
+ ManifestPermissionComparerLvl1()
+ compare(graph : AnalysisGraph) : AnalysisResultLvl1
- fillUpPermissions() : AnalysisGraph
- generateAnalysisResult() : AnalysisResultLvl1
1
uses
A3DataStorage
- instance : A3DataStorage
- database : HashMap<HashSet<String>,List<String>>
- A3DataStorage()
+ getInstance() : DataStorage
Enhancer
+ enhance(apk : File) : EnhancedInput
+ collectResults(results : List<AnalysisResult>) : ResultInput
<<interface>>
DataStorage
+ mapAPICall(class : String, method : String) : List<String>
+ mapImplicitIntent(actionName : String) : List<String>
+ mapContentProviderURI(uri : String) : List<String>
+ getAllPermissions() : List<String>
+ getMaxAPILevel() : int
produces
1
AnalysisResult
+ getGraphicalResult(vl : DetailLevel, filters : List<String>) : String
+ getTextualResult(vl : DetailLevel, filters : List<String>) : String
+ getDetailLevels() : List<DetailLevel>
+ getFilters() : List<String>
+ addMessage(msg : Message) : void
+ getMessages() : Collection<Message>
+ getAnalysisGraph() : AnalysisGraph
AnalysisResultLvl1
- app : ResultLeafLvl1[5]
- components : ResultTreeLvl1[5]
- classes : ResultTreeLvl1[5]
- methods : ResultTreeLvl1[5]
- messages : Collection<Message>
- resultGraph : AnalysisGraph
- comparison : boolean
+ AnalysisResultLvl1()
+ setResultGraph(result : AnalysisGraph) : void
+ addMessage(Message : int) : void
+ setApp(type : ResultTypeLvl1, result : ResultLeafLvl1) : void
+ setComponents(type : ResultTypeLvl1, result : ResultTreeLvl1) : void
+ setClasses(type : ResultTypeLvl1, result : ResultTreeLvl1) : void
+ setMethods(type : ResultTypeLvl1, result : ResultTreeLvl1) : void
+ compare(prevRes : AnalysisResult) : void
<<enum>>
ResultTypeLvl1
+ REQUIRED : EnumerationElement
+ MAYBE_REQUIRED : EnumerationElement
+ UNUSED : EnumerationElement
+ MISSING : EnumerationElement
+ MAYBE_MISSING : EnumerationElement
uses
uses
uses
ResultLeafLvl1
- name : String
- permissions : List<String>
+ ResultLeafLvl1(name : String, permissions : List<String>) : void
+ getName() : String
+ getPermissions() : List<String>
0..*
has
ResultTreeLvl1

Figure 25: Level 1 Class Diagram

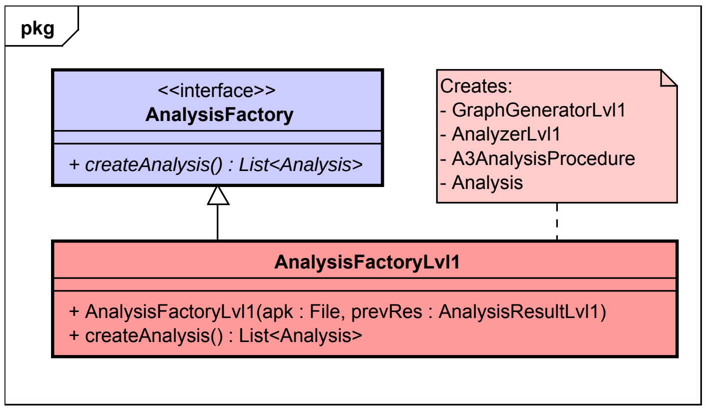

pkg
<<interface>>
AnalysisFactory
+ createAnalysis() : List<Analysis>
Creates:
- GraphGeneratorLvl1
- AnalyzerLvl1
- A3AnalysisProcedure
- Analysis
AnalysisFactoryLvl1
+ AnalysisFactoryLvl1(apk : File, prevRes : AnalysisResultLvl1)
+ createAnalysis() : List<Analysis>

Figure 26: `AnalysisFactoryLvl1` Class Diagram

The abstract method `createAnalysis(...)` is implemented by the `AnalysisFactoryLvl1`. This method will create a list of `Analysis` objects. Each object will know both parameters provided in the constructor of the factory object. In case of Level 1 the list will only contain one item, because only one analysis on a single App is created.

While executing the `createAnalysis()` method a `GraphGeneratorLvl1` and an `AnalyzerLvl1` as well as an `A3AnalysisProcedure` object are created. When creating the `A3AnalysisProcedure` object it gets access to the `GraphGenerator` and the `Analyzer`, because these object are given to the `A3AnalysisProcedure` as parameters. Since the `A3AnalysisProcedure` implements the `AnalysisProcedure` interface, the factory can now create an `Analysis` object and put this into the list that will be returned. The relationship between the `Analysis`, `AnalysisProcedure` and the `AnalysisFactory` is described in Section 4.1.1.

4.2.2 GraphGeneratorLvl1

The `GraphGeneratorLvl1` is an implementation of the `GraphGenerator` interface. By that it implements the `generateGraph(...)` function. This function will take an `Input` object as input and generates an object of type `AnalysisGraph` as output. In case of Level 1 the `Input` object will always be an object of type `EnhancedInput`, because the Level 1 analysis does not rely on other analyses that have to be executed before. Level 2b for example will rely on one or more Level 1 analyses (see Section 4.4).

One execution of the `generateGraph(...)` function is reflected in a sequence diagram shown in Figure 27. Once the `generateGraph(...)` function is executed an object of type `IntentAnalyzerLvl1` (IA) will be generated. The instantiation of this object directly leads to another instantiation of class `NodeLinkerLvl1` (NL). After the instantiation of the NL the analysis graph is created, namely by instantiating an object of type `AnalysisGraph`. Primarily the IA object has one function: `analyzeExplicitIntents()`. This function

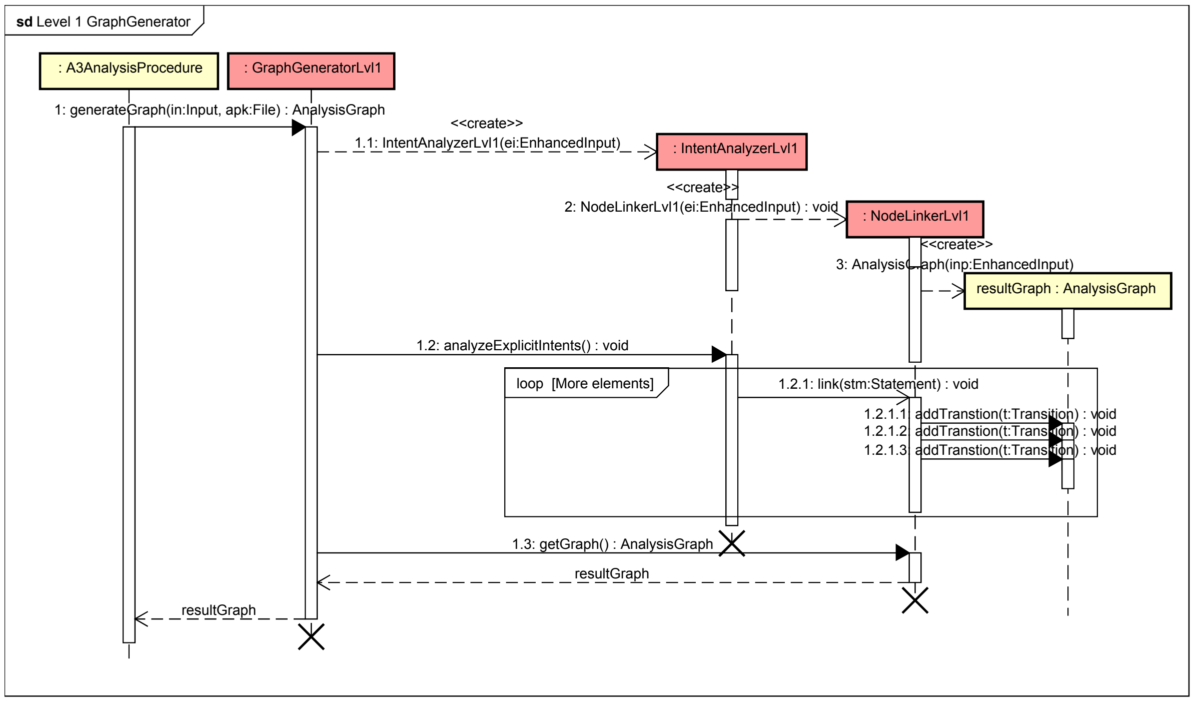

sd Level 1 GraphGenerator
: A3AnalysisProcedure
: GraphGeneratorLvl1
1: generateGraph(in:Input, apk:File) : AnalysisGraph
<<create>>
1.1: IntentAnalyzerLvl1(ei:EnhancedInput)
: IntentAnalyzerLvl1
<<create>>
2: NodeLinkerLvl1(ei:EnhancedInput) : void
: NodeLinkerLvl1
<<create>>
3: AnalysisGraph(inp:EnhancedInput)
resultGraph : AnalysisGraph
1.2: analyzeExplicitIntents() : void
loop [More elements]
1.2.1: link(stm:Statement) : void
1.2.1.1 addTranstion(t:Transition) : void
1.2.1.2 addTranstion(t:Transition) : void
1.2.1.3 addTranstion(t:Transition) : void
1.3: getGraph() : AnalysisGraph
resultGraph
resultGraph

Figure 27: `GraphGeneratorLvl1` Sequence Diagram

includes a loop, which is iterating over `Elements` stored in the `EnhancedInput` object. Once an Element is visited, which is a `Statement`, more precisely a definition of an explicit intent, the NL will be called to link that `Statement` to the targeted `Component`. This will happen by calling the `link(...)` function of the NL. Hence the NL will add three transitions to the analysis graph via `addTransition(...)` (see Figure 27). One from the `Statement` itself to the targeted `Component`. Another one from the `Method`, the `Statement` belongs to, to the targeted `Component`. And a last one from the `Component`, this `Method` is part of, to the targeted `Component`.

The whole execution of the loop is visualized by the statechart in Figure 28. The first reachable state is the `Searching` state. This state stands for the head of the loop and will always be reached before executing the loop's body. Once all `Elements` have been visited the endstate is reached and the execution of the loop ends. But as long as there are `Elements` which have not been visited yet the next state (`Element found`) will be reached. If the visited `Element` is not of type `Statement` the next `Element` will be visited. This means we return to the `Searching` state. If it is of type `Statement` we continue by entering the state `Statement found`. The same happens depending on the fact, whether the `Statement` is an explicit intent definition or not. Once reaching the state `Explicit intent found`, the intent definition itself will be checked. If it is valid according to the assumptions and limitations of the Target Level Agreement, the next state (`Target found`) will be reached. For example it will be checked whether the intent's target exists. Otherwise we will return to

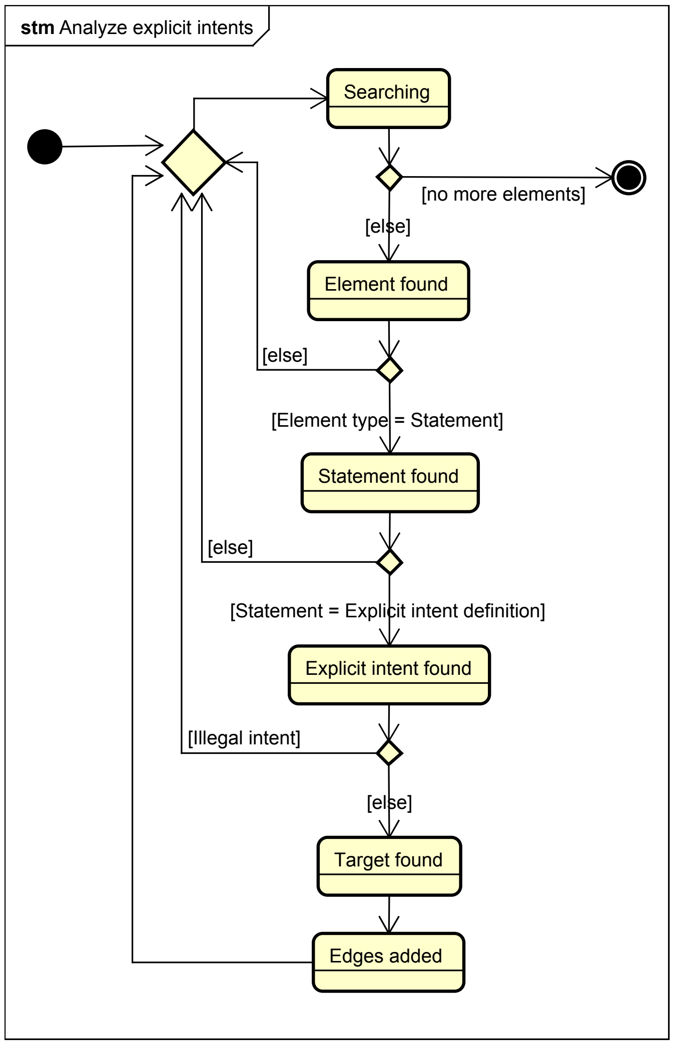


Figure 28: State Chart Showing the Loop while Analyzing Explicit Intents

the `Searching` state and the tool will provide a warning, that an illegal intent has been found. After reaching the `Target found` state, the NL will be called to add the required edges and we return to the `Searching` state.

After the execution of the loop the `GraphGeneratorLvl1` gets the generated `AnalysisGraph` object from the NL and returns it to the `A3AnalysisProcedure`. The `A3AnalysisProcedure` will start the next step described in the following subsection.

4.2.3 AnalyzerLvl1

The `Analyzer` interface serves as a blueprint for the `AnalyzerLvl1`. By that, the `AnalyzerLvl1` implements the method `analyze(...)`. This method will be called after the `GraphGeneratorLvl1` has finished creating the `AnalysisGraph`. This graph is the only input to the `analyze(...)` method, if SUMMARY mode is chosen. In COMPARISON mode a second input has to be provided, namely the previous result in form of an object of type `AnalysisResult`. Once the `analyze` method is executed, a `ManifestPermissionComparerLvl1` object is constructed. Objects of this class are used to compare the permissions declared in the Android manifest with permissions required by Android components and permissions needed by statements in the source code. All information about all involved permissions are already available since the `Enhancer` finished creating the `EnhancedInput` (see Section 4.1.4). Every `EnhancedInput` object contains one `App` object. All permissions which are declared in the Android manifest through uses-tags are already assigned to that object in form of one or more objects of type `Permission`. Equally, if a component of an App requires a permission, a `Permission` object will be assigned to the `Component` object representing that component. Same is valid for a statement in the source code that requires a permission for being executed. But there does not exist any information about permissions being assigned to methods which do not belong to Android API Calls and classes which are no Android components, yet.

The sequence diagram in Figure 29 reflects a possible execution of the `analyze(...)` method. First the `ManifestPermissionComparerLvl1` object is created. Then the `compare(...)` method of that object is called. Essentially, it will compute the analysis result by extending and comparing the provided permission information. For instance, if there exists a `Permission` object assigned to a `Statement` and to the `App` object then this permission is a REQUIRED permission.

The first step while computing the analysis result is to instantiate an object of type `AnalysisResultLvl1`. For any Level 1 analysis the result will always be represented by such an object. **The second step** is to fill up each node in the `AnalysisGraph` with the associated permissions. For example, assume permission A is assigned to a statement. This statement is part of a method. But permission A is not assigned to that method. Filling up the graph will assign these missing permission assignments. Therefore the method `fillUpPermissions()` is executed. How exactly this method works is reflected in the statechart in Figure 30. The

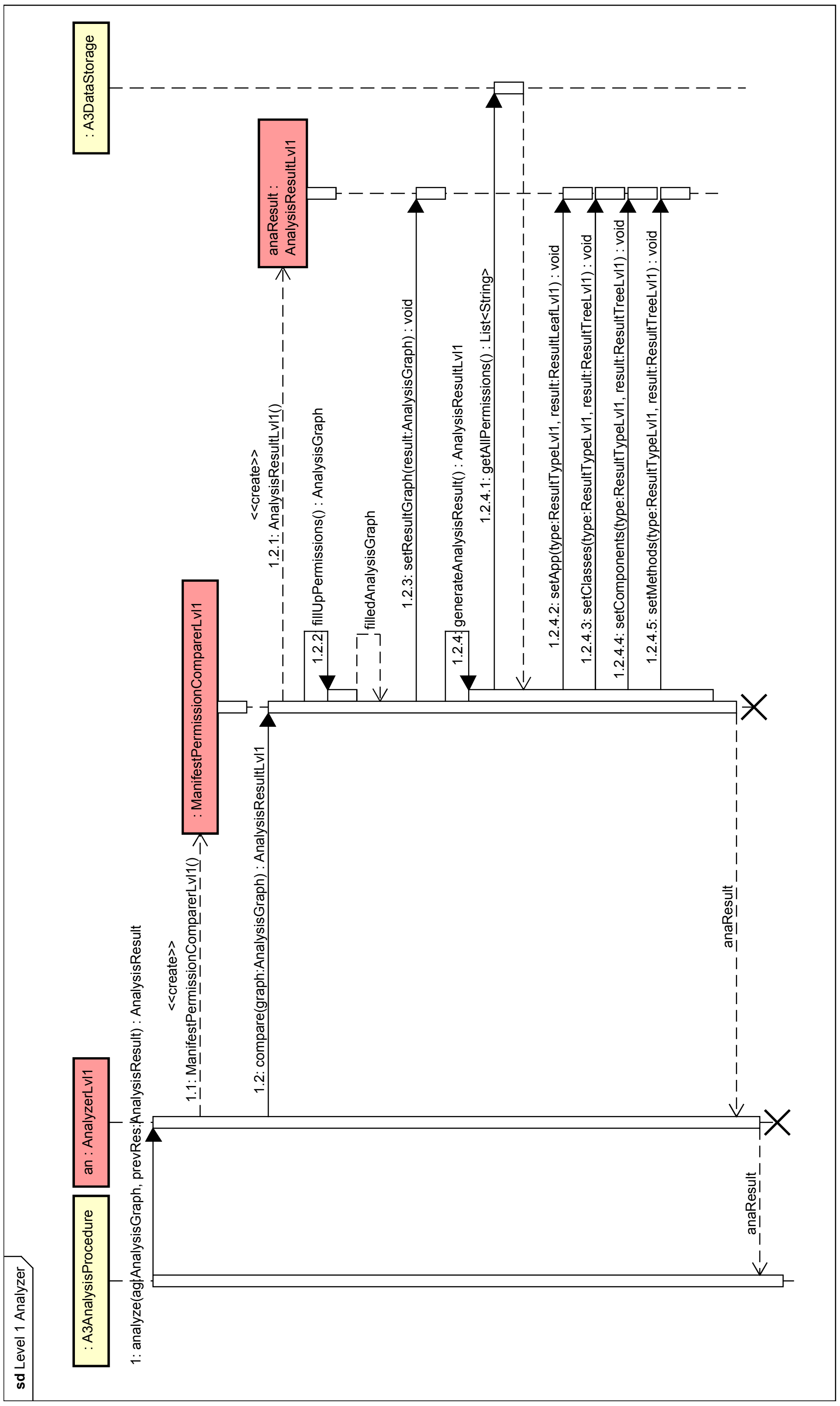

sd Level 1 Analyzer
: A3AnalysisProcedure
an : AnalyzerLvl1
: ManifestPermissionComparerLvl1
anaResult : AnalysisResultLvl1
: A3DataStorage
1: analyze(ag:AnalysisGraph, prevRes:AnalysisResult) : AnalysisResult
<<create>>
1.1: ManifestPermissionComparerLvl1()
1.2: compare(graph:AnalysisGraph) : AnalysisResultLvl1
<<create>>
1.2.1: AnalysisResultLvl1()
1.2.2: fillUpPermissions() : AnalysisGraph
filledAnalysisGraph
1.2.3: setResultGraph(result:AnalysisGraph) : void
1.2.4: generateAnalysisResult() : AnalysisResultLvl1
1.2.4.1: getAllPermissions() : List<String>
1.2.4.2: setApp(type:ResultTypeLvl1, result:ResultLeafLvl1) : void
1.2.4.3: setClasses(type:ResultTypeLvl1, result:ResultTreeLvl1) : void
1.2.4.4: setComponents(type:ResultTypeLvl1, result:ResultTreeLvl1) : void
1.2.4.5: setMethods(type:ResultTypeLvl1, result:ResultTreeLvl1) : void
anaResult
anaResult

Figure 29: `AnalyzerLvl1` Sequence Diagram

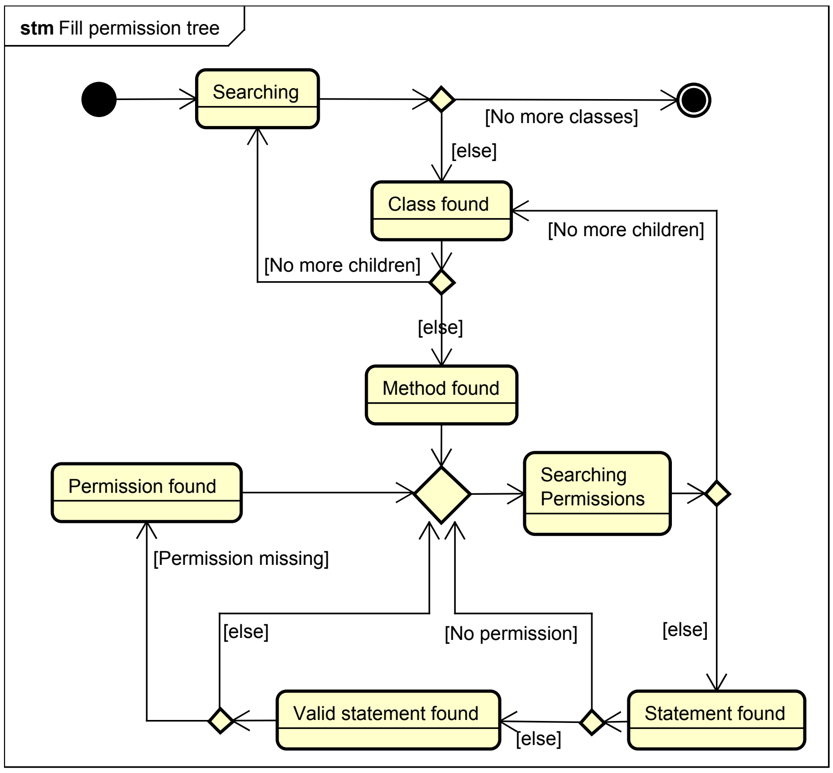


Figure 30: State Chart Reflecting the Method `fillUpPermissions()`

first reachable state is called `Searching`. The method iterates over all classes including components. In other words, a loop is running through all direct children of the `App` element. Once a new class is visited, the `Class found` state is reached. The same way the method iterates over all methods of that class and reach the state `Method found`. Then another two loops will iterate over all permissions assigned to any child of the visited method. Overall this makes four loops nested in each other, so that each permission assigned to any statement in any method of any class will be visited. The state `Searching Permissions` symbolizes the iteration over all permissions in all statements that are part of the currently visited method. When finding a `Statement`, the state `Statement found` is reached. If this `Statement` has no permissions assigned, the method returns to the previous state and if this `Statement` is an invalid statement, that `Statement` is marked by adding the associated `Element` object and all ancestors except the `App` node to the `maybeMore` list of the `ManifestPermissionComparerLvl1` and warning is provided. A statement is invalid if it fulfills one of the following criteria:

- It is an implicit intent and cannot be mapped by the `DataStorage` to a system App.
- It is an explicit intent and the targeted component cannot be resolved.
- It is a method call that does not refer to a Android API call.
- It is requesting a URI of a content provider, that
  - cannot be mapped to any URI in the manifest or the Android system.
  - can be mapped to content provider, which is granting a temporary permission for this URI.
- It is a statement which receives data from another component. In this special case the tool will also suggest a Level 2b analysis (see Section 4.4)
  (Example: *getIntent().getStringExtra("data")*)

In any other case the next reached state is `Valid statement found`. This means, that a statement has been found which has permissions assigned to it. In case of an explicit intent this will be the permissions required by the targeted component in the manifest. If this permissions are already assigned to the currently visited method, the method returns to the `Searching Permissions` state. Otherwise a permission is found that should be assigned to the method. The state `Permission found` is reached then and the permission is be assigned to the currently visited method. If that permission is not assigned to the currently visited class too, it will be assigned.
After one run through the whole graph the execution of `fillUpPermissions()` will end. All `Elements` are now in relation to any permission used by them.

After explaining the `fillUpPermissions()` method in detail, the description of the sequence diagram (see Figure 29) will be continued now. **The third step** (see Step 1.2.3) will set this filled graph as result graph in the `AnalysisResultLvl1` object by using the `setResultGraph(...)` method. This again can be seen in the sequence diagram

Table 1: Decision table (Generating analysis result)

Element type	**Group assigned**	**Permission is assigned to current element**	**Permission is assigned to any direct child of this element**	**Element is included in the maybeMore list**
App	REQUIRED	✓	✓	
	MAYBE_REQUIRED	✓	✗	✓
	UNUSED	✓	✗	✗
	MISSING	✗	✓	
	MAYBE_MISSING	✗	✗	✓
Component or class	REQUIRED	✓	0	
	MAYBE_REQUIRED	✓	✗	✓
	UNUSED	✓	✗	✗
	MISSING	✗	✓	
	MAYBE_MISSING	✗	✗	✓
Method	REQUIRED	✓	0	
	MAYBE_REQUIRED	✓	✗	✓
	UNUSED	✓	✗	✗
	MISSING	✗	✓	
	MAYBE_MISSING	✗	✗	✓

The fourth step while computing the analysis result is primarily the execution of the `generateAnalysisResult()` method of the `ManifestPermissionComparerLvl1`. But before doing that, the `ManifestPermissionComparerLvl1` requests a list of all permissions available in the `DataStorage` (see 1.2.4.1 in Figure 29). Once the `generateAnalysisResult()` method is called, the method iterates over all `Elements` except `Statements` in the `AnalysisGraph` which was filled up before. Each time an `Element` is visited, the method iterates over all available permissions existing in the current Android API version. Every permission will be assigned to the analysis result if it fulfills all conditions of one row in the decision table (see Table 1). For example, if a `Class` element A is visited and the current permission considered is B, then the following facts will be checked:

- Is permission B assigned to A?
- Is permission B assigned to any direct child of A?
- Is A contained in the `maybeMore` list, which holds a list of marked elements that might use more permissions than currently known.

According to the result of these checks the permission will be added appropriately to the analysis result. After iterating through the whole graph, the `maybeMore` list will be visited once more. If this list is not empty, a warning is provided and for each element in it all permissions that are not assigned will be considered as MAYBE_MISSING in the analysis result.

Finally, after executing all the above described steps, the `AnalysisResultLvl1` object has been instantiated and filled with all the information needed to show and filter the result. The next section will explain, how the information is stored in this object.

4.2.4 AnalysisResultLvl1

An `AnalysisResultLvl1` (see Figur 25) object stores the analysis result. Since the constructor has no parameters, the result parts are set after instantiation through the methods `setApp`, `setComponents`, `setClasses` and `setMethod`. One call of the `setApp(...)` method is creating a `ResultLeafLvl1` object. This object will be saved in the array `app`. The position in the array is defined by the `type` parameter of type `ResultTypeLvl1`. One call of the `setComponents(...)` method is adding a whole `ResultTreeLvl1` object including several leafs to the array `components`. Again the position in the array is defined by the result type. This method has to save a whole tree with depth one because in contrast to the `App` element there might exist more than one `Component` element. Equally, the `setClasses(...)` and `setMethod(...)` methods work. They will assign `ResultTreeLvl1`s to the arrays `classes` and `methods`.

Once the result is saved in this structure all filters can be applied easily. The unfiltered result will show all permissions saved in the array `app`. If the user wants to filter out e.g. the MISSING permissions the tool simply takes only one item of the array. On the other hand if the user wants to switch to detail level METHOD the tool can just change the array used to `methods`.

In COMPARISON mode the previous result will be available and can be displayed and filtered in the same way. Hence, the user can compare both results.

Additional warnings are added to the result, if the following permission groups are not empty: MAYBE_REQUIRED, UNUSED and MAYBE_MISSING. In the case, that the permission group MISSING is not empty, an error is provided. These warning messages and the error message will include the number of items in the associated group.

4.3 AnalysisProcedure for Level 2a

This section describes the class structure of the Android App Analysis tool for Level 2a. In addition to introducing this structure and describing it in detail, some sequence diagrams will explain how the classes typically will behave when executing a Level 2a analysis.

The important classes our tool uses to do a Level 2a analysis are displayed in Figure 32, 35, 36 and 31. The yellow and blue classes can be found in more detail in Sec. 4.1. In this section they are only added to the class diagram to visualize the relationship between the Level 2a

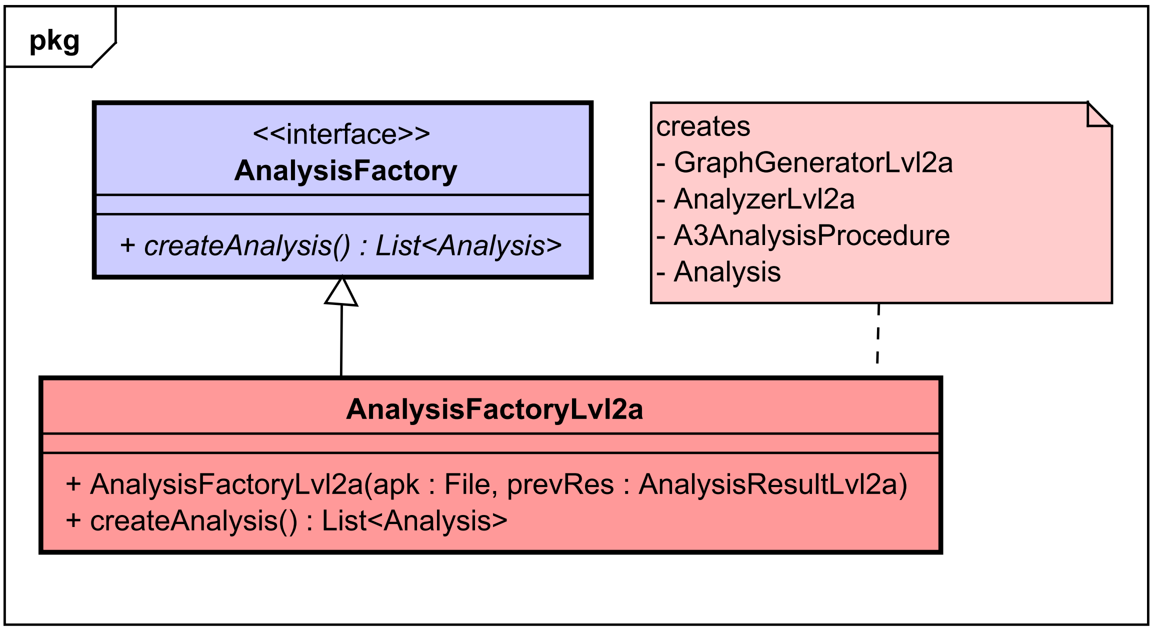

pkg
<<interface>>
AnalysisFactory
+ createAnalysis() : List<Analysis>
creates
- GraphGeneratorLvl2a
- AnalyzerLvl2a
- A3AnalysisProcedure
- Analysis
AnalysisFactoryLvl2a
+ AnalysisFactoryLvl2a(apk : File, prevRes : AnalysisResultLvl2a)
+ createAnalysis() : List<Analysis>

Figure 31: Class Diagram for the `Level2aFactory`

specific classes and those of the general framework. Therefore, not all attributes, methods and associations are displayed in the following figures. Furthermore, abstract methods of interfaces are not explicitly visualized in all concrete classes implementing the corresponding interface for better overview.

As explained in section 4.1.1 the `A3AnalysisProcedure` consists of the three main components `Enhancer`, `GraphGenereator` and `Analyzer`. Two of them are specific for each level and are reachable for the procedure via the interfaces `GraphGenerator` and `Analyzer`. Therefore, describing the structure of those two components (Fig. 32 and 35) will be the main part in this section. But before starting with the two components the `AnalysisFactory` for Level 2a will be shortly introduced.

4.3.1 AnalysisFactoryLvl2a

Figure 31 shows the concrete class `AnalysisFactoryLvl2a` which implements the interface `AnalysisFactory` (see Fig. 12). The constructor gets as parameters the `.apk` to be analyzed and for COMPARISON mode in addition a previous result of a level 2a analysis. It will be an object of type `AnalysisResultLvl2a` (see Sec. 4.3.4). If SUMMARY mode is selected, the second parameter will be `null`.

The `AnalysisFactoryLvl2a` implements the abstract method `createAnalysis()` provided by the interface `AnalysisFactory`. In this method the `AnalysisFacto-`

ryLvl2a constructs a list of Analysis objects passing the .apk and if available the previous result automatically to the Analysis object via its constructor. For Level 2a the list of Analysis objects contains only one element.

The method createAnalysis() creates a GraphGeneratorLvl2a and an AnalyzerLvl2a and then builds an A3AnalysisProcedure passing the GraphGenerator and the Analyzer via its constructor to the A3AnalysisProcedure. Since the A3AnalysisProcedure implements the interface AnalysisProcedure, the factory can create now an Analysis and pass the A3AnalysisProcedure, the .apk to be analyzed and if existing the previous result as parameters via the constructor to the just created Analysis. This Analysis is added to an empty list and that list is then returned. The relationship between the Analysis, AnalysisProcedure and the AnalysisFactory was described in Section 4.1.1 in detail.

4.3.2 GraphGeneratorLvl2a

The GraphGenerator has two subcomponents, an IntentAnalyzer and a NodeLinker (see Sec. 2.2). Therefore, the GraphGeneratorLvl2a has the two aggregation associations to the classes NodeLinkerLvl2a and GraphGeneratorLvl2a. The GraphGeneratorLvl2a has to create an AnalysisGraph in the method generateGraph(...). Since the analysis graph for Level 2a needs some additional specific nodes in addition to those provided in the overall framework (see Fig. 15), a class AnalysisGraphLvl2a which inherits the class AnalysisGraph is needed(see Fig. 36). The behaviour of the GraphGeneratorLvl2a and its corresponding classes can be explained best by looking at a typical execution shown in the sequence diagram in Figure 33. The A3AnalysisProcedure will call the method generateGraph(...) passing over the .apk as well as an Input object which in case of Level 2a is an EnhancedInput (see Fig. 15). The GraphGeneratorLvl2a then creates an AnalysisGraphLvl2a from that Input and enriches the graph via calling analyzeExplicitIntents(...) on the IntentAnalyzerLvl2a and link(...) on the NodeLinkerLvl2a. Already at that stage the Input will be converted to an AnalysisGraphLvl2a to avoid that the NodeLinkerLvl2a or the IntentAnalyzerLvl2a have to do the transformation. This workflow allows it to call the IntentAnalyzerLvl2a and the NodeLinkerLvl2a in an arbitrary order since none of them is dependent on the transformation done by one of them but they both work on an AnalysisGraphLvl2a object.

When calling analyzeExplicitIntents(...) the IntentAnalyzerLvl2a gets the AnalysisGraphLvl2a which at that point has the EnhancedInput but no transitions yet. All Statements in the EnhancedInput will be analyzed to find out whether they are explicit intents and if this is the case, a transition from the Statement to the corresponding Component will be added to the AnalysisGraphLvl2a. After adding the Transition the IntentAnalyzerLvl2a calls addTransitionsForExplicitIntent(...) on the NodeLinkerLvl2a. The NodeLinkerLvl2a then has

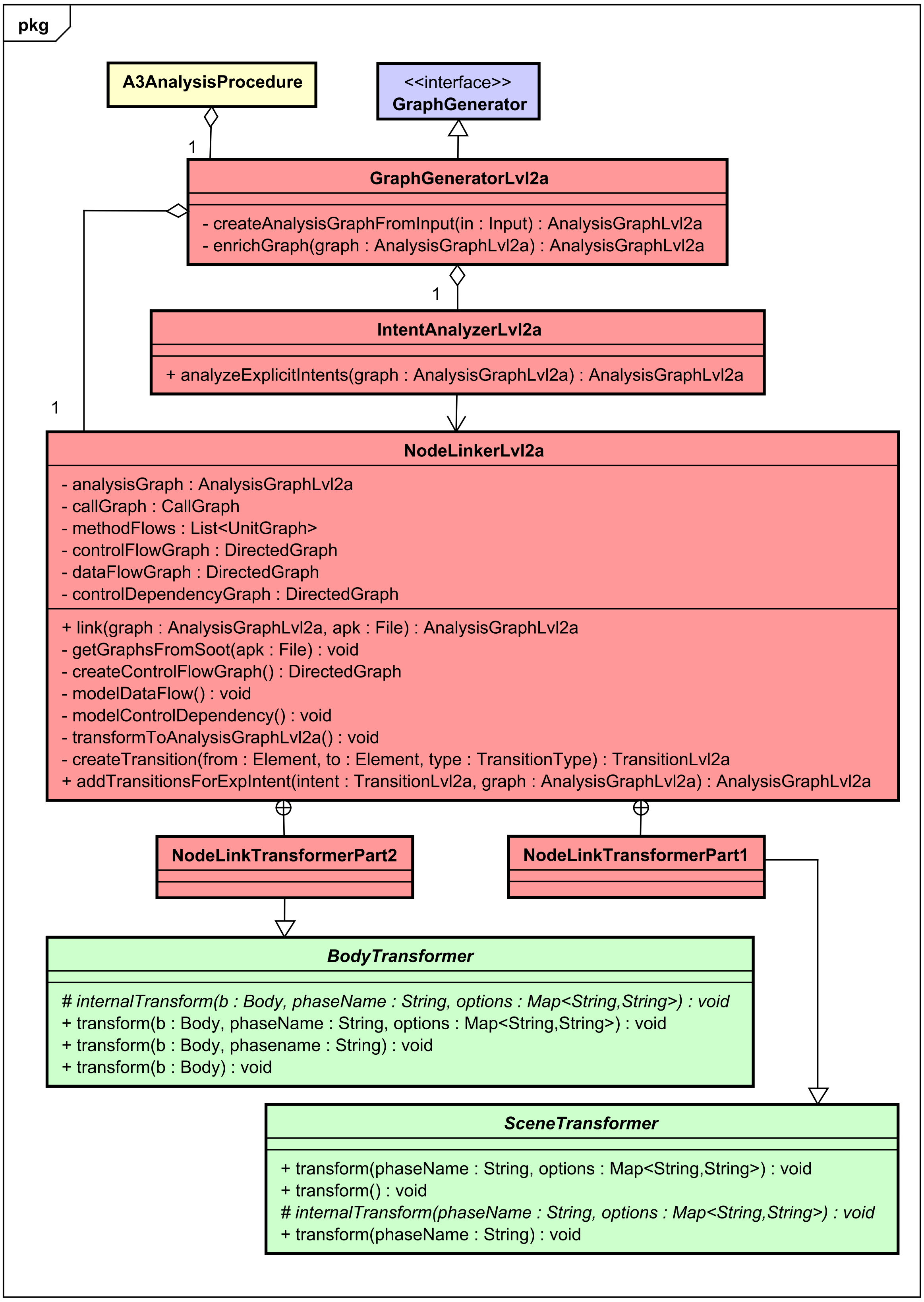

pkg
A3AnalysisProcedure
<<interface>>
GraphGenerator
1
GraphGeneratorLvl2a
- createAnalysisGraphFromInput(in : Input) : AnalysisGraphLvl2a
- enrichGraph(graph : AnalysisGraphLvl2a) : AnalysisGraphLvl2a
1
IntentAnalyzerLvl2a
+ analyzeExplicitIntents(graph : AnalysisGraphLvl2a) : AnalysisGraphLvl2a
1
NodeLinkerLvl2a
- analysisGraph : AnalysisGraphLvl2a
- callGraph : CallGraph
- methodFlows : List<UnitGraph>
- controlFlowGraph : DirectedGraph
- dataFlowGraph : DirectedGraph
- controlDependencyGraph : DirectedGraph
+ link(graph : AnalysisGraphLvl2a, apk : File) : AnalysisGraphLvl2a
- getGraphsFromSoot(apk : File) : void
- createControlFlowGraph() : DirectedGraph
- modelDataFlow() : void
- modelControlDependency() : void
- transformToAnalysisGraphLvl2a() : void
- createTransition(from : Element, to : Element, type : TransitionType) : TransitionLvl2a
+ addTransitionsForExplIntent(intent : TransitionLvl2a, graph : AnalysisGraphLvl2a) : AnalysisGraphLvl2a
NodeLinkTransformerPart2
NodeLinkTransformerPart1
BodyTransformer
# internalTransform(b : Body, phaseName : String, options : Map<String,String>) : void
+ transform(b : Body, phaseName : String, options : Map<String,String>) : void
+ transform(b : Body, phasename : String) : void
+ transform(b : Body) : void
SceneTransformer
+ transform(phaseName : String, options : Map<String,String>) : void
+ transform() : void
# internalTransform(phaseName : String, options : Map<String,String>) : void
+ transform(phaseName : String) : void

Figure 32: Class Diagram of the `GraphGeneratorLvl2a`

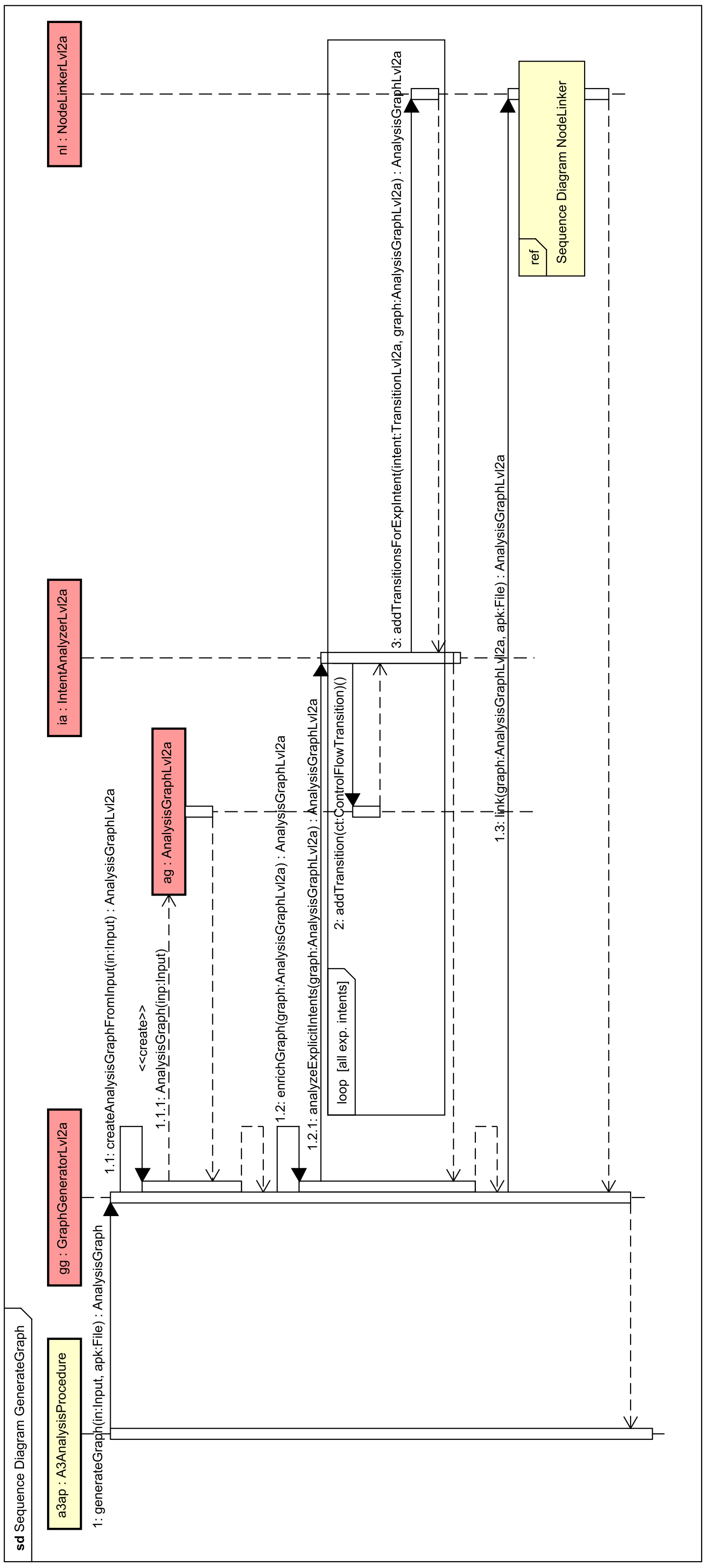

sd Sequence Diagram GenerateGraph
a3ap : A3AnalysisProcedure
gg : GraphGeneratorLvl2a
ag : AnalysisGraphLvl2a
ia : IntentAnalyzerLvl2a
nl : NodeLinkerLvl2a
1: generateGraph(in:Input, apk:File) : AnalysisGraph
1.1: createAnalysisGraphFromInput(in:Input) : AnalysisGraphLvl2a
<<create>>
1.1.1: AnalysisGraph(inp:Input)
1.2: enrichGraph(graph:AnalysisGraphLvl2a) : AnalysisGraphLvl2a
1.2.1: analyzeExplicitIntents(graph:AnalysisGraphLvl2a) : AnalysisGraphLvl2a
loop [all exp. intents]
2: addTransition(ct:ControlFlowTransition)()
3: addTransitionsForExpIntent(intent:TransitionLvl2a, graph:AnalysisGraphLvl2a) : AnalysisGraphLvl2a
1.3: link(graph:AnalysisGraphLvl2a, apk:File) : AnalysisGraphLvl2a
ref
Sequence Diagram NodeLinker

Figure 33: Sequence Diagram for the GraphGeneratorLvl2a

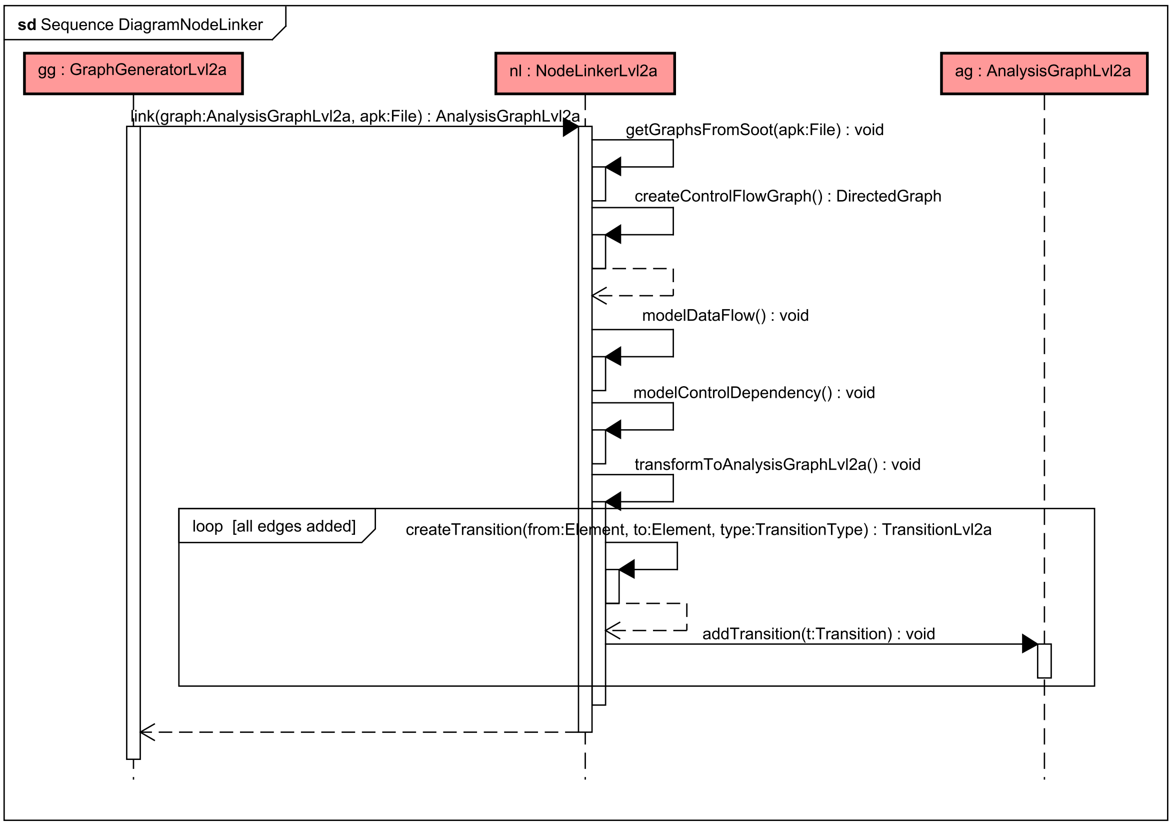

sd Sequence DiagramNodeLinker
gg : GraphGeneratorLvl2a
nl : NodeLinkerLvl2a
ag : AnalysisGraphLvl2a
link(graph:AnalysisGraphLvl2a, apk:File) : AnalysisGraphLvl2a
getGraphsFromSoot(apk:File) : void
createControlFlowGraph() : DirectedGraph
modelDataFlow() : void
modelControlDependency() : void
transformToAnalysisGraphLvl2a() : void
loop [all edges added]
createTransition(from:Element, to:Element, type:TransitionType) : TransitionLvl2a
addTransition(t:Transition) : void

Figure 34: Sequence Diagram for the `NodeLinkerLvl2a`

to add further transitions to the `AnalysisGraphLvl2a`, for example from the `Component` the intent originates from to the `Component` started by the intent. Such a transition is needed in detail level COMPONENT FLOW to model that a flow from the first to the second component exists. If the `IntentAnalyzerLvl2a` has analyzed all statements, the `AnalysisGraphLvl2a` enriched with `Transitions` describing flow concerning intents is returned to the `GraphGeneratorLvl2a`. This `AnalysisGraphLvl2a` is then forwarded to the `NodeLinkerLvl2a` via calling the method `link(...)`. In addition the `.apk` is passed as a parameter since the `NodeLinkerLvl2a` will call Soot and needs the `.apk` for that.

The behavior of the `NodeLinkerLvl2a` is shown in Figure 34. When `link(...)` is called it gets an `AnalysisGraphLvl2a` which then will be enriched with transitions and further elements. To do this the first step is to call Soot in the method `getGraphsFromSoot(...)`. The `NodeLinkerLvl2a` has two inner classes `NodeLinkTransformerPart1` and `NodeLinkTransformerPart2` which are extensions for the Soot classes `SceneTransformer` and `BodyTransformer`. These classes will instruct Soot to create a `CallGraph` for the `.apk` and a `UnitGraph` for every method representing the control flow in the method.

Since the `main()` method call from Soot has no return value, Soot will be instructed to store the created `UnitGraphs` and the `CallGraph` in class variables of the `NodeLinkerLvl2a` (see Fig. 32). The `Soot.main()` call invoked in the method `getGraphsFromSoot(...)` will therefore be responsible for initializing the class variables `callGraph` and `method-Flows`. In `getGraphsFromSoot(...)` dummy main methods will be created. For more information why this is needed and how this is done see Section 3.

The next step the `NodeLinkerLvl2a` has to execute is combining the `UnitGraphs` and the `CallGraph` to one `DirectedGraph` via the method `createControlFlow-Graph()`. Note that `UnitGraph`, `CallGraph` and `DirectedGraph` are Soot classes. After executing this step the `NodeLinkerLvl2a` has a graph representing the control flow. This graph has still the structure provided by Soot. Before it can be passed to the `Analyzer-Lvl2a` it has to be transformed into an `AnalysisGraphLvl2a`. But to perform a Level 2a analysis the data flow has to be modeled and since the `EnhancedInput` structure of the overall framework would have to be enriched with additional classes an attributes to provide the possibility to evaluate statements (e.g. no recognition of variables in a statement), we decided to model the data flow on top of the graph structure provided by Soot before transforming their graph structure into the A3 structure because Soots representation already provides the required structure. This means the `NodeLinkerLvl2a` will call `modelDataFlow()` as well as `modelControlDependency()` and the methods will model the corresponding flow on the directed graph. The newly modeled transitions are collected into a data structure and a source and target as `Unit` objects are saved for each of those transition. `Unit` is the type which is resembled in our data structure by the class `Element`(Fig. 15).

The data flow will be computed by doing a reaching definitions analysis whereas the control dependency will be modeled by first computing postdominance relations and based on this the control dependencies.

After modeling the dependencies described above, the directed graph with the newly added transitions will be transformed into an `AnalysisGraphLvl2a`. Therefore, the partially enriched `AnalysisGraphLvl2a` that was passed from the `GraphGeneratorLvl2a` to the `NodeLinkerLvl2a` has to be enriched by different types of transitions. This happens by mapping each node in the directed graph to a node in the `AnalysisGraphLvl2a` and transform the existing transitions in the directed graph to transitions in the `AnalysisGraphLvl2a`. Since different flows and dependencies have to be modeled, the class `Transition` will be extended by the class `TransitionLvl2a` (see Fig. 36) which in addition to the inherited attributes and methods has a `type` attribute. This enables the `AnalyzerLvl2a` to distinguish between the different sorts of transitions. The necessity of this distinction will be seen in Section 4.3.3. For every `Transition` in the directed graph the `NodeLinkerLvl2a` executes its method `createTransition(...)` and then adds the just created `Tran-sitionLvl2a` to the `AnalysisGraphLvl2a` via the method `addTransition(...)` provided by the interface `AnalysisGraph` (see Fig. 15).

In addition to the transitions and the already existing nodes between them, some additional nodes have to be added to the `AnalysisGraphLvl2a`. The first class of nodes are `EntryPoints` which were added as starting nodes to Soot in the method `getGraphs-FromSoot(...)`. The dummy main method will be added as an `EntryPoint` object to the `AnalysisGraphLvl2a`. Another additional node class are `Parameter` nodes. For every method call, four parameter nodes are added to the `AnalysisGraphLvl2a` via the method `addParameterElement(...)`. For every enumeration element of the enumeration `ParamType` (see Fig. 36) one `Parameter` is created. In addition to the control flow, data flow and the control dependency transitions CALL transitions from a method call `Statement` to the `Method` as well as SUMMARY `Transitions` between ACTUAL_IN and ACTUAL_-OUT nodes of one `Method` and PARAM transitions from ACTUAL_IN to FORMAL_IN and from FORMAL_OUT to ACTUAL_OUT will be added to the `AnalysisGraphLvl2a`. These additional nodes and transitions will be needed to perform context sensitive backward slicing. [11]

After the graph is transformed successfully the enriched `AnalysisGraphLvl2a` is returned to the `GraphGeneratorLvl2a` which in turn then forwards the graph to the `A3AnalysisProcedure`.

4.3.3 AnalyzerLvl2a

The `A3AnalysisProcedure` gets an `AnalysisGraphLvl2a` that is added as parameter to the method `analyze(...)` called on the `AnalyzerLvl2a`. The method is provided since the `AnalyzerLvl2a` implements the interface `Analyzer` (see Fig. 12). Up to the point where the `A3AnalysisProcedure` calls `analyze(...)` on the `Analyzer-Lvl2a` the level 2a specific workflow in SUMMARY and in COMPARISON is exactly the same. But when instructing the `AnalyzerLvl2a` to execute the method `analyze(...)` and COMPARISON mode is selected a previous result is passed in addition to the `Analy-sisGraph`. If the previous result is passed, the `AnalyzerLvl2a` knows that is has to do a comparison after finishing the analysis of the graph. But nevertheless the graph has to be analysed as first step in both modes.

The structure of the analysis done by the `AnalyzerLvl2a` is shown in Figure 37. The first step will be the computation of sources and sinks. This is done in the `SourceAndSinkCom-puter`. As described in the Requirement Specification we will identify two types of sources. They can be identified by looking at the transition types of incoming or outgoing transitions of the statements in the `AnalysisGraphLvl2a`. If a statement has a call transition to a method and that method requires a permission, it is a source. Note that in this case the `SourceAndSinkComputer` has to identify the transition as call transition via analyzing

[11]according to Hammer: Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs, Universit´at Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3

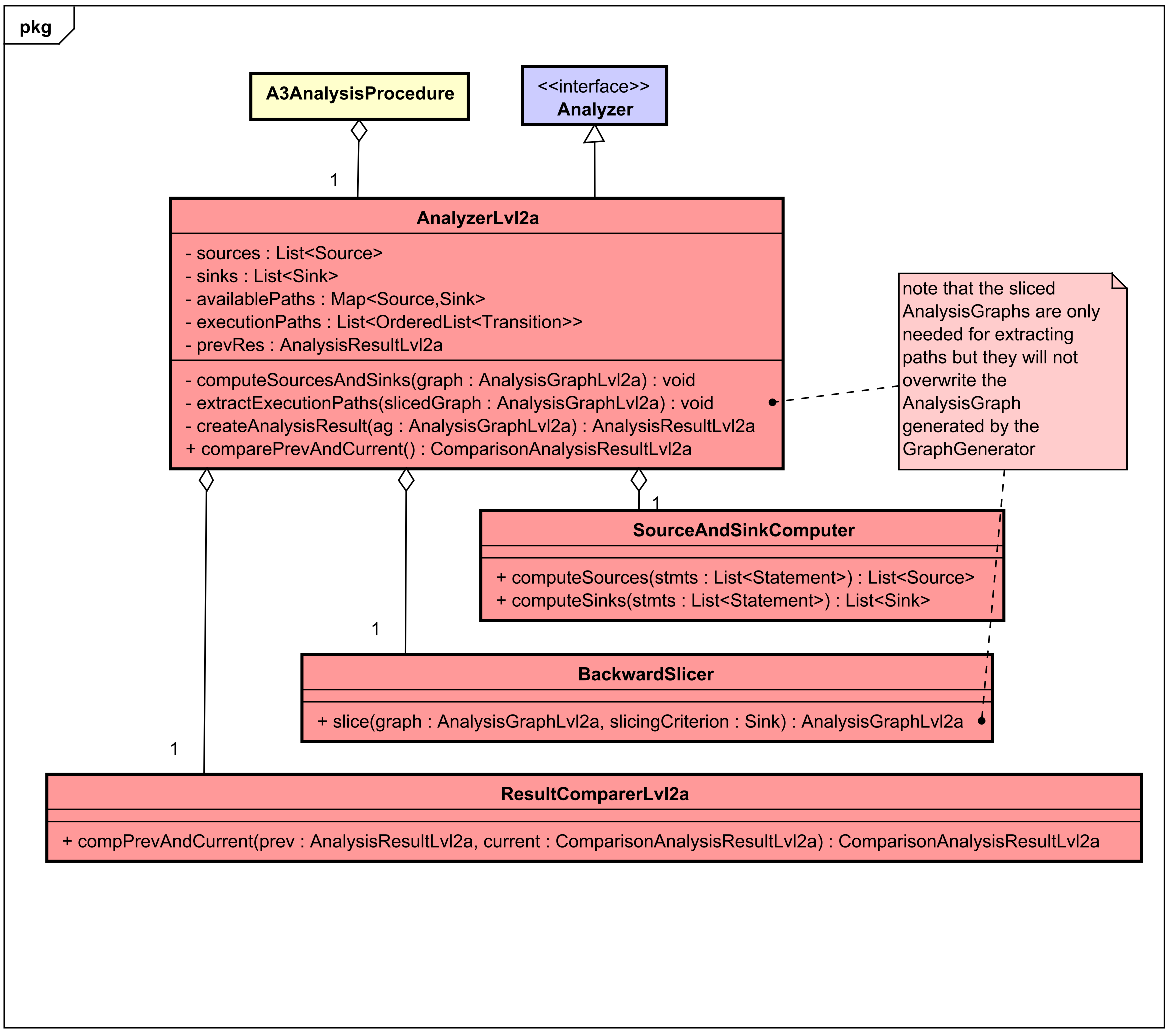

pkg
A3AnalysisProcedure
<<interface>>
Analyzer
1
AnalyzerLvl2a
- sources : List<Source>
- sinks : List<Sink>
- availablePaths : Map<Source,Sink>
- executionPaths : List<OrderedList<Transition>>
- prevRes : AnalysisResultLvl2a
- computeSourcesAndSinks(graph : AnalysisGraphLvl2a) : void
- extractExecutionPaths(slicedGraph : AnalysisGraphLvl2a) : void
- createAnalysisResult(ag : AnalysisGraphLvl2a) : AnalysisResultLvl2a
+ comparePrevAndCurrent() : ComparisonAnalysisResultLvl2a
note that the sliced AnalysisGraphs are only needed for extracting paths but they will not overwrite the AnalysisGraph generated by the GraphGenerator
1
SourceAndSinkComputer
+ computeSources(stmts : List<Statement>) : List<Source>
+ computeSinks(stmts : List<Statement>) : List<Sink>
1
BackwardSlicer
+ slice(graph : AnalysisGraphLvl2a, slicingCriterion : Sink) : AnalysisGraphLvl2a
1
ResultComparerLvl2a
+ compPrevAndCurrent(prev : AnalysisResultLvl2a, current : ComparisonAnalysisResultLvl2a) : ComparisonAnalysisResultLvl2a

Figure 35: Class Diagram of the `AnalyzerLvl2a`

the type attribute of the `TransitionLvl2a` but then has to check whether the `Statement` has a `Permission` since the information whether a method needs a permission will be stored in the `Statements` that are method calls (for details see Sec. 4.1.4). Analogously to this behaviour a return result of another component can be detected as source.

As sink we detect only library method calls where information is passed via parameters. This can be detected by analysing whether there is any data flow transition to such a method call, because this indicates that information flows via parameters to that method call.

The methods `computeSources(...)` and `computeSinks(...)` will do the just described analysis on the list of all statements passed to them as parameter and return a subset of those statements as list of `Sinks` respectively list of `Sources`. In Figure 36 is shown that `Source` and `Sink` are classes representing special `Statements`. The returned lists will be stored in class variables of the `AnalyzerLvl2a`, since this is some of the information that will be used to create the `AnalysisResultLvl2a`.

After `computeSourceAndSinks(...)` has finished, the `BackwardSlicer` will be called by the `AnalyzerLvl2a` to perform the backward slicing algorithm. One such backward slice is computed for every of the detected sources, since for every of the sources all possible flows from any sink to that source have to be detected.

The backward slice for a sink is computed mainly in two phases accordingly to the context-sensitive approach described by Hammer. [12] This approach is a kind of reachability problem for graphs and can be executed in time O(size of graph). The first phase is executed on the `AnalysisGraphLvl2a` and a `Sink` as slicing criterion. It traverses all transitions of type DATAFLOW, CONTROLDEPENDENCY, CALL, SUMMARY and those PARAM transitions going from a `Parameter` of type FORMAL_OUT to a `Parameter` of type ACTUAL_OUT. This includes all flows in the method of the slicing criterion as well as those flows from methods calling this method.

The second phase starts from the list of `Parameter` nodes that were omitted in the first phase and ignores CALL, CONTROLFLOW and PARAM transitions going from a ACTUAL_-IN to a FORMAL_IN `Parameter`. All other transitions will be taken into account and a subset of reachable nodes and transitions between them will be computed.

After executing both phases the method `slice(...)` computes the union of the two phases by using the two sliced `AnalysisGraphLvl2as` and computes the union of the two slices. This means the union of the set of `Elements` of both graphs as well as the union of the sets of `Tranistions` is calculated and a new `AnalysisGraph` consisting of those `Elements` is created. This united graph is then returned to the `AnalyzerLvl2a` who can analyze in the method `extractExecutionPaths(...)` from which source to the sink, that was slicing criterion for that sliced graph, a data flow path exists. If paths are found, they are added to the `Analyzer`'s attribute `executionPaths` and the corresponding source and

[12]Hammer: Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs, Universit́at Karlsruhe (TH), Fak. f. Informatik, July 2009. ISBN 978-3-86644-398-3

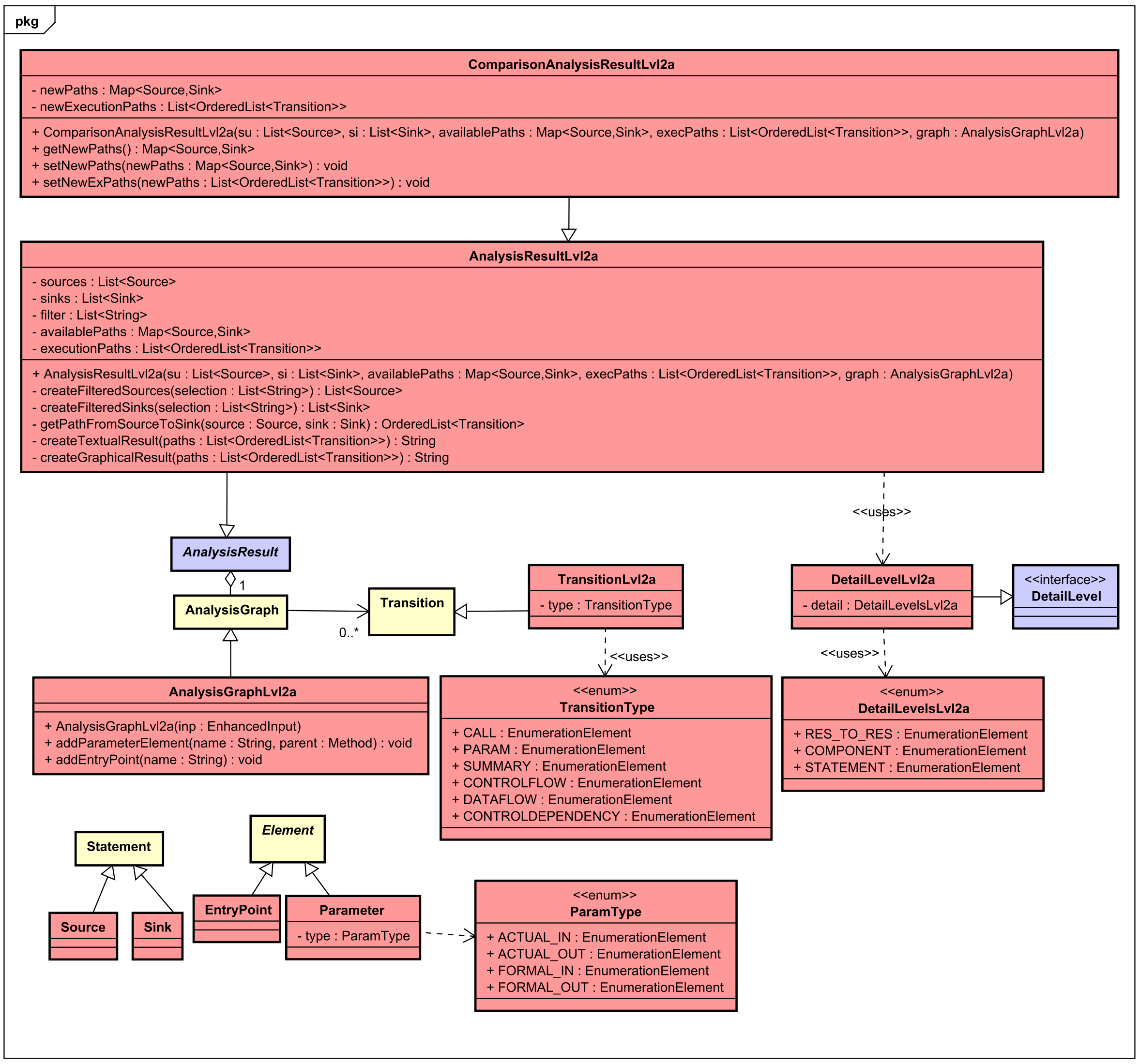

pkg
ComparisonAnalysisResultLvl2a
- newPaths : Map<Source,Sink>
- newExecutionPaths : List<OrderedList<Transition>>
+ ComparisonAnalysisResultLvl2a(su : List<Source>, si : List<Sink>, availablePaths : Map<Source,Sink>, execPaths : List<OrderedList<Transition>>, graph : AnalysisGraphLvl2a)
+ getNewPaths() : Map<Source,Sink>
+ setNewPaths(newPaths : Map<Source,Sink>) : void
+ setNewExPaths(newPaths : List<OrderedList<Transition>>) : void
AnalysisResultLvl2a
- sources : List<Source>
- sinks : List<Sink>
- filter : List<String>
- availablePaths : Map<Source,Sink>
- executionPaths : List<OrderedList<Transition>>
+ AnalysisResultLvl2a(su : List<Source>, si : List<Sink>, availablePaths : Map<Source,Sink>, execPaths : List<OrderedList<Transition>>, graph : AnalysisGraphLvl2a)
- createFilteredSources(selection : List<String>) : List<Source>
- createFilteredSinks(selection : List<String>) : List<Sink>
- getPathFromSourceToSink(source : Source, sink : Sink) : OrderedList<Transition>
- createTextualResult(paths : List<OrderedList<Transition>>) : String
- createGraphicalResult(paths : List<OrderedList<Transition>>) : String
<<uses>>
AnalysisResult
1
AnalysisGraph
0..*
Transition
TransitionLvl2a
- type : TransitionType
DetailLevelLvl2a
- detail : DetailLevelsLvl2a
<<interface>>
DetailLevel
<<uses>>
<<uses>>
AnalysisGraphLvl2a
+ AnalysisGraphLvl2a(inp : EnhancedInput)
+ addParameterElement(name : String, parent : Method) : void
+ addEntryPoint(name : String) : void
<<enum>>
TransitionType
+ CALL : EnumerationElement
+ PARAM : EnumerationElement
+ SUMMARY : EnumerationElement
+ CONTROLFLOW : EnumerationElement
+ DATAFLOW : EnumerationElement
+ CONTROLDEPENDENCY : EnumerationElement
<<enum>>
DetailLevelsLvl2a
+ RES_TO_RES : EnumerationElement
+ COMPONENT : EnumerationElement
+ STATEMENT : EnumerationElement
Statement
Element
Source
Sink
EntryPoint
Parameter
- type : ParamType
<<enum>>
ParamType
+ ACTUAL_IN : EnumerationElement
+ ACTUAL_OUT : EnumerationElement
+ FORMAL_IN : EnumerationElement
+ FORMAL_OUT : EnumerationElement

Figure 36: Extended Data Structure for Level 2a

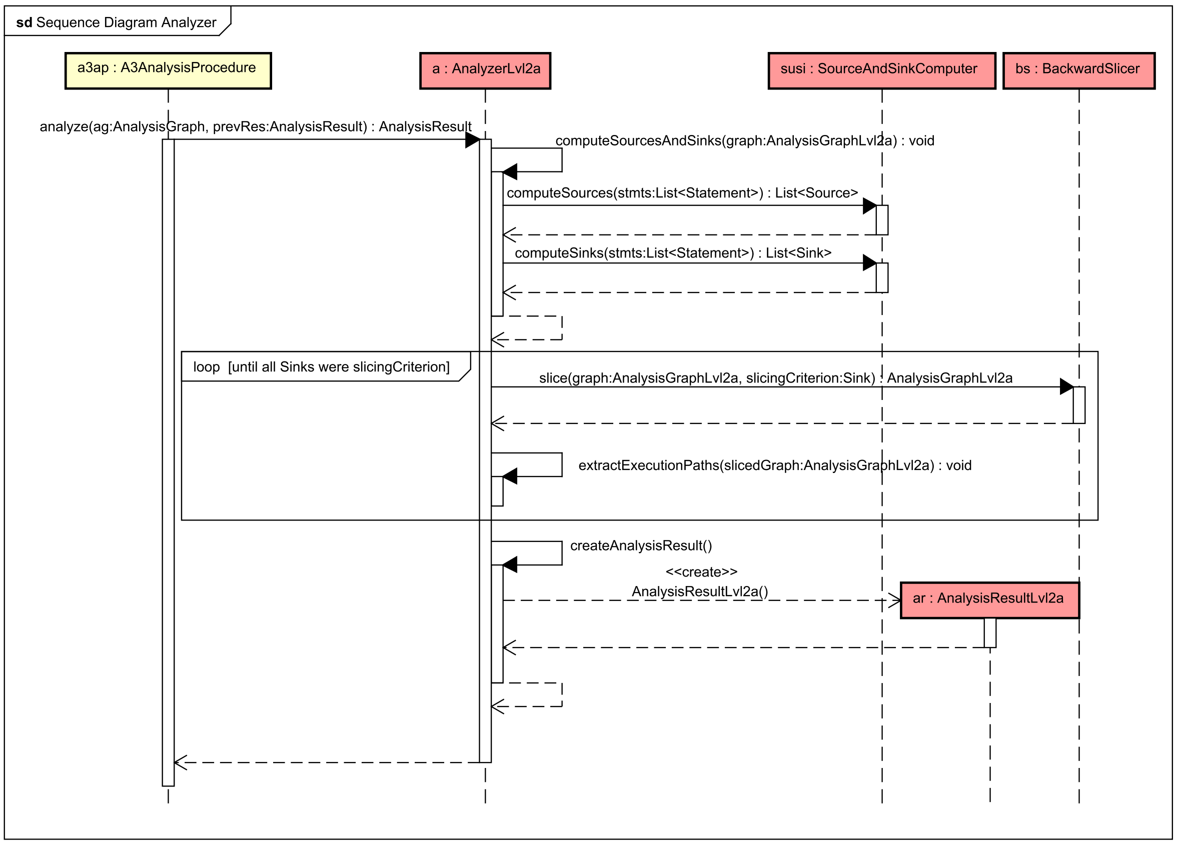

sd Sequence Diagram Analyzer
a3ap : A3AnalysisProcedure
a : AnalyzerLvl2a
susi : SourceAndSinkComputer
bs : BackwardSlicer
analyze(ag:AnalysisGraph, prevRes:AnalysisResult) : AnalysisResult
computeSourcesAndSinks(graph:AnalysisGraphLvl2a) : void
computeSources(stmts:List<Statement>) : List<Source>
computeSinks(stmts:List<Statement>) : List<Sink>
loop [until all Sinks were slicingCriterion]
slice(graph:AnalysisGraphLvl2a, slicingCriterion:Sink) : AnalysisGraphLvl2a
extractExecutionPaths(slicedGraph:AnalysisGraphLvl2a) : void
createAnalysisResult()
<<create>>
AnalysisResultLvl2a()
ar : AnalysisResultLvl2a

Figure 37: Sequence Diagram for the `AnalyzerLvl2a`

sink are stored in the map `availablePaths` which collects from which source to which sink a flow exists.

If all sinks were slice, the analysis part for SUMMARY mode is finished and an `AnalysisResult` can be created by invoking the method `createAnalysisResult(...)` which then uses the class attributes of the `AnalyzerLvl2a` to create an object of type `AnalysisResultLvl2a`. This behaviour is shown in Figure 37. If COMPARISON mode is selected, which means the parameter `prevRes` was not `null`, `createAnalysisResult(...)` creates a result object of type `ComparisonAnalysisResultLvl2a` which is a special type of `AnalysisResultLvl2a` as visualized in Figure 36. In addition to the inherited methods and attributes of the `AnalysisResultLvl2a` this type has a list of new paths from source to sink that are in the newly analyzed `.apk` but have not been in the previous result. To fill this list the `AnalyzerLvl2a` has to execute an additional step after creating an `AnalysisResult` if he is in COMPARISON mode. For this case the class `ResultComparer` exists (see Fig. 35) and provides the method `compPrevAndCurrent(...)` which gets the previous analysis result and the just created one and compares which paths are new. Those are then stored in the class attribute of the `ComparisonAnalysisResultLvl2a` via the setter (see Fig. 36).

Finally, depending on the mode the `ComparisonAnalysisResultLvl2a` respectively the `AnalysisResultLvl2a` is returned to the `A3AnalysisProcedure`.

If during creation of the `AnalysisGraphLvl2a` or the `AnalysisResultLvl2a` one of the cases is observed, which produce a warning or an error according to the Target Level Agreement these `Messages` are added to the `AnalysisResultLvl2a` using the `Message` class of the overall framework (see Sec. 4.1.2).

4.3.4 AnalysisResultLvl2a

The last point which is specific for Level 2a and has to be mentioned in this section is the way the methods `getTextualResult(...)` and `getGraphicalResult(...)` of the abstract class `AnalysisResult` (see Fig. 15) are realized in the Level 2a specific class `AnalysisResultLvl2a`.

The `Client` can request the filters as a list of Strings. For Level 2a the user can filter the result output to take only a subset of the found sources and sinks into account. Therefore, the method `getFilters()` of the `AnalysisResultLvl2a` will return a list of Strings, where each String begins with the keyword 'source' or 'sink' plus the name String of that source or sink. This list is stored in the class `AnalysisResultLvl2a` as attribute `filter` to avoid that the list has to be constructed every time `getFilters()` is called. Instead it will be initialized when constructing the `AnalysisResultLvl2a` object.

When the client wants to display the result he can call `getTextualReult(...)` or `getGraphiclResult(...)` passing as parameters a detail level of type `DetailLevel` which is an interface provided by the framework (see Fig. 15) and a list of Strings as filter. The methods are both provided by the abstract class `AnalysisResult` (see Fig. 15). If one of the methods is called on the `AnalysisResultLvl2a` the detail level will be an element of type `DetailLevelLvl2a`. This class has one attribute `detail` which is an element of the enumeration *DetailLevelsLvl2a* consisting of the three elements RES_TO_RES, COMPONENT and STATEMENT which resemble the three possible detail levels described in the Target Level Agreement. These detail level classes are displayed in Figure 36.

An exemplary behavior for the method `getTextualResult(...)` describes Figure 38. Except for the last method call the strategy for `getGraphicalResult(...)` is equivalent. The first step is to retrieve the selected sources and sinks represented by the list of Strings. Therefore, the method `createFilteredSources(...)` has to map every String starting with the substring 'source' back to a source which then are stored in a local variable of the method `getTextualResult(...)/getGraphicalResult(...)`. `createFilteredSinks(...)` behaves equally to the just explained method just looking for sinks instead of sources.

The next step is to create a subset of the list of paths from sources to sinks which was stored in the `AnalysisResultLvl2a` by the `Analyzer`. Therefore, for every combination of the

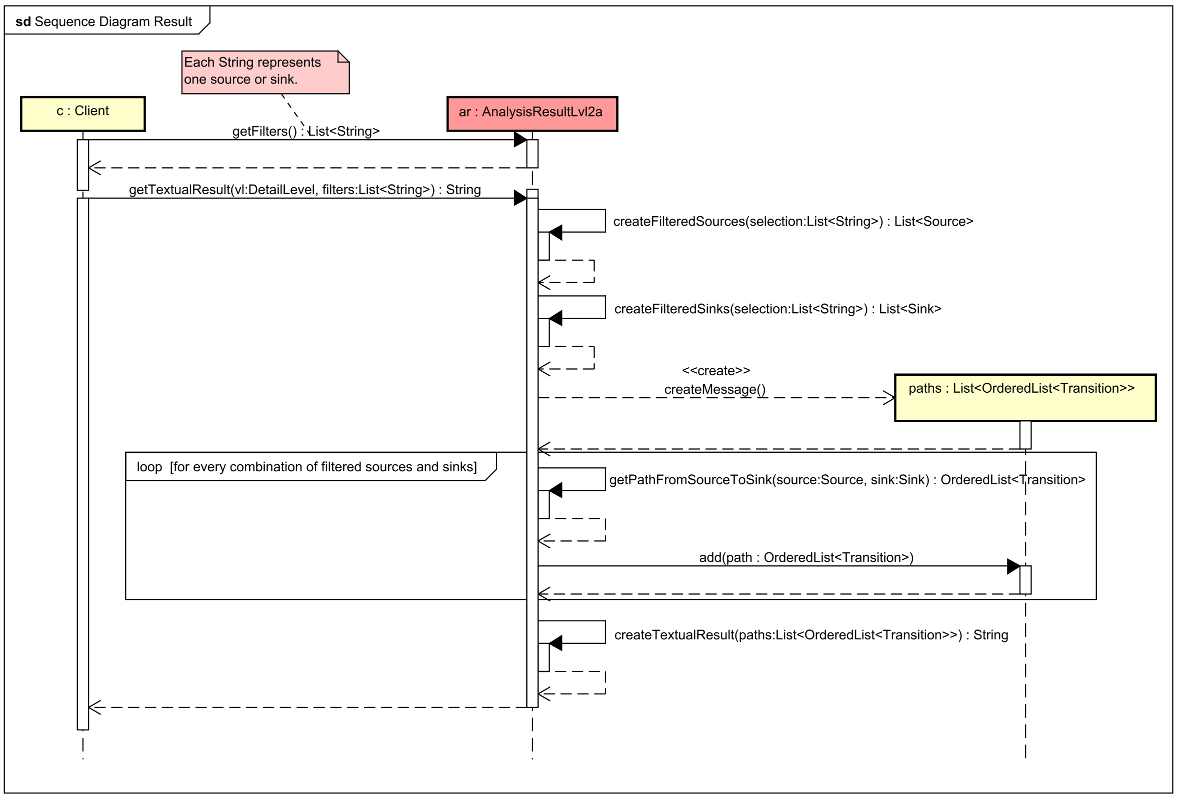

sd Sequence Diagram Result
Each String represents one source or sink.
c : Client
ar : AnalysisResultLvl2a
getFilters() : List<String>
getTextualResult(vl:DetailLevel, filters:List<String>) : String
createFilteredSources(selection:List<String>) : List<Source>
createFilteredSinks(selection:List<String>) : List<Sink>
<<create>>
createMessage()
paths : List<OrderedList<Transition>>
loop [for every combination of filtered sources and sinks]
getPathFromSourceToSink(source:Source, sink:Sink) : OrderedList<Transition>
add(path : OrderedList<Transition>)
createTextualResult(paths:List<OrderedList<Transition>>) : String

Figure 38: Class Diagram of the `AnalysisResultLvl2a`

filtered subsets of sources and sinks the method `getPathFromSourceToSink(...)` can be called to get the paths or `null` if no path exists. An alternative way of implementing is to create a map from source to sink which is a subset of the attribute *availablePaths* of the `AnalysisResultLvl2a` and then call `getPathFromSourceToSink(...)` only for the existing paths. However, if the subset of paths corresponding to the filter is created, the method `createTextualResult(...)` respectively `createGraphicalResult(...)` is called which then will transform the paths into the result String provided in HTML5 respectively the Graphviz DOT language (see Sec. 4.1.2).

4.4 AnalysisProcedure for Level 2b

In this section the class diagram for Level 2b is described in detailed and we will describe the behaviour of the classes with a sequence diagram. The class diagram (see Figure 39) is Level 2b specific, so only the important classes which are used in Level 2b are mentioned in the class diagram. Level 2b is an extension of Level 1 and in Level 2b the analysis is done for more than one App. First we perform the Level 1 analyses for all the input apks individually, which can be seen in the Level 1 description. Then with the collection of analysis results our tool performs Level 2b analysis.

4.4.1 AnalysisFactoryLvl2b

`AalysisFactory` works the same way as the other two levels. The `AnalysisFactoryLvl2b` implements an interface called `AnalysisFactory`. The `AnalysisFactoryLvl2b` has a constructor, the argument that are passed by the constructor are input `apk(s)` and non-native `apk(s)` (which are only supporting `apk(s)`) in case of SUMMARY mode and input `apk(s)`, non-native `apk(s)` and a previous `AnalysisResult` in case of COMPARISON mode. The abstract method `CreateAnalysis()` creates the `Enhancer`, GraphGeneratorLvl2b and Analyzerlvl2b which builds the `AnalysisProcedure`. The `CreateAnalysis()` creates the `Analysis` for SUMMARY or COMPARISON mode depending on the number of parameter passed.

4.4.2 A3AnalysisProcedure

The `A3AnalysisProcedure` class implements the `AnalysisProcedure` interface which has an aggregation relation with three classes which are `Enhancer, GraphGeneratorLvl2b, Analyzerlvl2b`. The function of the each and every class will be defined in detail bellow.

4.4.3 Enhancer

The `Enhancer` class working principle is already described above which also common for Level 2b. In addition to that in Level 2b the `Enhancer` uses the function `CollectResult(...)` for the first time to handle the collection of results. This method just passes the list of results from the `AnalysisRunner` to the `GraphGenerator` component.

4.4.4 GraphGeneratorLvl2b

The `GraphGeneratorLvl2b` class implements the interface class called `GraphGenerator` which has a method `generateGraph(...)` which passes the `ResultInput` object which it got from the `enhancer` and returns the value as an `Analysisgraph`. The `GraphGeneratorLvl2b` class also has aggregation relation with two other level specific classes `IntentAnalyzerLvl2b` and `NodeLinkerLvl2b`. Now the `GraphGenerator` component is called to build a graph for the collection of results which is given as input. Since we already seen, that the `GraphGenerator` work for a single `.apk` in Level 1, here we will see how the graph for the collection of results is generated. For this the `IntentAnalyzer` and

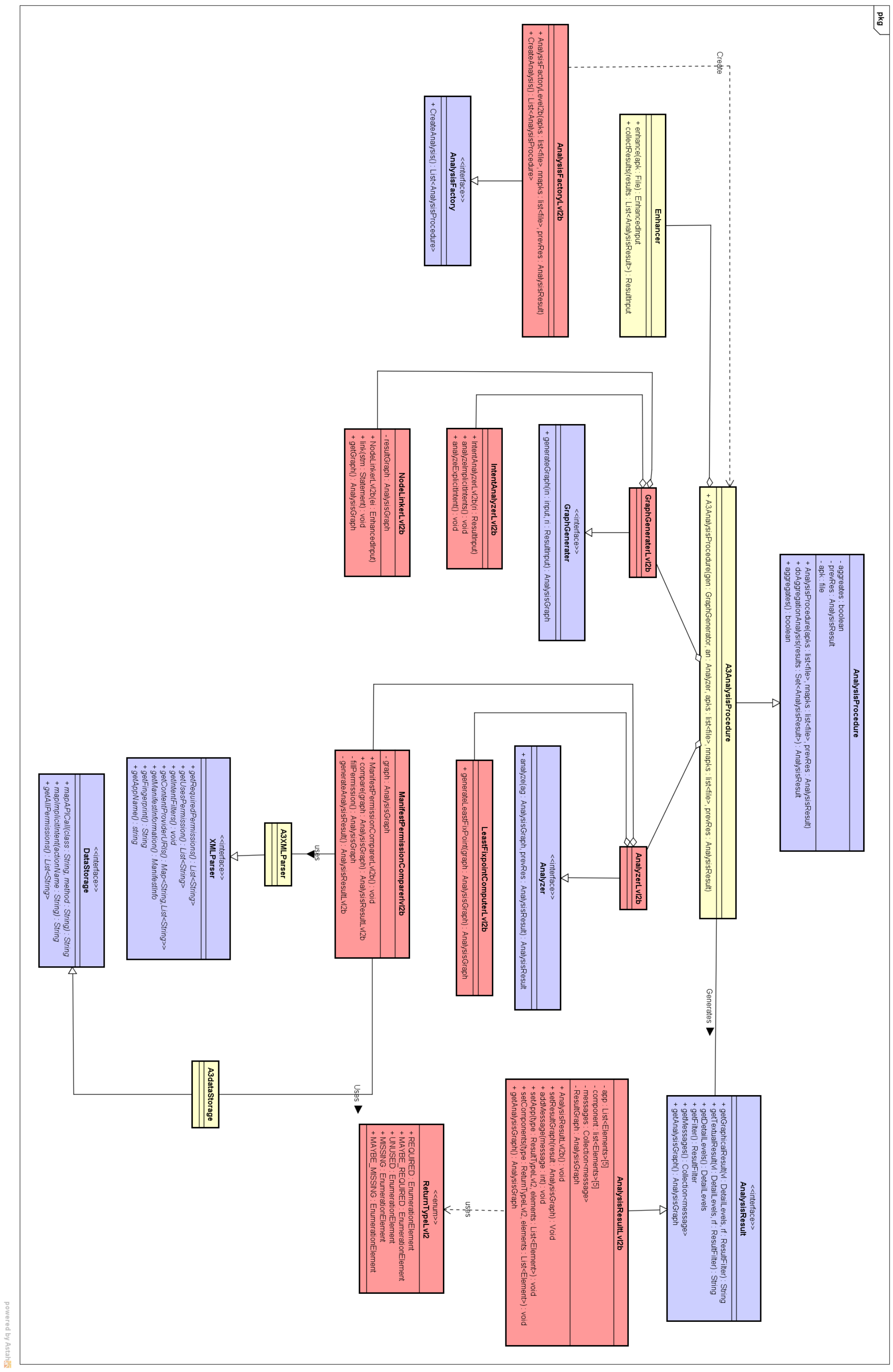


Figure 39: Class Diagram for Level 2b

`NodeLinker` which are the subcomponents of the `GraphGenerator` communicate arbitrary between them to analysis the intent and build a graph. The `IntentAnalyzerLvl2b` class has a constructor which passes `ResultInput` as argument. In Level 2b the `IntentAnalyzer` component is important to analyze implicit intent, to track the communication between two or more Apps. The function of the `IntentAnalyzer` varies between different modes.

- SUMMARY,APP and COMPARISON, APP modes the `IntentAnalyzer` considers first `apk` as starting point.
- SUMMARY,ALL and COMPARISON, ALL modes the `IntentAnalyzer` considers all `apks` as starting point.

The level 2b specific class `IntentAnalyzerLvl2b` has a method called `analyzeImplicitIntents()` which tracks the implicit intent in the `ResultInput`. The `analyzeImplicitIntents()` method analysis all the result statement in the `ResultInput` and if it comes across an implicit intent especially in MAYBE_MISSING group in the Level 1 analysis result and if the `IntentAnalyzerLvl2b` finds a suitable statement in the result of any other Apps, then the link method in the `NodeLinkerLvl2b` class is called, which links the implicit intent between the statement and the respected components and adds the transition to the `AnalysisGraph`. The functioning of `GraphGenerator` class can be seen in Figure 40. If the `IntentAnalyzerlvl2b` analysed all the statements for intents the `NodeLinkerlvl2b` links all intents between the statements and an `AnalysisGraph` is generated which the `GraphGenerater` finally returns.

4.4.5 AnalyzerLvl2b

The `AnalyzerLvl2b` class implements a interface called `Analyzer` which passes the `AnalysisGraph` as input object in case of SUMMARY mode. In COMPARISON mode the `Analyzer` passes the `AnalysisGraph` and a previous `AnalysisResult` as input objects. The `AnalyzerLvl2b` class also has an aggregation relation with two classes which are the `LeastFixpointComputerLvl2b` and the `ManifestPermissionComparer`. `LeastFixpointComputerLvl2b` class has a function called `generateLeastFixpoint(...)` which passes and returns an `AnalysisGraph`. The `LeastFixpointComputer` component is specific used in Level 2b, since Level 2b will analyze both explicit and implicit intent for more then one App, the flow of intent should be computed until a least fix point. During the `FixedpointComputer` results of are aggregated per App and they are analyzed to find out the resource usages through implicit intent. The `LeastFixpointComputerLvl2b` determines all components reachability from a component in the `AnalysisGraph`. The `ManifestPermissionComparer` component works in a similar to one in Level 1. The component compares the permissions used in Android manifest file with the permissions used in the statements of the source code and the permissions required by the components, for this the `ManifestPermissionComparer` class uses a list

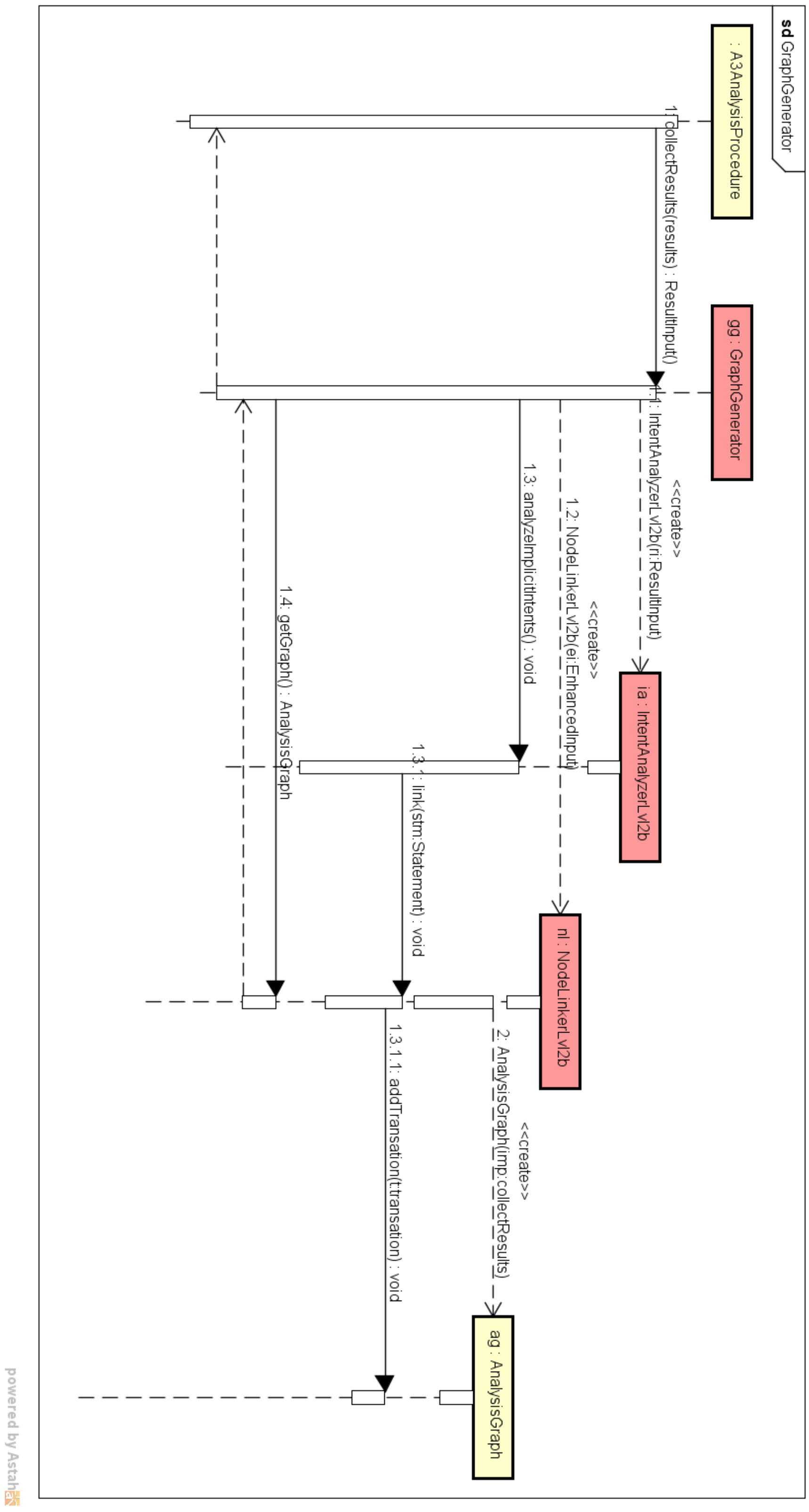

sd GraphGenerator
: A3AnalysisProcedure
gg : GraphGenerator
ia : IntentAnalyzerLvl2b
nl : NodeLinkerLvl2b
ag : AnalysisGraph
1: collectResults(results) : ResultInput()
<<create>>
1.1: IntentAnalyzerLvl2b(ri:ResultInput)
<<create>>
1.2: NodeLinkerLvl2b(ei:EnhancedInput)
1.3: analyzeImplicitIntents() : void
1.3.1: link(stm:Statement) : void
1.3.1.1: addTransation(t:transation) : void
<<create>>
2: AnalysisGraph(imp:collectResults)
1.4: getGraph() : AnalysisGraph
powered by Astah

Figure 40: Sequence Diagram for GraphGeneratorLevel2b

of permission that is obtained from the interface XMLParser including the intent filters in case of Level 2b and all the permissions from the DataStorage(seen Figure 41). Then the fillPermission() method is called to fill the permissions to the node in the Analysis-Graph. Here the tool differentiate which COMPONENT of which APP uses which resource. The generation of the results and the categorization of group of permissions are handled by the method generateAnalysisResult().

4.4.6 AnalysisResultLvl2b

Level 2b displays all the errors and warnings which are found during the Level 1 analyses except for those which are caused by Implicit intents. Apart from those if a component uses a permission which is not defined or a specific resource is unused then a warning is listed resource usage undefined and resource is unused respectively. If there is no matching intent filter it is listed as warning message. The results are displayed only for APP and COMPONENT level in which the element occurs.

4.5 User Interface Structure

This section provides information about the structure of the UI related classes of the Android App Analysis tool. Apart from the detailed UI class diagram description, the content of this section will be enhanced further by explaining about the behaviour of the classes too, with the help of the UI sequence diagram.

With respect to the UI Class diagram (see Figure 42), the important classes which will be covered as a part of this section are:

- ClientGUI
- ClientCommandLine
- GUI
- CommandLine
- GraphicalViewCreator
- TextualViewCreator

Other than the above mentioned classes, the structure and the methods of the necessary classes will be described wherever necessary. ClientGUI and ClientCommandLine classes are extended from the Client abstract class. So the common non-abstract functions and the attributes of the client classes are available in the Client abstract class. So the first subsection will introduce the Client abstract class structure and its methods and then the further subsections will describe the structure of the remaining important classes.

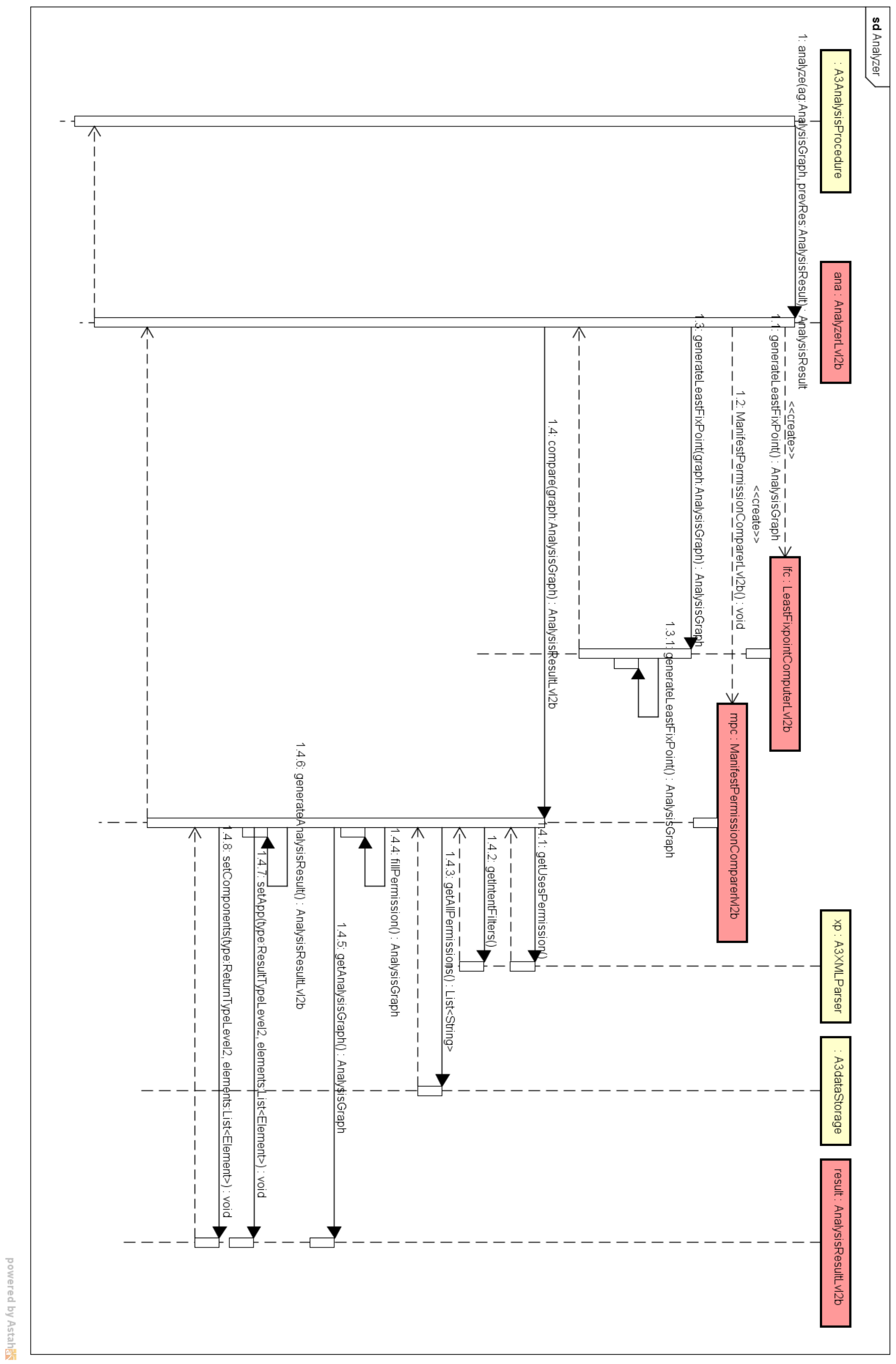

sd Analyzer
: A3AnalysisProcedure
ana : AnalyzerLvl2b
lfc : LeastFixpointComputerLvl2b
mpc : ManifestPermissionComparerlvl2b
xp : A3XMLParser
: A3dataStorage
result : AnalysisResultLvl2b
1: analyze(ag:AnalysisGraph, prevRes:AnalysisResult) : AnalysisResult
<<create>>
1.1: generateLeastFixPoint() : AnalysisGraph
<<create>>
1.2: ManifestPermissionComparerLvl2b() : void
1.3: generateLeastFixPoint(graph:AnalysisGraph) : AnalysisGraph
1.3.1: generateLeastFixPoint() : AnalysisGraph
1.4: compare(graph:AnalysisGraph) : AnalysisResultLvl2b
1.4.1: getUsesPermission()
1.4.2: getIntentFilters()
1.4.3: getAllPermissions() : List<String>
1.4.4: fillPermission() : AnalysisGraph
1.4.5: getAnalysisGraph() : AnalysisGraph
1.4.6: generateAnalysisResult() : AnalysisResultLvl2b
1.4.7: setApp(type:ResultTypeLevel2, elements:List<Element>) : void
1.4.8: setComponents(type:ReturnTypeLevel2, elements:List<Element>) : void
powered by Astah

Figure 41: Sequence Diagram for AnalyzerLevel2b

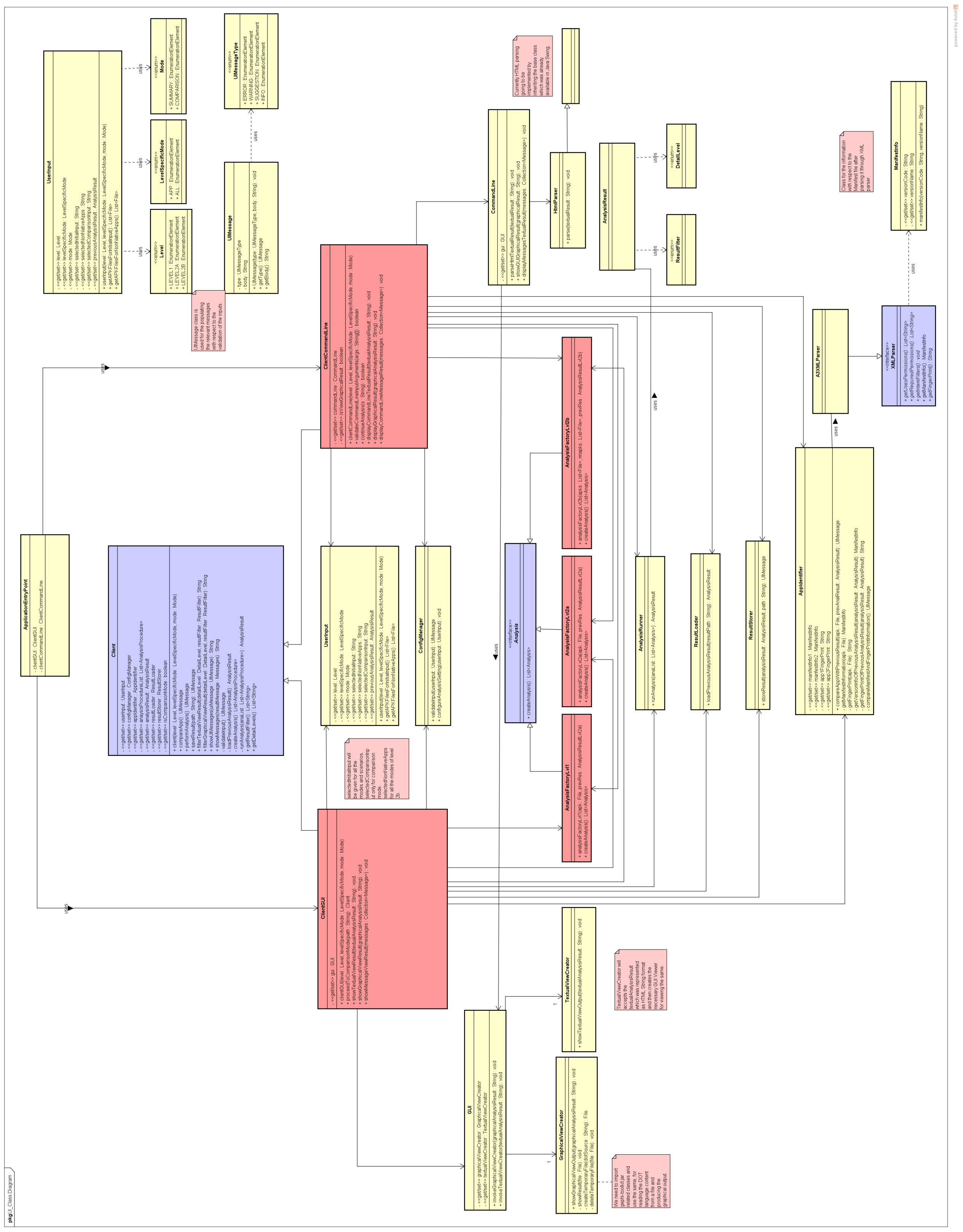


Figure 42: Class Diagram for User Interface

ApplicationEntryPoint From the class diagram, the entry point for starting the application in both the GUI and CommandLine mode by the user is done through a single entry point class named `ApplicationEntryPoint`. This class decides whether the application was triggered using the command line or GUI. Based on that, this Class will call the Business Logic classes named `ClientGUI` or `ClientCommandLine`. The instance of `ClientGUI` or `ClientCommandLine` will be created with the help of their related constructors and will be set in the respective attributes named `clientGUI` and `clientCommandLine` in the `ApplicationEntryPoint` class.

4.5.1 Client

As mentioned in the previous section, this `Client` abstract class serves the common attributes and the non-abstract methods, which will be useful for extending the client classes. This class provides a way to use the common attributes to access the common dependent classes such as `UserInput`, `ConfigManager`, `AppIdentifier`, `ResultLoader` and `ResultStorer`.

Consider this sequence diagram (see Figure 43), which actually represents how the input from the user has been processed in the tool with different interactions between the above mentioned UI classes as well as the other important classes to retrieve the analysis result. From the sequence diagram, as explained earlier, user will provide the input through user interface or command line with the help of `ApplicationEntryPoint` class which in turn will create the instance of the respective Business Logic class using its constructor. As both of those classes extend this abstract class `Client`, its constructor will get called too. One important point we need to consider here is when the user uses our application through GUI, the Business Logic class object from the entry point class will be created only when the user starts analyzing or comparing the application. From command line perspective, all the inputs as well as the option for doing the analysis will be already given by the user when they start the application by calling the entry point class.

UserInput This class is created along with the necessary attributes by setting the selected `Level`, `LevelSpecificMode` and `Mode`. From the class diagram perspective all the above mentioned level and mode related inputs are considered as enumeration which contains set of pre-defined constants. Enumeration `Level` consists of values such as LEVEL1, LEVEL2A and LEVEL2B. Similarly `MODE` consists of values such as SUMMARY and COMPARISON. `LevelSpecificMode` enumeration contain APP and ALL as its values which will be specific to Level 2b. Generally, attributes present in this `UserInput` class is of private field and it can be accessed outside this class through their setter and getter methods. Some of the attributes in this `UserInput` class represent the file location of the `.apk` and previous analysis result. There are important functions as well as one attribute in this class which will be discussed later.

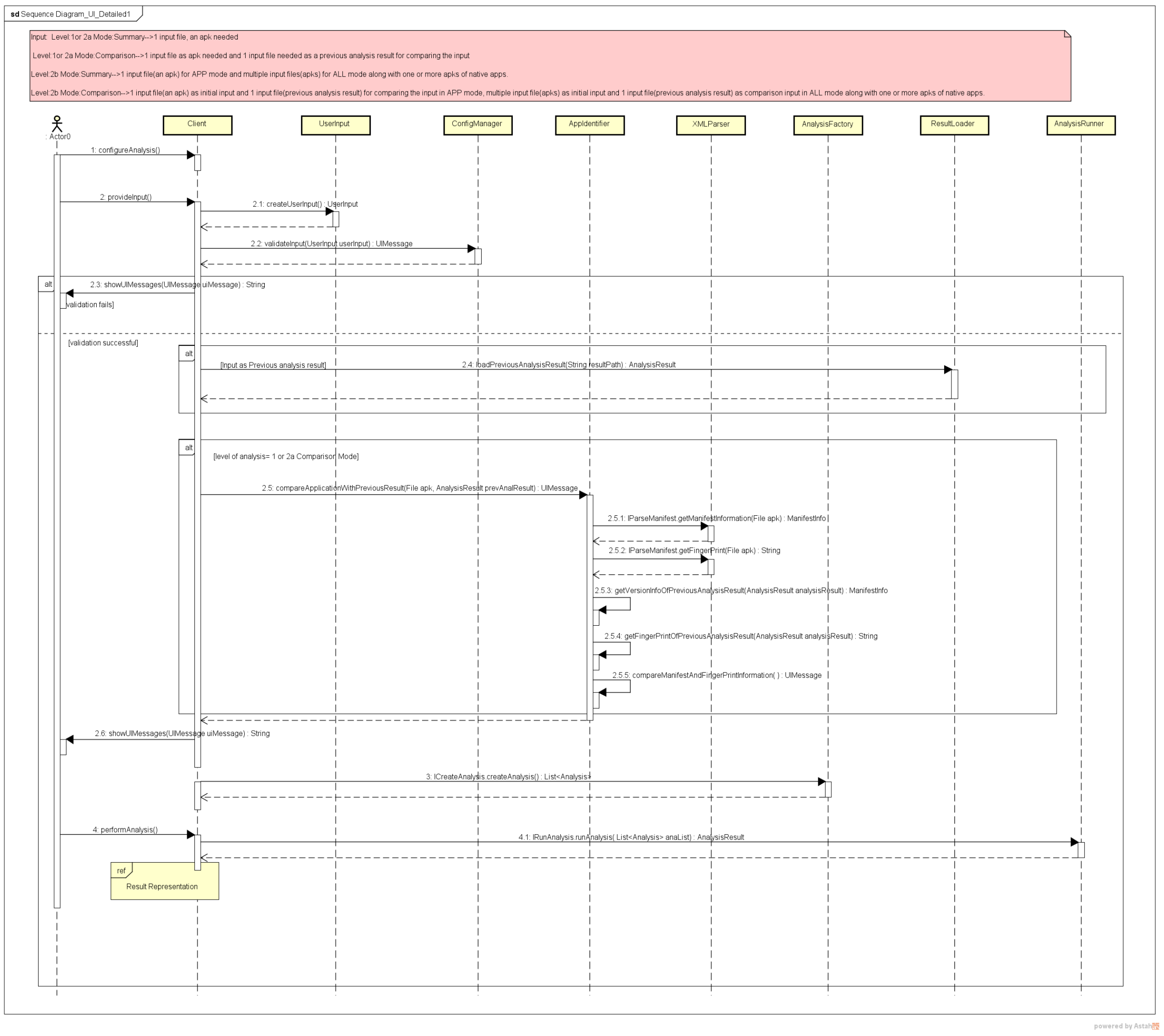

sd Sequence Diagram_UI_Detailed1
Input: Level:1or 2a Mode:Summary-->1 input file, an apk needed
Level:1or 2a Mode:Comparison-->1 input file as apk needed and 1 input file needed as a previous analysis result for comparing the input
Level:2b Mode:Summary-->1 input file(an apk) for APP mode and multiple input files(apks) for ALL mode along with one or more apks of native apps.
Level:2b Mode:Comparison-->1 input file(an apk) as initial input and 1 input file(previous analysis result) for comparing the input in APP mode, multiple input file(apks) as initial input and 1 input file(previous analysis result) as comparison input in ALL mode along with one or more apks of native apps.
: Actor0
Client
UserInput
ConfigManager
AppIdentifier
XMLParser
AnalysisFactory
ResultLoader
AnalysisRunner
1: configureAnalysis()
2: provideInput()
2.1: createUserInput() : UserInput
2.2: validateInput(UserInput userInput) : UIMessage
alt
2.3: showUIMessages(UIMessage uiMessage) : String
validation fails]
[validation successful]
alt
[Input as Previous analysis result]
2.4: loadPreviousAnalysisResult(String resultPath) : AnalysisResult
alt
[level of analysis= 1 or 2a Comparison Mode]
2.5: compareApplicationWithPreviousResult(File apk, AnalysisResult prevAnalResult) : UIMessage
2.5.1: IParseManifest.getManifestInformation(File apk) : ManifestInfo
2.5.2: IParseManifest.getFingerPrint(File apk) : String
2.5.3: getVersionInfoOfPreviousAnalysisResult(AnalysisResult analysisResult) : ManifestInfo
2.5.4: getFingerPrintOfPreviousAnalysisResult(AnalysisResult analysisResult) : String
2.5.5: compareManifestAndFingerPrintInformation( ) : UIMessage
2.6: showUIMessages(UIMessage uiMessage) : String
3: ICreateAnalysis.createAnalysis() : List<Analysis>
4: performAnalysis()
4.1: IRunAnalysis.runAnalysis(List<Analysis> anaList) : AnalysisResult
ref
Result Representation
powered by Astah

Figure 43: Sequence Diagram for User Interface - Detail Level1

ConfigManager The attribute for accessing the methods of the `ConfigManager` class through the `Client` class serves two purposes. One for configuring the initial analysis settings using the `configureAnalysisSetting(...)`, and the other for validating the provided user input with the help of the *validateInput(...)*. This method accepts the object of the `UserInput` class as the only parameter. This validation method returns the value of type `UIMessage`, which contains necessary information about the reason for the validation failure. If the validation failed, then our tool will display necessary messages and won't allow the user to proceed further until the correct inputs are entered.

ResultLoader With respect to the sequence diagram and the class diagram, the purpose for using the `ResultLoader` class is for loading the previous analysis result, which is required for the COMPARISON mode. The `loadPreviousAnalysisResult(...)` in the `ResultLoader` class will return the object of the `AnalysisResult` class which will be stored in the `previousAnalysisResult` attribute of the `UserInput` class. This stored result will be used for the subsequent analysis whenever required.

AppIdentifier `ApplicationEntryPoint` class will call the `compareApp()` in the Business Logic class only for Level 1 and Level 2a. Once the `Client` got the previous analysis result, it will call the `compareAppWithPreviousResult(...)` in the `AppIdentifier` class for the COMPARISON scenario. There are certain internal methods in this class which will get the application version code, version name and fingerprint through external method call to the `A3XmlParser` class. The external call will be made only for getting the information related to the input `.apk` file. The `ManifestInfo` class is used for getting the application version related information through the attributes `versionCode` and `versionName`. The internal methods used in this case for getting the manifest information and the fingerprint information are `getManifestInformation(...)` and `getFingerPrint(...)`.The related manifest and fingerprint information from the previous analysis result will be fetched using separate methods and finally based on the comparison between both the results, the `compareAppWithPreviousResult(...)` will return the value as `UIMessage` to the `Client` class which in turn will be displayed to the user as a String value.

In general, user can start performing the analysis once they have given the necessary input in the application. This can be done through the `performAnalysis()` in the Business Logic class. One important point to remember is that validating the user input will be skipped incase of COMPARISON mode of Level 1 and Level 2a as the valdation is done before starting the comparison process. For other scenarios,

1. The **first step** is user input validation through the method in `ConfigManager` class.
2. The **Second step**, Business Logic Class calls the `Analysis` Class with the necessary parameters to fetch the return value as *List<Analysis>*.
3. **Third step**, for getting the actual analysis result we will pass the result which we got from the previous step to an external method of `AnalysisRunner` class.

ResultStorer After getting the analysis result from the `AnalysisRunner`, the user had an option to either view or save the result. From the second detailed sequence diagram of the UI (see Figure 44) through the `Client` class the instance of this `ResultStorer` class will provides the `storeResult(...)` for saving the analysis result at the user provided file location.

ResultFilter If the user want to view the analysis result instead of saving it, then there is a possibility to view the result in either the graphical mode or textual mode. The results for both modes are available in the instance of `AnalysisResult` class which we got earlier. The results can be filtered by the users by selecting the detail level and result filter using the `filterTextualViewResult(...)` and *filterGraphicalViewResult(...).*

4.5.2 ClientGUI

As mentioned in the previous section this `ClientGUI` class contains few more specific methods especially for GUI, other than the extended common Business Logic methods from the `Client` abstract class. This sub class is having the constructor with the parameters similar to the one in its super class. Other than that it had one specific method named `proceed-ToComparison(...)`, which is used for saving the SUMMARY mode result in the result view page and will make sure the file location and other important attributes such as `Level`, `LevelSpecificMode` in its object. So when the GUI page moves to the menu page for the COMPARISON mode, these persisted values can be re-used by the GUI from the returned object of this class.

This class is having an attribute which is specific for creating the instance of the `GUI` class to access the related methods for displaying the graphical and textual result in the GUI.

GUI `GUI` class is specifically used for showing the result to the user in GUI. It has two attributes `graphicalViewCreator` and `textualViewCreator` which represents the respective instance of the `GraphicalViewCreator` class and `TextualViewCreator` class. The `invokeGraphicalViewCreator(...)` in this class is used for calling the methods using the `graphicalViewCreator` attribute for displaying the graphical analysis result in the GUI. Similarly, `invokeTextualViewCreator(...)` access the methods which are accessible through the `textualViewCreator` attribute for displaying the textual analysis result in the GUI. Parameter which needs to be passed for both these methods are string.

TextualViewCreator This class has a method named `showTextualViewOutput(...)`, which will take the textual analysis result which was represented as a html string format. Then using this parameter it will create the necessary GUI along with this html content for the user to view it.

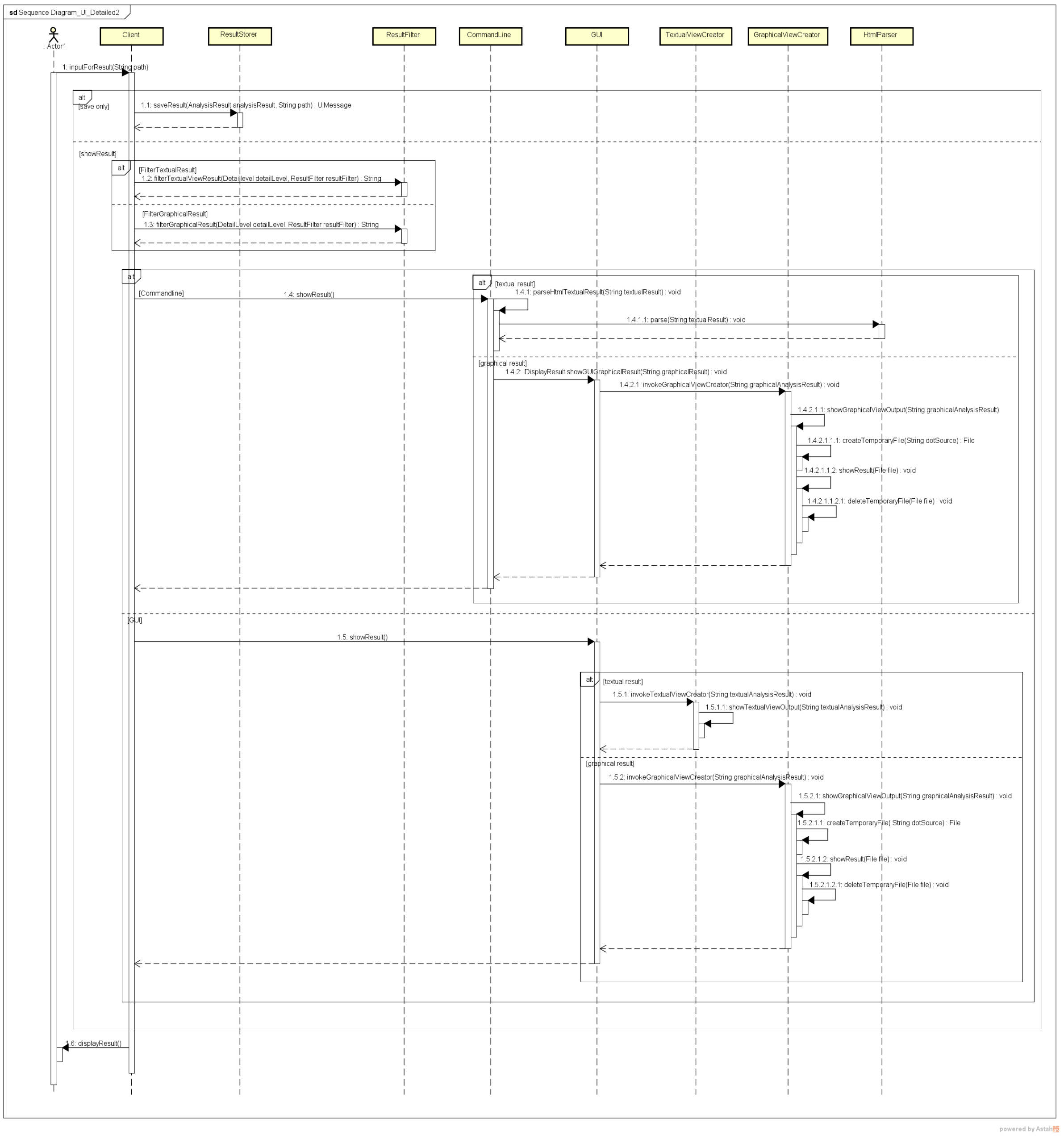

sd Sequence Diagram_UI_Detailed2
: Actor1
Client
ResultStorer
ResultFilter
CommandLine
GUI
TextualViewCreator
GraphicalViewCreator
HtmlParser
1: inputForResult(String path)
alt
[save only]
1.1: saveResult(AnalysisResult analysisResult, String path) : UIMessage
[showResult]
alt
[FilterTextualResult]
1.2: filterTextualViewResult(Detaillevel detailLevel, ResultFilter resultFilter) : String
[FilterGraphicalResult]
1.3: filterGraphicalResult(DetailLevel detailLevel, ResultFilter resultFilter) : String
alt
[Commandline]
1.4: showResult()
alt
[textual result]
1.4.1: parseHtmlTextualResult(String textualResult) : void
1.4.1.1: parse(String textualResult) : void
[graphical result]
1.4.2: IDisplayResult.showGUIGraphicalResult(String graphicalResult) : void
1.4.2.1: invokeGraphicalViewCreator(String graphicalAnalysisResult) : void
1.4.2.1.1: showGraphicalViewOutput(String graphicalAnalysisResult)
1.4.2.1.1.1: createTemporaryFile(String dotSource) : File
1.4.2.1.1.2: showResult(File file) : void
1.4.2.1.1.2.1: deleteTemporaryFile(File file) : void
[GUI]
1.5: showResult()
alt
[textual result]
1.5.1: invokeTextualViewCreator(String textualAnalysisResult) : void
1.5.1.1: showTextualViewOutput(String textualAnalysisResult) : void
[graphical result]
1.5.2: invokeGraphicalViewCreator(String graphicalAnalysisResult) : void
1.5.2.1: showGraphicalViewOutput(String graphicalAnalysisResult) : void
1.5.2.1.1: createTemporaryFile(String dotSource) : File
1.5.2.1.2: showResult(File file) : void
1.5.2.1.2.1: deleteTemporaryFile(File file) : void
6: displayResult()
powered by Astah

Figure 44: Sequence Diagram for User Interface - Detail Level2

GraphicalViewCreator This class has the method named `showGraphicalViewOutput` which is used for accessing showing the graphical analysis result to graphically in the GUI. As mentioned in the earlier paragraphs, the graphical analysis result represented as a string format where the structure of the graph was represented in the DOT language format. So for parsing this DOT language to the equivalent graphical representation, we are going to use the open source library name Gephi Toolkit for rendering the DOT language graphically. In this class, we are going to import the appropriate packages from the Gephi Toolkit for using the necessary classes for rendering the image. As the class in the toolkit only accepts the file for reading and rendering the graph, we are creating the temporary file using the `createTemporayFile(...)` by passing the graphical analysis result string. After rendering the graph from the file, the result will be shown in the GUI and then the temporary file will be deleted.

4.5.3 ClientCommandLine

As mentioned in the first section, other than the common Business Logic classes from the abstract base class this `ClientCommandLine` class has few additional methods which are very specific when the user accesses our application through the command line. Similar to the ClientGUI class, this one has constructor with parameters that match the base class. With respect to the validation, this class has a method named `validateCommandLineIn-putArguments(...)` which is used for the initial validation of the user input. This method will check whether the user has entered the commands and the option names correctly without any mistakes. So, if validation fails, then the tool will display the relevant information for the cause of validation failure and will ask the user to re-enter the input again.

There is a provision for continuing the analysis after the application comparison results displayed in the GUI. Similarly, command line mode gives the option for user to continue the analysis or not by asking the input from them. There is a specific method called `continue-Analysis(...)`, which is used for this scenario to let the application decide whether to continue the analysis or not based on the user input.

The remaining three methods, `displayCommandLineTextualResult(...)`, `dis-playGraphicalResult(...)`, `displayCommandLineMessageResult(...)` are specific for displaying the result to the user in either the textual analysis result through command line or graphical analysis result with the help of the instance of the `CommandLine` class which will be explained in the next paragraph. There is an attribute named `isView-GraphicalResult` in this class to decide whether the user has requested to view the result graphically or not, which will be used in the `ApplicationEntryClass` class to call the specific method.

CommandLine This class provides the methods which are related to displaying the results to the user. The `parseHtmlTextResult(...)` is specific for calling the instance of the `HtmlParser` class to parse the textual result which is of the html string, to display the output in the command line. More information about the HtmlParser class is mentioned in the next paragraph. For displaying the graphical result, the `showGUIGraphicalResult(...)` will internally call the method which was mentioned in the previous section in the `GraphViewCreator` class with the help of the instance of the `GUI` class.

HtmlParser As mentioned in the previous paragraph, this class has `parse(...)` which will parse the html string. It extend the `HTMLEditorKit.ParseCallback` class which is available as a part of the Java Swing package.

4.5.4 UI StateChart Diagram

The initial steps involved with UI can be explained clearly with a state machine diagram (see Figure 45).
For both the UI (GUI or Command Line), the first step is to get the inputs from the user.
After the inputs are provided, analysis will start either by comparing the version information of apps in COMPARISON mode by the calling method `compareApp()` or directly initiating the analysis by calling method `performAnalysis()`.In the analysis procedure, first it triggers `validateInput()` for the validation of inputs.
If the validation fails, a validation error message is shown through `showUIMessages()` and again user is asked to provide the correct input. If the validation succeeds then further analysis is done by checking whether we are performing the analysis in COMPARISON mode by checking `isComparisonMode`.
In COMPARISON mode, first the previous analysis result is loaded by calling method `loadPreviousAnalysisResult(...)`.After the previous analysis result is loaded, we check for COMPARISON mode in Level 1 or 2a and compare the version information of both the input provided and showed it to the user through the `compareAppWithPreviousResult(...)` method. After the comparison message is shown,the user can cancel or choose to proceed with further analysis.
Incase of Summary mode or Level 2b COMPARISON mode or user chooses to proceed with the analysis after the comparison message is shown, further analysis is started through `createAnalysis()`. In this Analysis object is created and send it to `AnalysisRunner` class where the actual analysis starts through `runAnalysis(...)`.

5 User Interface

Introduction This section describes the User Interface of our tool. It contains the information about two important scenarios with respect to the User Interface Design. They are, GUI Descriptions and Navigation and Command Line Configurations.

The content of this section is divided into the following sub-sections:

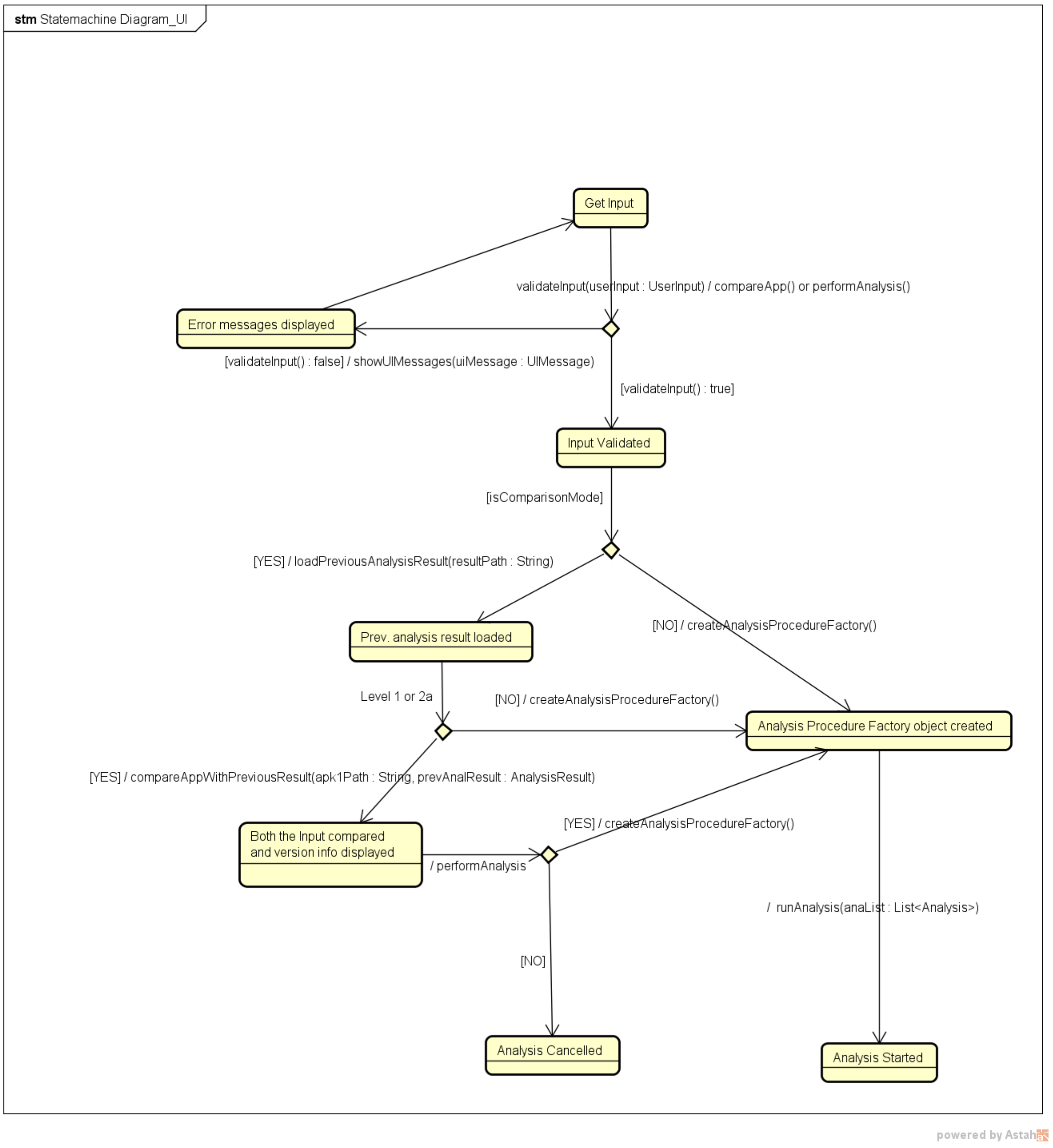

stm Statemachine Diagram_UI
Get Input
validateInput(userInput : UserInput) / compareApp() or performAnalysis()
Error messages displayed
[validateInput() : false] / showUIMessages(uiMessage : UIMessage)
[validateInput() : true]
Input Validated
[isComparisonMode]
[YES] / loadPreviousAnalysisResult(resultPath : String)
[NO] / createAnalysisProcedureFactory()
Prev. analysis result loaded
Level 1 or 2a
[NO] / createAnalysisProcedureFactory()
Analysis Procedure Factory object created
[YES] / compareAppWithPreviousResult(apk1Path : String, prevAnalResult : AnalysisResult)
[YES] / createAnalysisProcedureFactory()
Both the Input compared and version info displayed
/ performAnalysis
/ runAnalysis(anaList : List<Analysis>)
[NO]
Analysis Cancelled
Analysis Started
powered by Astah

Figure 45: State Chart Diagram for Initial Steps of the User Interface

- Sketches
  GUI Structure, describes the structure of the application, and the ways in which users can navigate.
  GUI Descriptions, describes how each individual screen is going to look like in our tool and represents that each individual component comprises a screen.
- Workflow, illustrates about how the navigation within any screen and between different screens are possible in our tool. It also mentions how the navigation between the screens was structured by considering the ease of usability from a user perspective as well as maintaining consistency from a UI Design perspective.
- Command Line
  Command Line Configuration, describes the configurations the user has to provide to run our tool from the Command Line mode.

So, we can use this section as a template or single point of reference for the UI development.

5.1 Sketches

GUI Structure The overall structure of our tool is relatively simple, as shown in the Figure 46. All the necessary screens of our tool are accessed directly from the `Main Page`.

GUI Descriptions

1. **Main Page**
   Refer Figure 47
   Descriptions When the tool is launched, at first the `Main Page` will be shown to the user.
   Elements The following is a list of all elements in this screen.
   a) **’Main’**

      Type: Menu
      Label: ’Main’
      Behaviour: Clicking on Main will open the `Main Page`.
   b) **’Analyze’**

      Type: Menu
      Label: ’Analyze’
      Behaviour: Clicking on Analyze will open the `Analysis Screen` where user can analyse App(s) on different level.
   c) **’View Result’**

      Type: Menu
      Label: ’View Result’

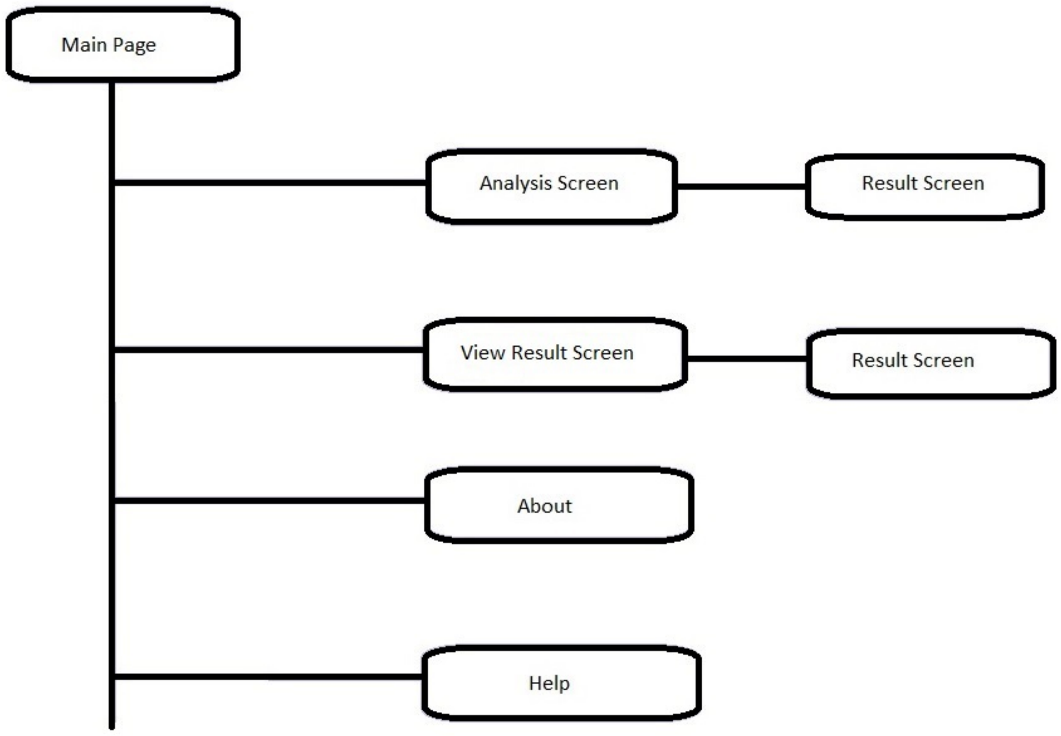


Figure 46: GUI Structure

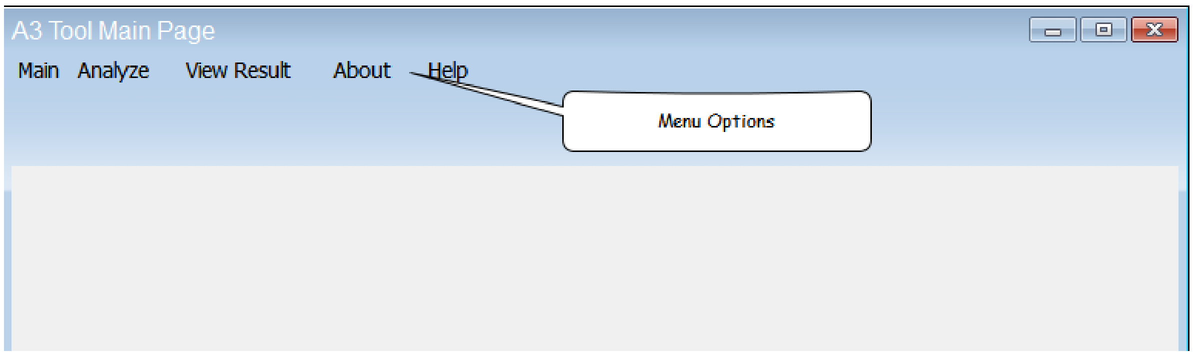


Figure 47: Main Page

Behaviour: Clicking on View Result will open another screen where user can load any previously stored result and view it.

d) **'About'**

Type: Menu
Label: 'About'
Behaviour: Clicking on About will open another screen which gives general information about our tool.

e) **'Help'**

Type: Menu
Label:] 'Help'
Behaviour: Clicking on Help will open another screen which gives information about how to navigate through the windows available in our tool.

2. **Analysis Screen**
Refer Figure 48
Descriptions `Analysis Screen` will allow user to analyse Apps in different analysis level and modes. This screen contains all the elements that will appear on `Analysis Screen` during different analysis levels.
Elements The following is a list of all elements in this screen.

a) **'Select the Level'**

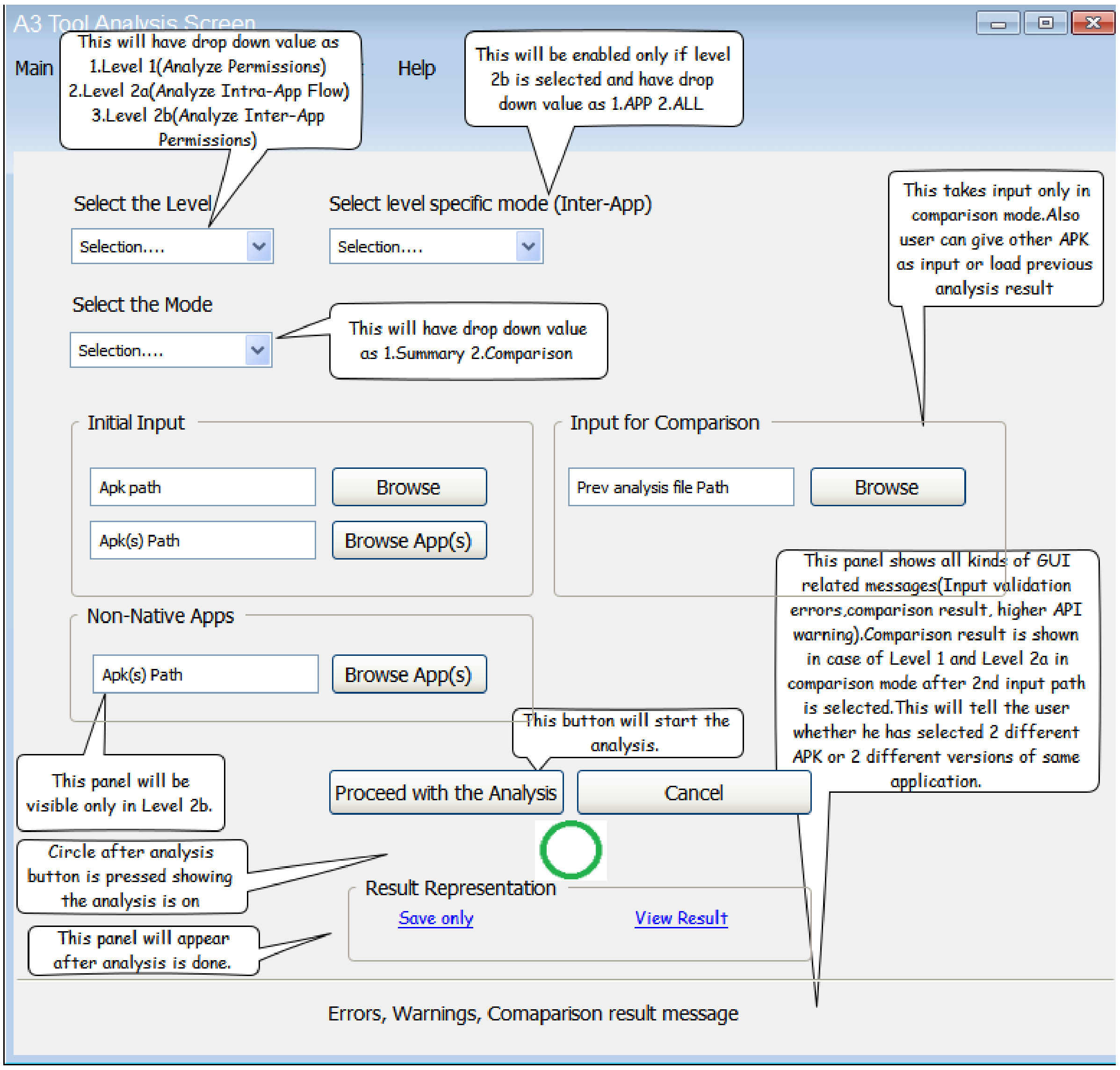
Type: Dropdown list
Label: 'Select the Level'
Content: This will have drop down value as 1.Level 1(Analyze Permissions) 2.Level 2a(Analyze Intra-App Flow) 3.Level 2b(Analyze Inter-App Permissions)
Default: User has to select the level in which he wants to perform analysis.

b) **'Select the level specific mode'**

Type: Dropdown list
Label: 'Select the level specific mode(Inter-App)'
Content: This will have drop down value as 1. APP 2. ALL
Default: User has to select the mode in which he wants to perform analysis.
Comment: This will be visible only if Level 2b is selected in the *Select the Level* dropdown list.

c) **'Select the Mode'**

Type: Dropdown list
Label: 'Select the Mode'
Content: This will have drop down value as 1. Summary 2. Comparison
Default: User has to select the mode in which he wants to perform analysis.



This will have drop down value as 1.Level 1(Analyze Permissions) 2.Level 2a(Analyze Intra-App Flow) 3.Level 2b(Analyze Inter-App Permissions)

This will be enabled only if level 2b is selected and have drop down value as 1.APP 2.ALL

This takes input only in comparison mode.Also user can give other APK as input or load previous analysis result

This will have drop down value as 1.Summary 2.Comparison

This panel shows all kinds of GUI related messages(Input validation errors,comparison result, higher API warning).Comparison result is shown in case of Level 1 and Level 2a in comparison mode after 2nd input path is selected.This will tell the user whether he has selected 2 different APK or 2 different versions of same application.

This button will start the analysis.

This panel will be visible only in Level 2b.

Circle after analysis button is pressed showing the analysis is on

This panel will appear after analysis is done.

Figure 48: Analysis Screen

d) **’Initial Input’**

Type: Group Box
Label: ’Initial Input’
Sub-elements: It contains two text boxes and two buttons with label: Browse, Browse App(s).
Behaviour of Sub-element: Clicking on *Browse* button will open an explorer window to select and load Apk file.
Clicking on *Browse App(s)* button will open an explorer window to select and load Apk files.
Comment: Browse App(s) button and corresponding text box will be visible only if the user has selected ALL mode in Level 2b. Also for providing multiple Apk files, all the input files should be kept in one folder.

e) **’Input for Comparison’**

Type: Group Box
Label: ’Input for Comparison’
Sub-elements: It contains a text box and a button with label: Browse.
Behaviour of Sub-element : Clicking on *Browse* button will open an explorer window to select and load previously saved result file.
In Level 1 and 2a, as soon as the input file is loaded, it will start the comparison between initial input file and the input file given for comparison and the comparison message will be shown in the footer.

f) **’Non-Native Apps’**

Type: Group Box
Label: ’Non-Native Apps’
Sub-elements: It contains a text box and a button with label: Browse App(s).
Behaviour of Sub-element : Clicking on *Browse App(s)* button will open an explorer window to select and load Apk files which will create an environment for Inter-App analysis.
Comment: Non-Native App(s) group box will be visible only if the user has selected Level 2b. Also for providing multiple Apk files, all the input files should be kept in one folder.

g) **’Proceed’**

Type: Button
Label: ’Proceed with the Analysis’
Behaviour: Clicking on this button will start the analysis.

h) **’Cancel’**

Type: Button
Label: ’Cancel’

Behaviour: Clicking on this button will cancel the analysis and take the user to the `Main page`.

i) **'Rotating Circle'**

Type: Image
Behaviour: A *Rotating Circle* will be shown to the user as long as analysis is going on by making other screen elements disabled.

j) **'Result Representation'**

Type: Group Box
Label: 'Result Representation'
Sub-elements: It contains two link button with label: Save only, View Result.
Behaviour of Sub-element : Clicking on *Save only* link button will ask the user for the path and save the result, then the user will be redirected to the `Main Page`. Clicking on *View Result* link button will take user to the `Result Screen`.

k) **'Footer'** *Type:* Label
Behaviour: It will show all kinds of GUI related messages(Input validation errors, comparison result message, higher API warning). Comparison result will be shown in case of Level 1 and Level 2a in COMPARISON mode. Higher API warning will be shown if the user selects an APK built in an API version above the maximum API version supported by our tool.

3. **View Saved Result Screen**
Refer Figure 49
Descriptions `View Result Screen` will allow user to load previously analysed(saved) result file and view the result.
Elements The following is a list of all elements in this screen.

a) **'Select File'**

Type: Group Box
Label: 'Select File'
Sub-elements: It contains a text box and a button with label: Browse.
Behaviour of Sub-element : Clicking on *Browse* button will open an explorer window to select and load the previously saved result file.

b) **'View Result'**

Type: Button
Label: 'View Result'
Behaviour: Clicking on *View Result* button will open `Result Screen` which contains the loaded result of the file selected.

4. **About Screen**
Descriptions `About Screen` will provide general information about our tool.
Elements The following is a list of all elements in this screen.

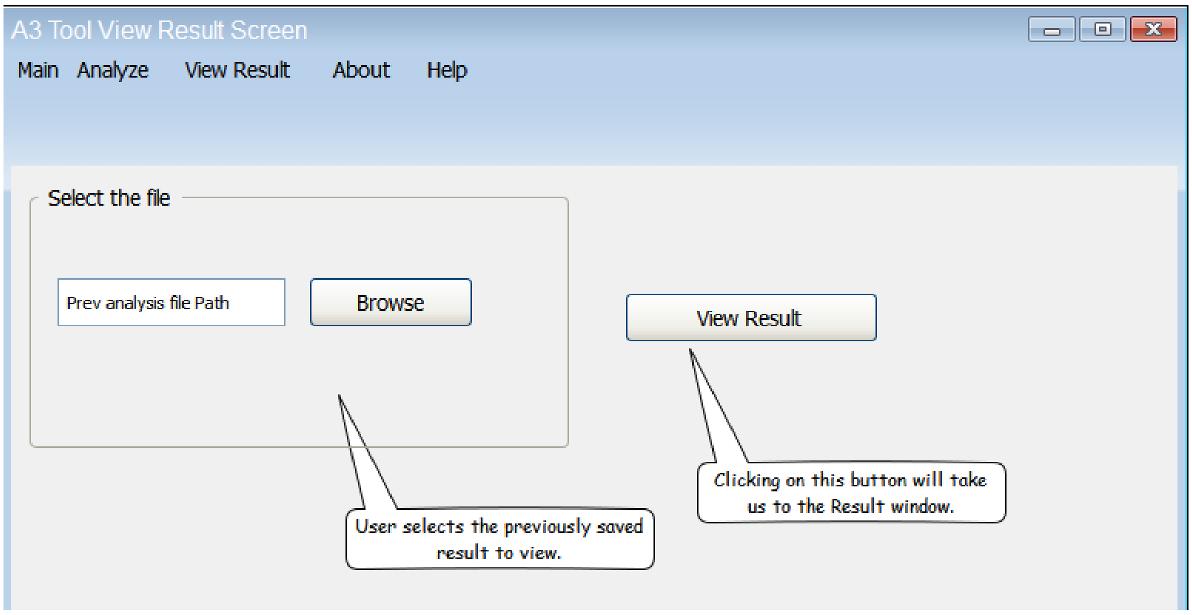

A3 Tool View Result Screen
Main
Analyze
View Result
About
Help
Select the file
Prev analysis file Path
Browse
View Result
User selects the previously saved result to view.
Clicking on this button will take us to the Result window.

Figure 49: View Saved Result Screen

a) **'About'**

   Type: Label
   Label: About

b) **'Information'**

   Type: Label
   Content: general information about our tool.

5. **Help Screen**
   Descriptions User can refer to `Help Screen` incase of any clarification needed for operating our tool.
   Elements The following is a list of all elements in this screen.

   a) **'Help'**

      Type: Label
      Label: Help

   b) **'Information'**

      Type: Label
      Content: information about how to navigate through the windows available in our tool.

6. **Result Screen**
   Refer Figure 50

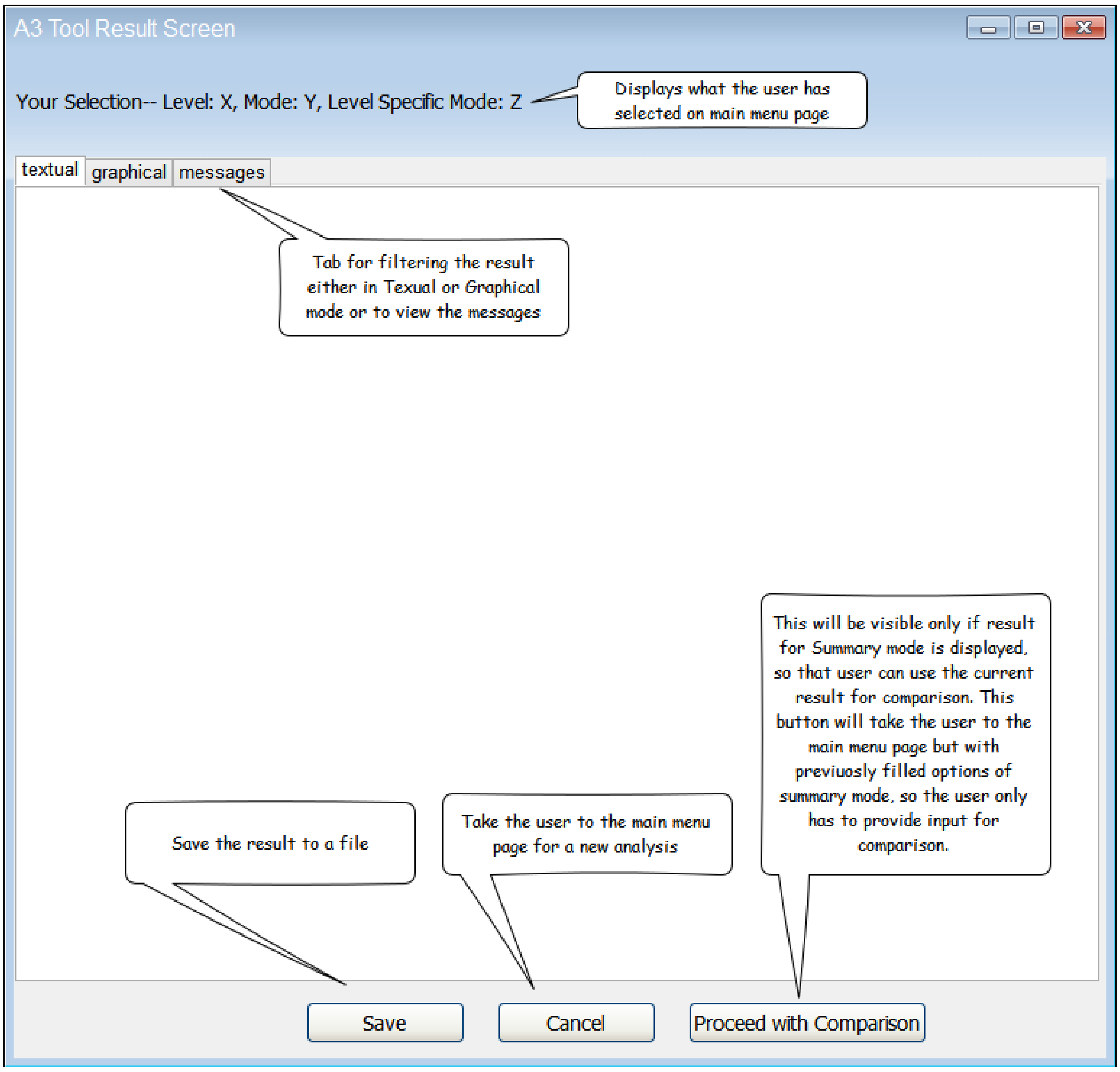

A3 Tool Result Screen
Your Selection-- Level: X, Mode: Y, Level Specific Mode: Z
Displays what the user has selected on main menu page
textual
graphical
messages
Tab for filtering the result either in Texual or Graphical mode or to view the messages
This will be visible only if result for Summary mode is displayed, so that user can use the current result for comparison. This button will take the user to the main menu page but with previuosly filled options of summary mode, so the user only has to provide input for comparison.
Save the result to a file
Take the user to the main menu page for a new analysis
Save
Cancel
Proceed with Comparison

Figure 50: Result Screen

Descriptions `Result Screen` will allow the user to view the result in two different ways i.e textual and graphical. Also user can view the analysis specific messages on `Result Screen`. Figure 50 shows all the common elements present on the `Result Screen`.
Elements The following is a list of all elements in this screen.

a) **'Your Selection'**

   Type: Label
   Content: This label will show what options the user has selected on analysis page before coming to result page.

b) **'Tabs'**

   Type: Tab Panel
   Content: There will be three tabs on `Result Screen`. **textual** tab will show the corresponding analysis result in textual mode and this tab will be selected by default when the `Result Screen` is shown for the first time. **graphical** tab will show the corresponding analysis result in graphical mode. **message** tab will show all the messages related to the analysis result.

c) **'Save'**

   Type: Button
   Label: 'Save'
   Behaviour: Clicking on *Save* button will ask the user for the path and save the result,then the user will be redirected to the `Main Page`.

d) **'Cancel'**

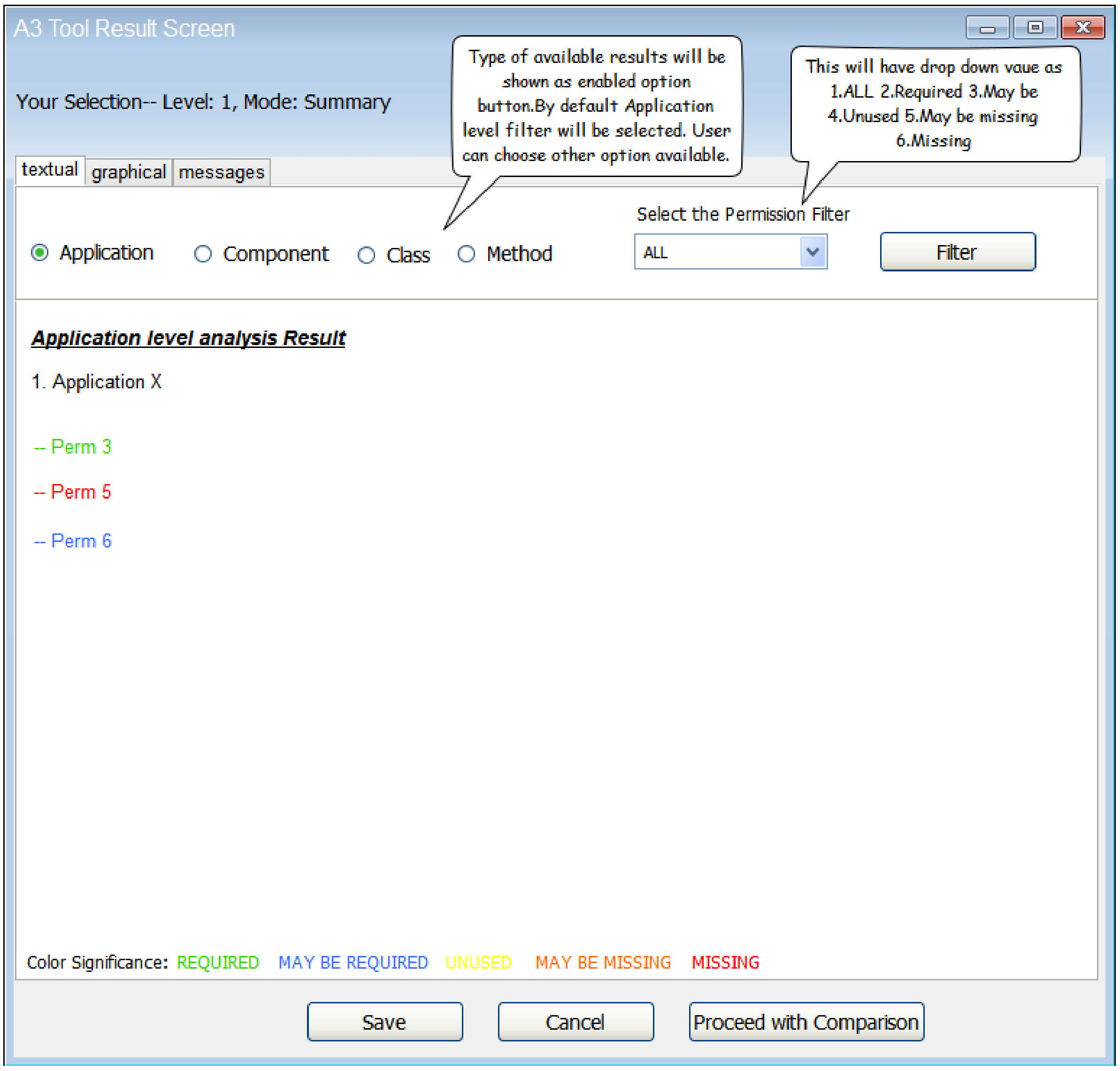
   Type: Button
   Label: 'Cancel'
   Behaviour: Clicking on *Cancel* button will take the user to the `Main Page` without saving the results.

e) **'Proceed with Comparison'**

   Type: Button
   Label: 'Proceed with Comparison'
   Behaviour: This button will be enabled only if the result screen is shown for SUMMARY mode. Clicking on this button will save the result and take user to the `Analysis Screen` with the elements visible for the COMPARISON mode for the analysis level selected earlier by the user.

Along with the above described common elements, `Result Screen` will have different set of filters as per different level of analysis.

- **Level 1 Result Screen**
  Descriptions `Level 1 Result Screen` will have specific set of filters along


A3 Tool Result Screen

Your Selection-- Level: 1, Mode: Summary

Type of available results will be shown as enabled option button.By default Application level filter will be selected. User can choose other option available.

This will have drop down vaue as 1.ALL 2.Required 3.May be 4.Unused 5.May be missing 6.Missing

textual
graphical
messages
[✓ Application] Component Class Method
Select the Permission Filter
ALL
Filter

Application level analysis Result

1. Application X

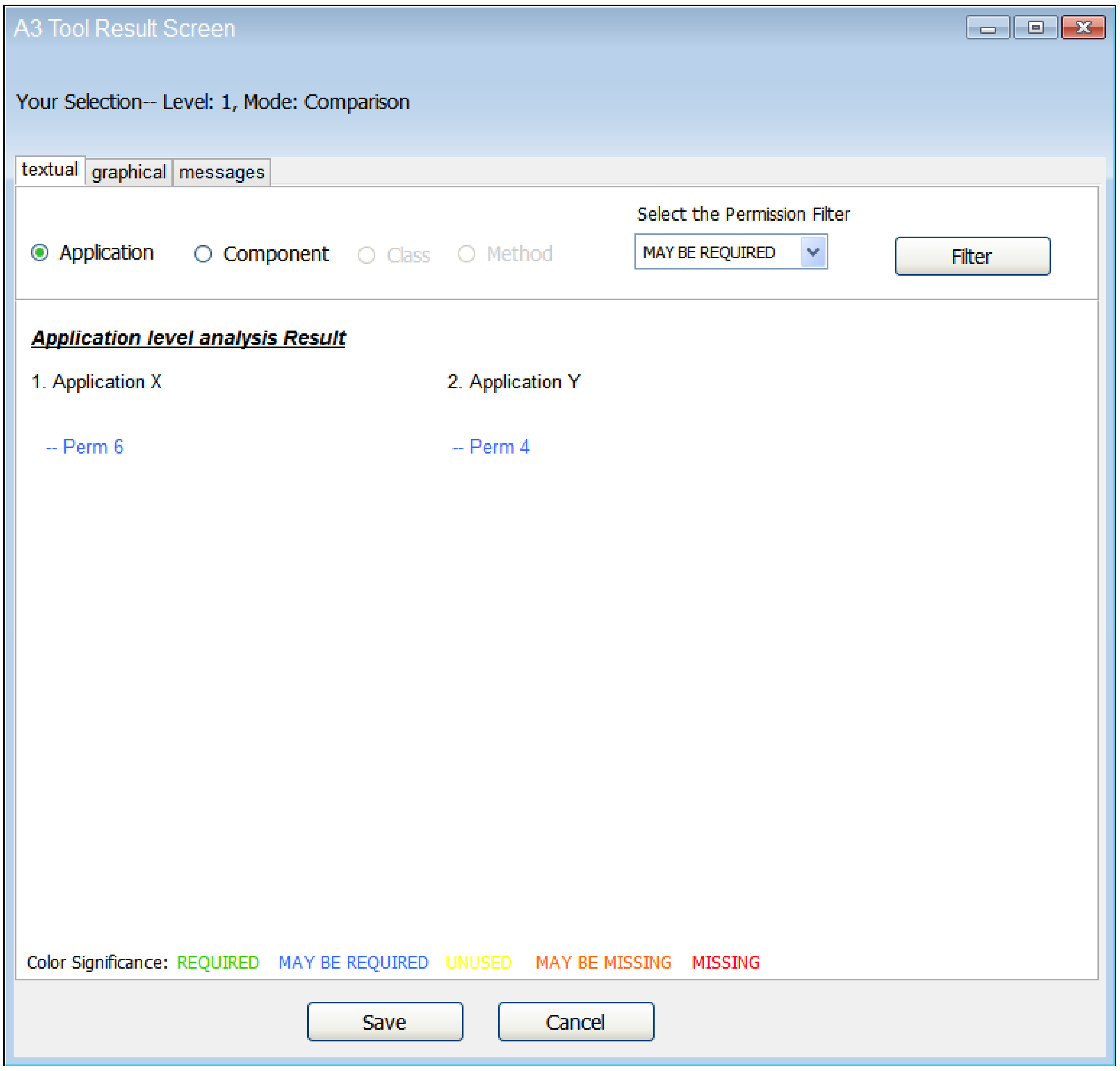
-- Perm 3

-- Perm 5

-- Perm 6

Color Significance: REQUIRED MAY BE REQUIRED UNUSED MAY BE MISSING MISSING

Figure 51: Result Screen for Level 1 Analysis in Summary Mode



Your Selection-- Level: 1, Mode: Comparison

[✓ Application] Component Class Method

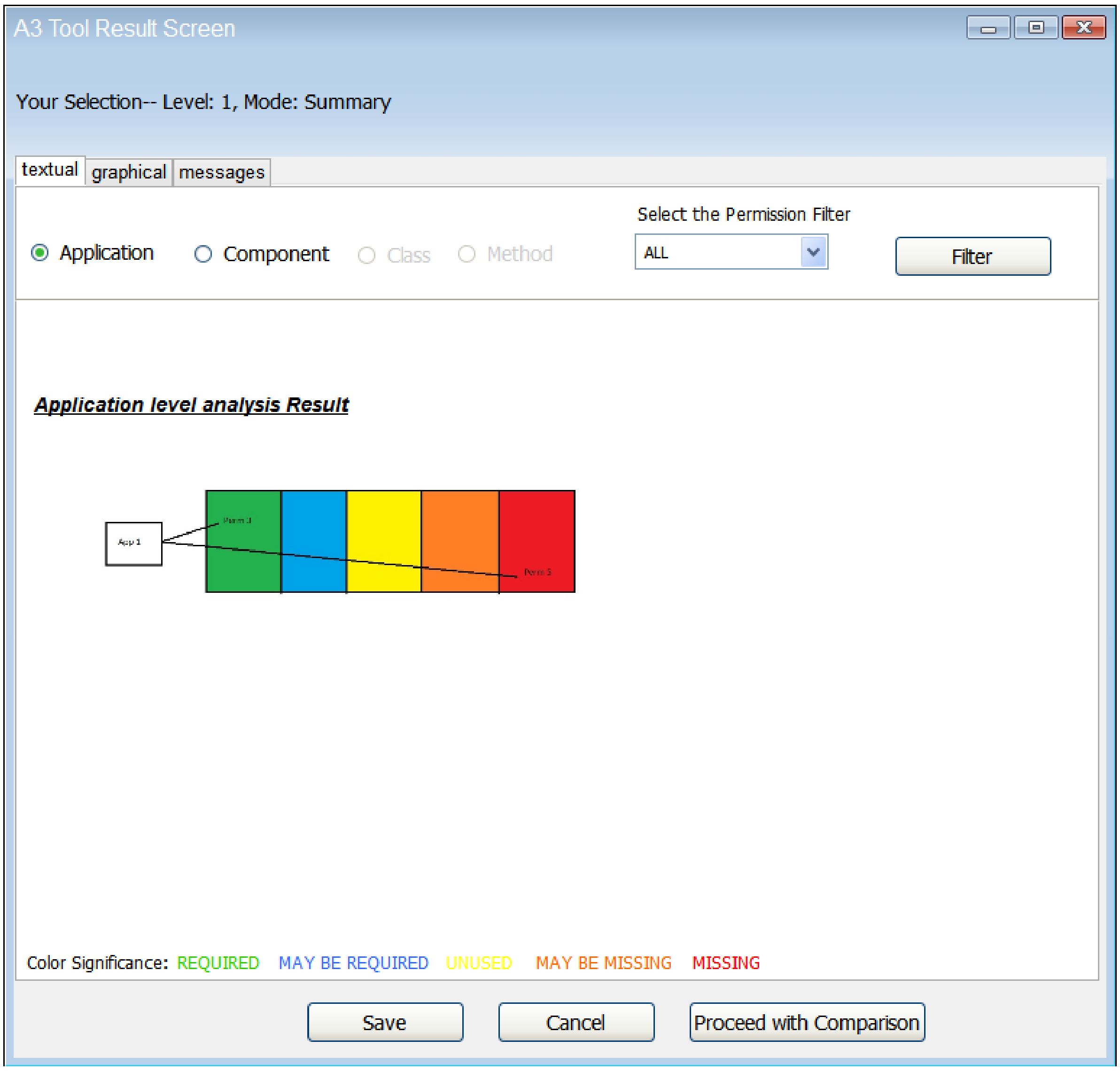
Select the Permission Filter: MAY BE REQUIRED

Application level analysis Result

1. Application X
   -- Perm 6
2. Application Y
   -- Perm 4

Color Significance: REQUIRED MAY BE REQUIRED UNUSED MAY BE MISSING MISSING

Figure 52: Result Screen for Level 1 Analysis in Comparison Mode


A3 Tool Result Screen

Your Selection-- Level: 1, Mode: Summary

textual | graphical | messages

[✓ Application] Component Class Method

Select the Permission Filter: ALL

Filter

Application level analysis Result

Color Significance: REQUIRED MAY BE REQUIRED UNUSED MAY BE MISSING MISSING

Save | Cancel | Proceed with Comparison

Figure 53: Result Screen for Level 1 Analysis in Graphical View

with the common elements of result screen as shown in Figure 51, Figure 52, Figure 53
Elements The following is a list of all elements specific to this screen.

a) **'Detail level Filters'**

   Type: Option Buttons
   Label: 'Application','Component','Class','Method'
   Behaviour: These filters will re-arrange the result into different detail level i.e *application level, component level, class level, method level*. Only those option button will be enabled for which the results are available. For ex. if we are comparing two different Apps then only *application* and *component* filter will be enabled, but if we are comparing two different version of same App then all four filters will be enabled. By default *application level* filter will be selected.

b) **'Permission Filters'**

   Type: Dropdown List
   Label: 'Select the Permission Filter'
   Content: This will have drop down vaue as 1.ALL 2.Required 3.May be 4.Unused 5.May be missing 6.Missing
   Behaviour: Only that category of permissions will be shown which is selected.User can differentiate between permissions by the color described at the lower part of the screen. By default ALL will be selected.

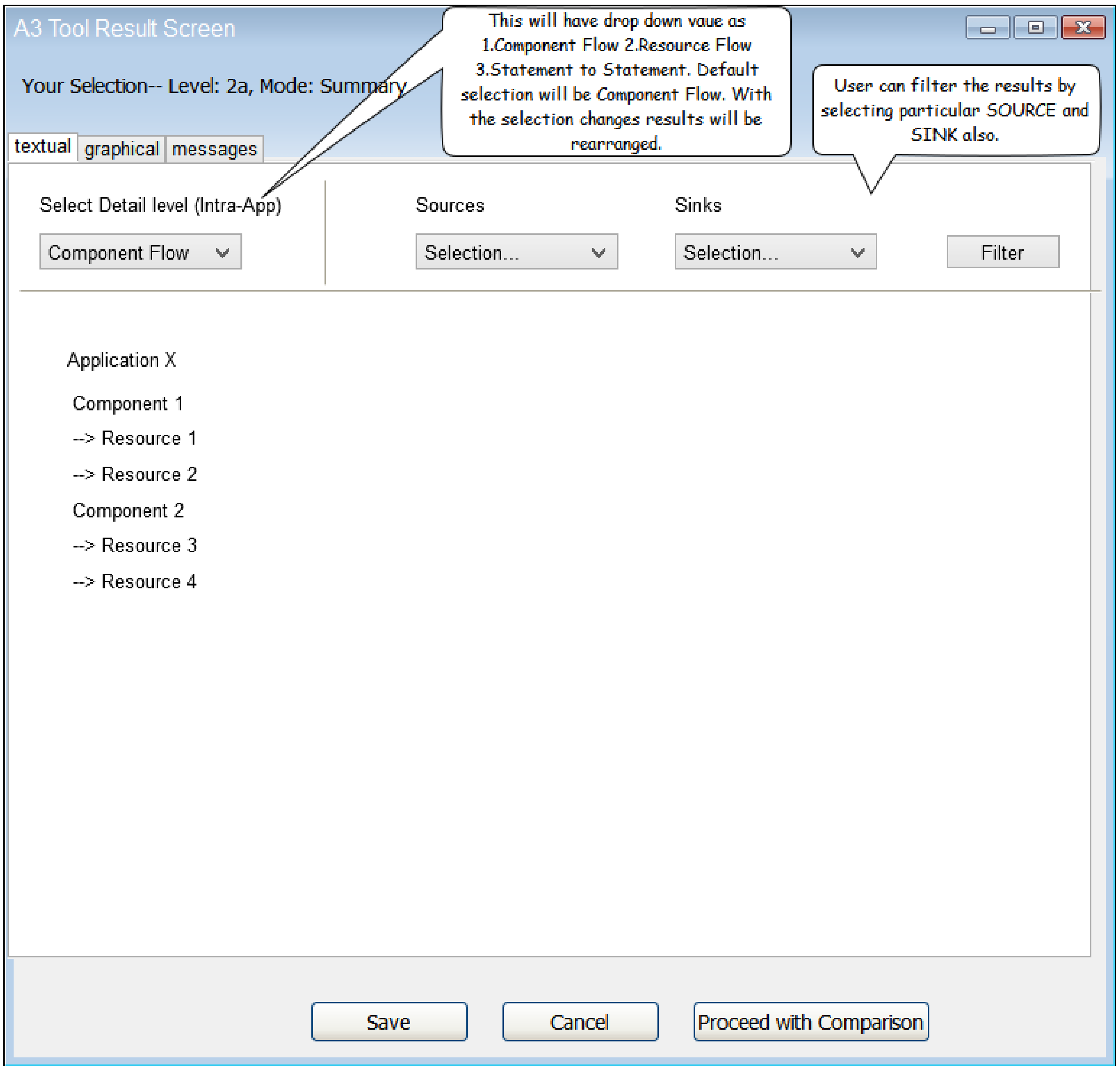
c) **'Filter Button'**

   Type: Button
   Label: 'Filter'
   Behaviour: User needs to click the filter button after the filters i.e Detail Level Filters and Permission Filters are selected. This will re-arrange the result according to the filters selected.

- **Level 2a Result Screen**
  Descriptions `Level 2a Result Screen` will have specific set of filters along with the common elements of `Result Screen` as shown in Figure 54, Figure 55, Figure 56
  Elements The following is a list of all elements specific to this screen.

  a) **'Detail level Filters'**

     Type: Dropdown List
     Label: 'Select Detail level(Intra-App)'
     Content: This will have drop down value as 1.Component Flow 2.Resource Flow 3.Statement to Statement.
     Behaviour: These filters will re-arrange the result into different detail level i.e Component Flow, Resource Flow, Statement to Statement. Default selection



A3 Tool Result Screen

Your Selection-- Level: 2a, Mode: Summary

This will have drop down vaue as 1.Component Flow 2.Resource Flow 3.Statement to Statement. Default selection will be Component Flow. With the selection changes results will be rearranged.

User can filter the results by selecting particular SOURCE and SINK also.

textual | graphical | messages

Select Detail level (Intra-App): Component Flow

Sources: Selection...

Sinks: Selection...

Filter

Application X

Component 1

--> Resource 1

--> Resource 2

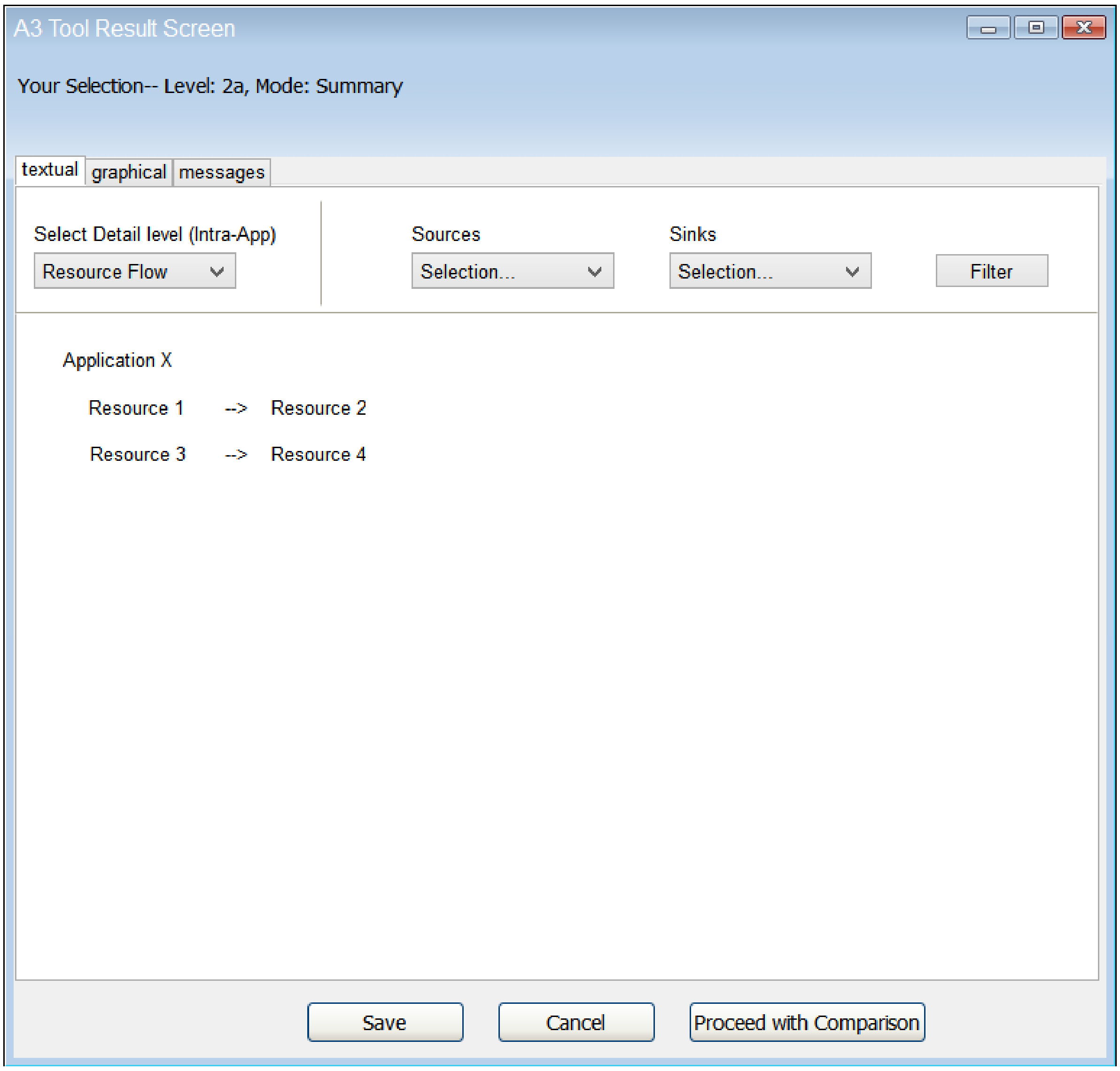
Component 2

--> Resource 3

--> Resource 4

Save | Cancel | Proceed with Comparison

Figure 54: Result Screen for Level 2a Analysis in Summary Mode (Component Flow)



Your Selection-- Level: 2a, Mode: Summary

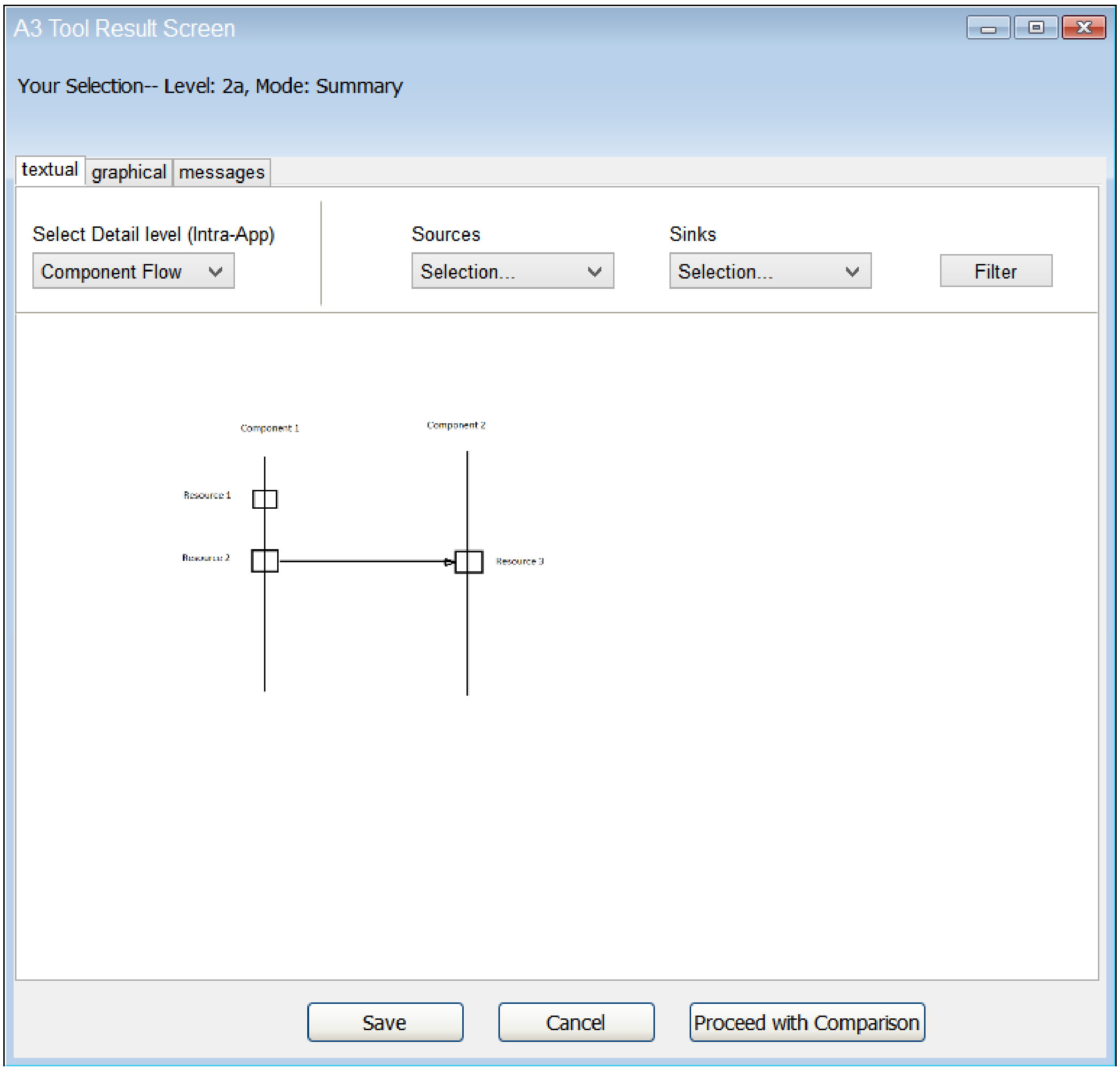
Select Detail level (Intra-App): Resource Flow

Sources: Selection...

Sinks: Selection...

```
Application X
    Resource 1   -->   Resource 2
    Resource 3   -->   Resource 4
```

Figure 55: Result Screen for Level 2a Analysis in Summary Mode (Resouce Flow)



A3 Tool Result Screen

Your Selection-- Level: 2a, Mode: Summary

Select Detail level (Intra-App)
Component Flow

Sources
Selection...

Sinks
Selection...

Figure 56: Result Screen for Level 2a Analysis in Graphical View (Component Flow)

will be Component Flow and with the selection changes results will be re-arranged.

b) **'Source and Sink Filters'**

Type: Dropdown Lists
Label: 'Sources','Sinks'
Content: It will contain all the sources and sinks present in the App(s).
Behaviour: User can filter the results by selecting particular *Source* and *Sink*.

c) **'Filter Button'**

Type: Button
Label: 'Filter'
Behaviour: User needs to click the filter button after the filters i.e Detail Level Filters, Source and Sink Filters are selected. This will re-arrange the result according to the filters selected.

- **Level 2b Result Screen**
Descriptions `Level 2b Result Screen` will have specific set of filters along with the common elements of `Result Screen` as shown in Figure 57, Figure 58
Elements The following is a list of all elements specific to this screen.

a) **'Detail level Filters'**

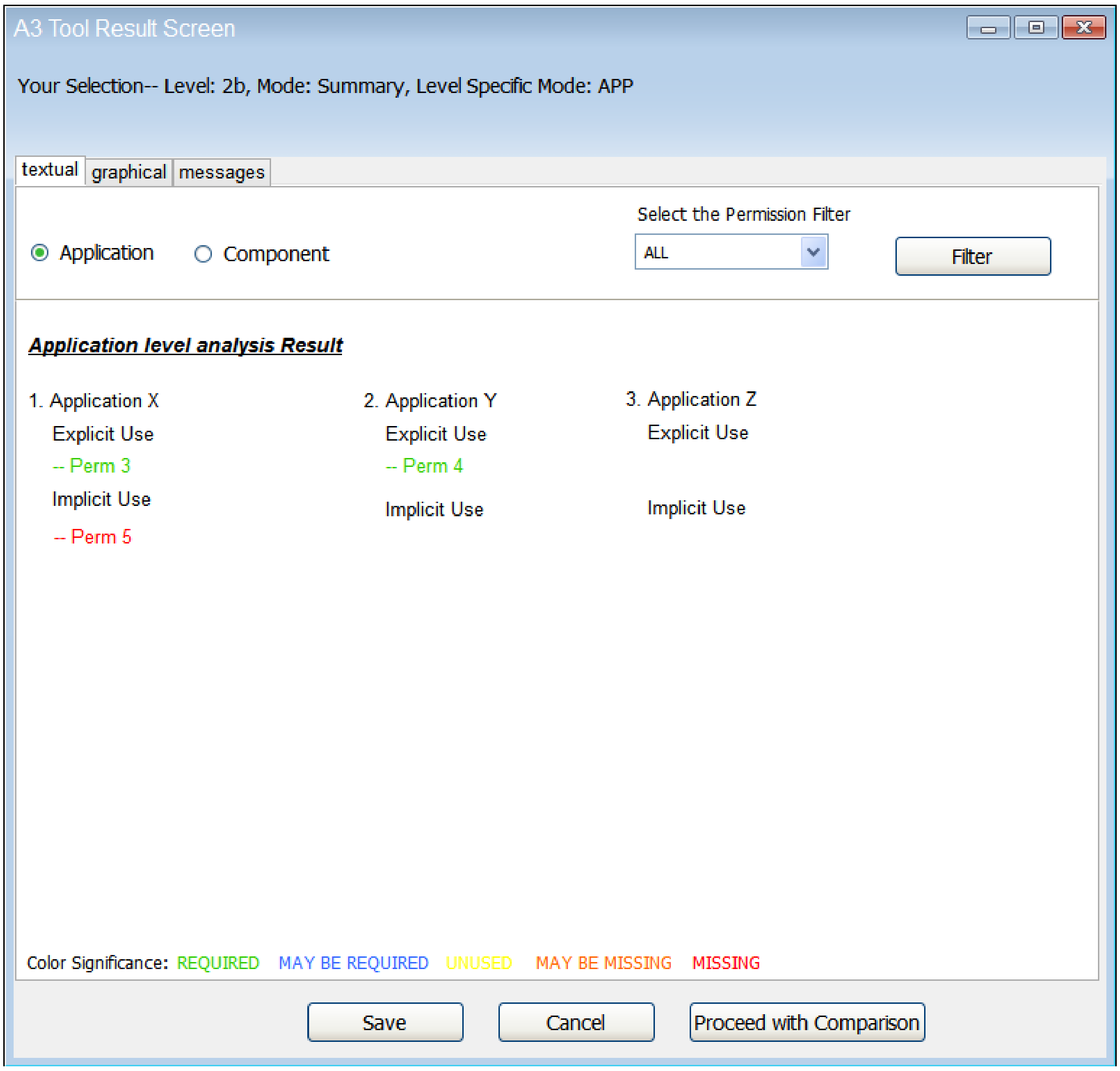
Type: Option Buttons
Label: 'Application','Component'
Behaviour: These filters will re-arrange the result into different detail level i.e *application level, component level*.By default *application level* filter will be selected.

b) **'Permission Filters'**

Type: Dropdown List
Label: 'Select the Permission Filter'
Content: This will have drop down vaue as 1.ALL 2.Required 3.May be 4.Unused 5.May be missing 6.Missing
Behaviour: Only that category of permissions will be shown which is selected. User can differentiate between permissions by the color described at the lower part of the screen.

c) **'Filter Button'**

Type: Button
Label: 'Filter'
Behaviour: User needs to click the filter button after the filters i.e Detail Level Filters and Permission Filters are selected. This will re-arrange the result according to the filters selected.



A3 Tool Result Screen

Your Selection-- Level: 2b, Mode: Summary, Level Specific Mode: APP

textual | graphical | messages

Select the Permission Filter

[✓ Application] Component

ALL

Filter

Application level analysis Result

1. Application X
   Explicit Use
   -- Perm 3
   Implicit Use
   -- Perm 5

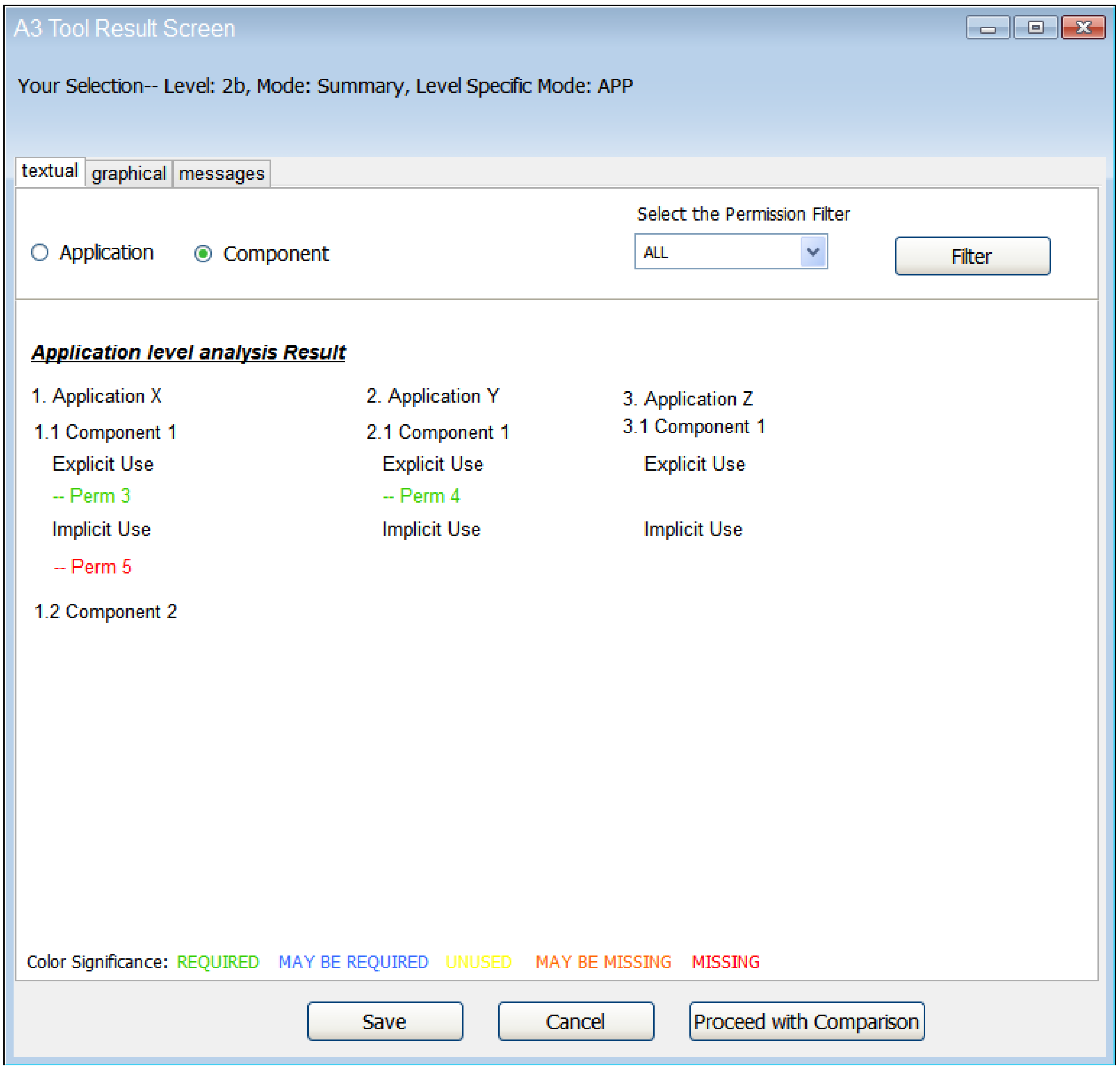
2. Application Y
   Explicit Use
   -- Perm 4
   Implicit Use

3. Application Z
   Explicit Use
   Implicit Use

Color Significance: REQUIRED MAY BE REQUIRED UNUSED MAY BE MISSING MISSING

Save | Cancel | Proceed with Comparison

Figure 57: Result Screen for Level 2b Analysis(Detail Level: Application)



Your Selection-- Level: 2b, Mode: Summary, Level Specific Mode: APP

Application [✓ Component]

Select the Permission Filter: ALL

Application level analysis Result

1. Application X
1.1 Component 1
Explicit Use
-- Perm 3
Implicit Use
-- Perm 5
1.2 Component 2

2. Application Y
2.1 Component 1
Explicit Use
-- Perm 4
Implicit Use

3. Application Z
3.1 Component 1
Explicit Use
Implicit Use

Color Significance: REQUIRED MAY BE REQUIRED UNUSED MAY BE MISSING MISSING

Figure 58: Result Screen for Level 2b Analysis (Detail Level: Component)

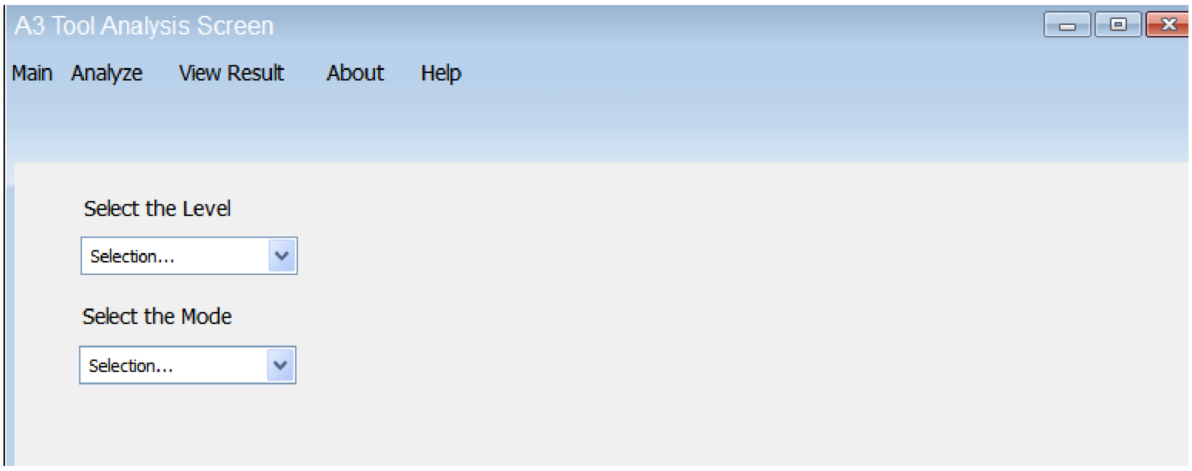


Figure 59: WorkFlow1 for Level 1 Summary Mode

5.2 Workflow

This section describes the navigation within and between the screens of our tool.
Example: Analysis for Level 1 in SUMMARY mode.

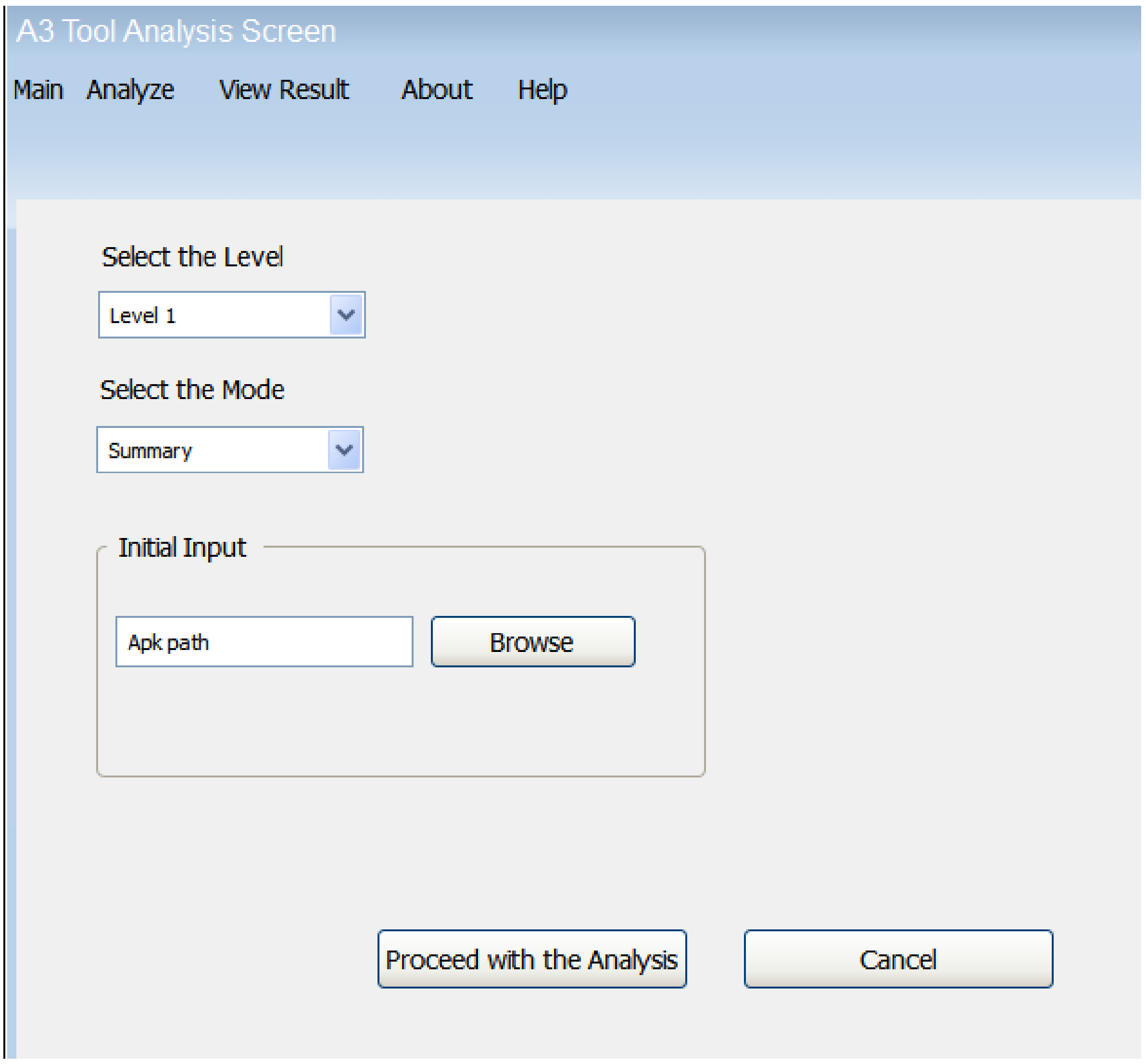
Step 1: After clicking on the *Analyze* menu on the Menu bar of the `Main Page`, `Analysis Screen` will appear with *Select the Level* and *Select the Mode* dropdown list. See Figure 59

Step 2: Then user will choose Level 1 as level and SUMMARY as mode from the available values in the dropdown lists. After the selection of mode, all the necessary elements required for the execution of the analysis will appear. In this case, *Initial Input* group box, *Proceed with Analysis* button, *Cancel* button. See Figure 60

Step 3: Then user will choose the path of an apk file for the analysis by clicking on the *Browse* button in *Initial Input* group box. After the path is selected, user will click on the *Proceed with Analysis* button.Then analysis will start and *Rotating Circle* will be shown till the time analysis is going on. See Figure 61

Step 4: After the analysis is done, *Circle* will be disappeared and *Result Representation* group box will appear with two link buttons: *Save only*, *View Result*. See Figure 62

Step 5: If the user clicks *Save only* link button, our tool will ask the user to choose the path where the result needs to be saved and upon selection of the path, the result file will be saved with a message that result saved successfully. Then user will be redirected to the `Main Page` automatically. See Figure 62



Select the Level

Level 1

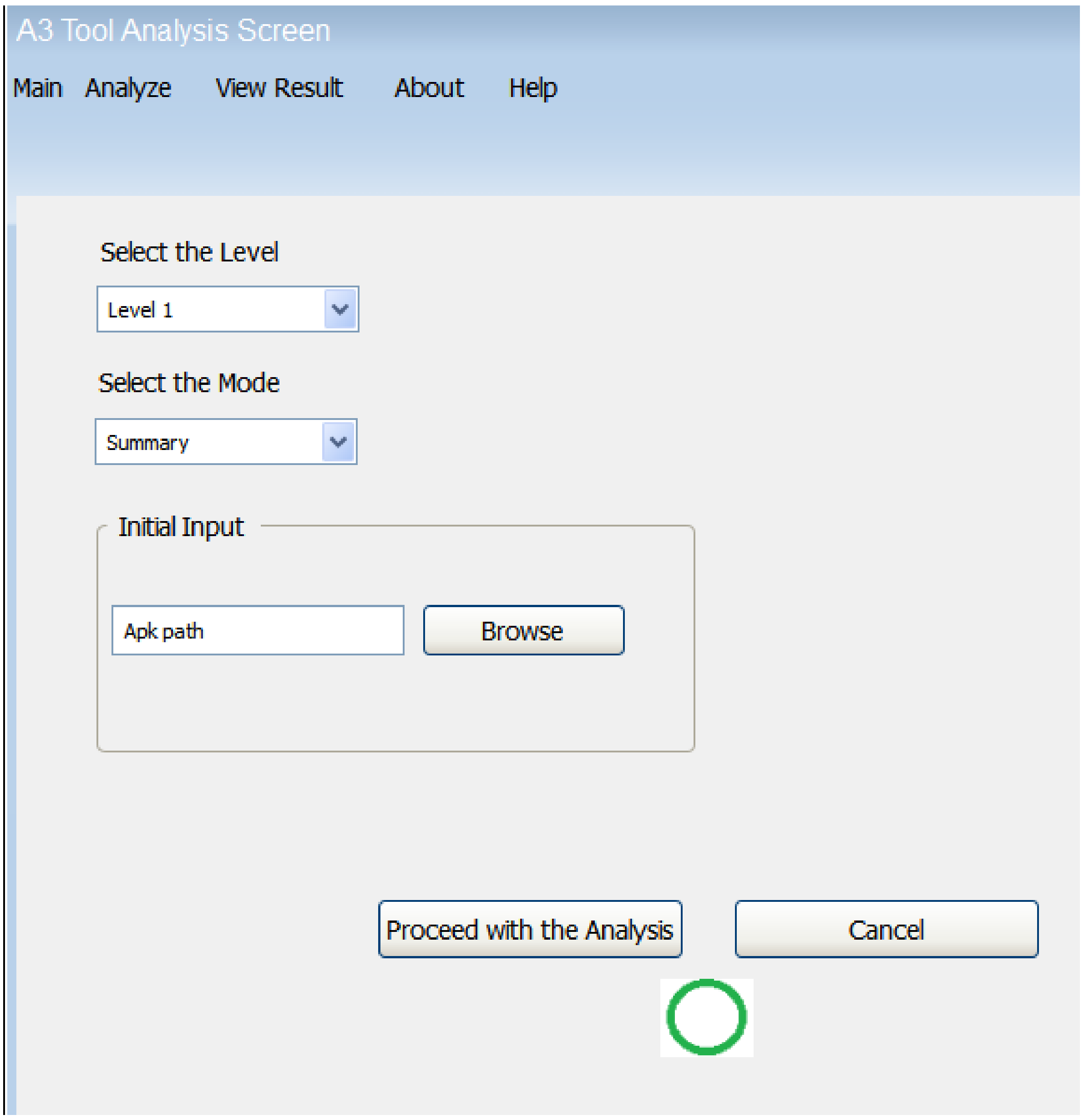
Select the Mode

Summary

Initial Input

Apk path

Figure 60: WorkFlow2 for Level 1 Summary Mode



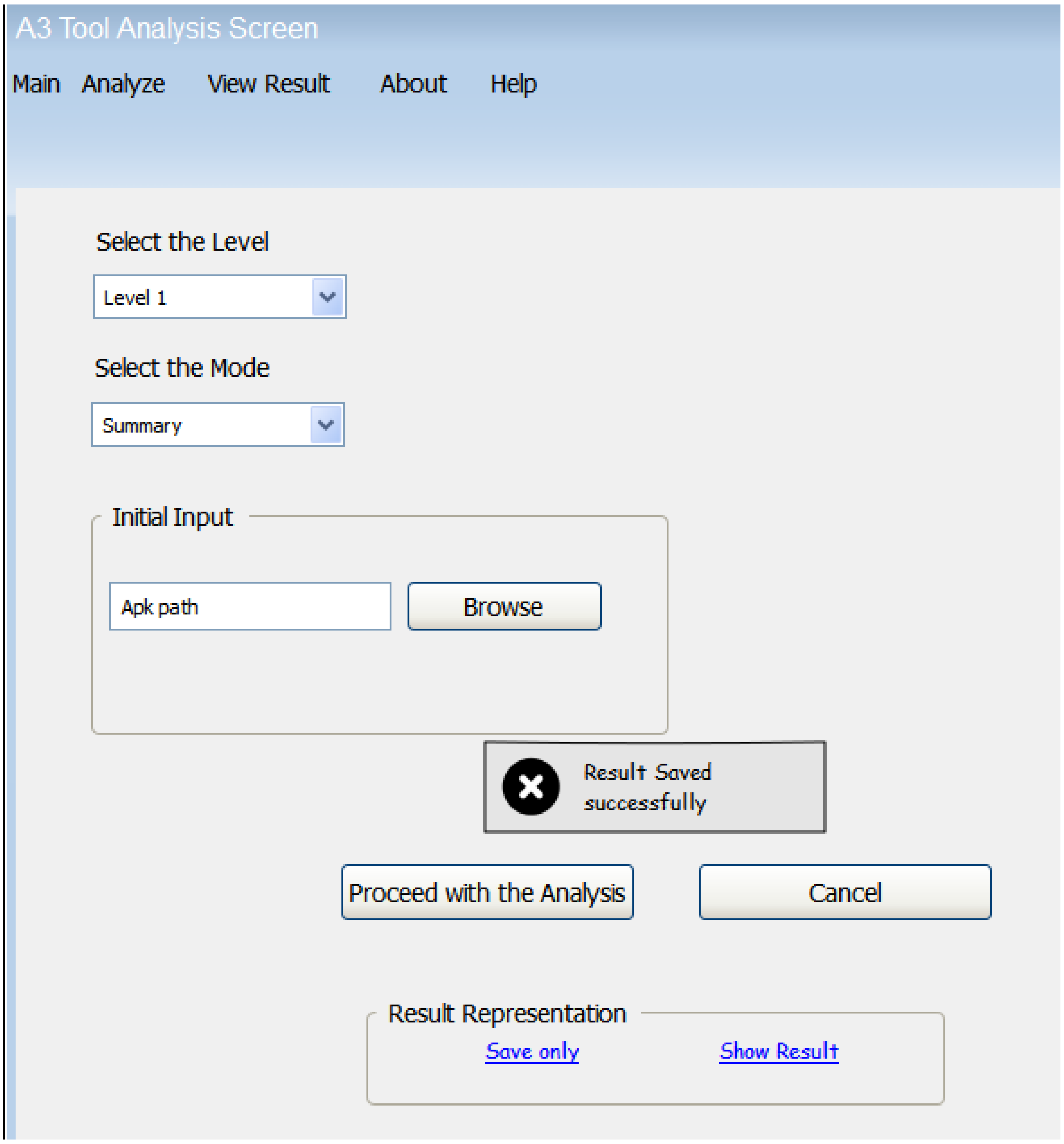
Select the Level: Level 1

Select the Mode: Summary

Initial Input

Apk path

Figure 61: WorkFlow3 for Level 1 Summary Mode


A3 Tool Analysis Screen

Select the Level: Level 1

Select the Mode: Summary

Initial Input

Apk path

Result Saved successfully

Figure 62: WorkFlow4 for Level 1 Summary Mode

Step 6: But if the user clicks on the *View Result* link button then the `Result Screen` of the corresponding analysis will be shown as described in Section 5.1 under GUI descriptions(Level 1 Result Screen). See Figure 51

The work flow steps described above is applicable to all the level of analyses, only the input elements changes as per the different combination selected in level and mode.

5.3 Command Line

Command Line Configuration Our tool can also be run from Command Line mode with all the configurations and almost all the features which are available in GUI mode. Basically, in command line mode we will be running the jar file of our application. So the command for running the jar file using the command prompt is,

- **java -jar <jar-file-names>.jar <ClassName>**

Above command will only initialize our tool and ask the user to provide inputs to start the analysis.

- **java -jar <jar-file-names>.jar <ClassName> <inputs>**

Above command will initialize our tool and start the analysis as well.
In the above command only <inputs> will vary depending upon the type of analysis the user wants to perform otherwise the remaining command will be same for all the configurations.

<jar-file-names>.jar represent as our application and its dependent jar files which we are going to use for developing our tool. For example, **a3-analysis.jar;soot.jar**

<ClassName> represents the class name(entry point for Command line mode) along with the full package name. For example,**'com.upb.a3-analysis.testclasses.Main'**

<Inputs> represents the actual values needed for the analysis to perform. Input parameters differs from for each level of analysis and result representation.

To distinguish between the input parameter, our tool allows distinct prefix keyword for each parameter. So the user has to provide the keyword first followed by the actual value of the parameter. With these keywords, our tool can easily differentiate between parameters.

Parameters

1. **Level of Analysis:** This parameter accepts the name of the level for which the user wants perform the analysis. The actual value will be provided following the keyword.
   Keyword: –l or –level
   Example:"–l 1", for Level 1 analysis
   "–l 2a", for Level 2a analysis
   "–l 2b", for Level 2b analysis

2. **Mode of Analysis:** This parameter accepts name of the mode in which the user wants perform the analysis. The actual value will be provided following the keyword.
   Keyword: –m or –mode
   Example: ”–m sum”, for SUMMARY mode
   ”–m comp”, for COMPARISON mode
3. **Level specific Mode:** This parameter accepts name of the mode specific to the level for which the user wants perform the analysis. This parameter is required only for Inter-App level(Level 2b). The actual value will be provided following the keyword.
   Keyword: –lm or –levelmode
   Example: ”–lm app”, for APP level analysis
   ”–lm all”, for ALL level analysis
4. **Initial Input:** This parameter accepts name and path of the initial input file(s) (Apk file(s)) for which the user wants perform the analysis. The user has to provide the path first and then the filename. The actual value will be provided following the keyword.
   Keyword: –i or –input
   Example: ”–i d:\alex: game.apk”
   ”–i d:\alex: game1.apk,game2.apk..”, only for Level 2b analysis in ALL mode
5. **Input for Comparison:** This parameter accepts name and path of the file(previously saved analysis result file) with which the comparison has to be done in COMPARISON mode. The user has to provide the path first and then the filename. The actual value will be provided following the keyword.
   Keyword: –ci or –compareinput
   Example: ”–ci d:\alex: game.extension”
6. **Non-Native Apps:** This parameter accepts name and path of the file(s) which will create the environment for the Inter-App(Level 2b) analysis. This can be a set of apk files. For providing multiple Apk files, all the input files should be kept in one folder, the user has to provide the path first and then the filename(s). The actual value will be provided following the keyword.
   Keyword: –nn or –nonnative
   Example: ”–nn d:\alex: aoe.apk,cs.apk..”
7. **Result Representation:** This parameter accepts what the user wants to do with the result that has been generated after the analysis is done. The actual value will be provided following the keyword.
   Keyword: –r or –result
   Example: ”–r save”, only save the result
   ”–r view”, show the results in different ways
8. **Result View:** This parameter accepts how the results will be shown if the user wants to view the result. So this parameter will be compulsory if the user chooses to view the result. The actual value will be provided following the keyword.
   Keyword: –v or –view

Example: ”–v text”, show the results in textual mode
”–v graph” , launch the GUI and show the results in graphical mode

9. **Confirmation for Analysis:** This parameter will be shown after the comparison message appears on the screen in case of Level 1 or Level 2a analysis in comparison mode. Then the user will be asked to enter YES/NO in order to proceed with the analyis or to cancel the analysis.
   Example: Would you like to proceed with the analysis? YES/NO

10. **Result filters:** These parameters will be shown after the textual result appears on the screen.The set of filter parameters differs for each level of analysis.

    - **Level 1 and Level 2b**

      a) *Detail level Filters*, These filters will re-arrange the result into different detail level i.e *application level, component level, class level, method level*. Only those option will be shown for which the results are available. By default *application level* result will be shown.

      b) *Permission Filters*, These filters will allow the user to categorize between different set of permissions i.e ALL, Required, May be, Unused, May be missing, Missing.By default ALL permissions will be shown.

      Example Would you like to filter the result? YES/NO
      Available filters, Detail Level Filter: a or application, c or component, cl or class, mt or method and Permission Filter:al or all, req or required, mb or maybe, un or unused, mbm or maybemiss, mi or Miss
      Enter the filters.
      Detail Level Filter:
      Permission Filter:

      After the required filters are entered, result will be re-arranged and shown. Filter options will be shown to the user till the user chooses YES.If the user chooses NO, the user will be asked to save the result.

      Would you like to save the result? YES/NO

    - **Level 2a**

      a) *Detail Level Filters*, These filters will re-arrange the result into different detail level i.e *Component Flow, Resource Flow, Statement to Statement*. By default *Component Flow* result will be shown.

      Example Would you like to filter the result? YES/NO
      Available filters, Detail Level Filter:c or component, res or resource, stm or statement
      Enter the filters.
      Detail Level Filter:

After the required filters are entered, result will be re-arranged and shown. Filter options will be shown to the user till the user chooses YES.If the user chooses NO, the user will be asked to save the result.

Would you like to save the result? YES/NO

Note: In Command line, filter for Source and Sink will not be available for Level 2a. As the number of sources and sinks may be high and showing all of them to the user for choosing will create problem.

6 Projectplan

After `Design Phase` (from June 1st to July 31st), the project continues with `Development Phase` (from August 2015 to February 2016) and ends with `Delivery` (on March 2016). The Figure 63 depicts all milestones (internal and external) as well as all sub-phases and the final delivery. Those sections belows will describe in detail the schedule and the goal of the `Development Phase` and the `Delivery`.

6.1 Development Phase (August 2015 - February 2016)

In general, the `Development Phase` is divided mainly into five sub-phases with **three external milestones** and **two internal milestones**. This phase starts on the first of August in 2015 and ends on the 28th of February in 2016.

The first sub-phase (Phase 1) is from 1st August to 3rd September. The goals of this phase are constructing the skeleton (or framework) with the prototypes for running in CMD-Line mode, creating a pseudo `EnhancedInput` to process for only the textual result of Level 1. The framework does not include the `ResultStorer/ResultLoader` or the `Enhancer` component. These components will be implemented in next phases. In addition the constructed `EnhancedInput` consists of TestCase based on the result definition of Level 1.

As soon as the **Phase 1** has started, the implementation for Level 2a and GUI also need to start because the workload of Level 2a and the GUI are very high. The implementation for Level 2a takes 135 days. It finishes at exactly when the **Phase 3** finishes. The implementation for GUI takes 69 days and it is done at the same time with the finish of **Phase 2**.

The first external milestone (External Milestone 1) with the presentation for the first phase is ready on the 4th of September. In the first talk, all features already obtained in **Phase 1** will be shown. Additionally, the trouble during the phase as well as feedback should be given for discussion in the meeting.

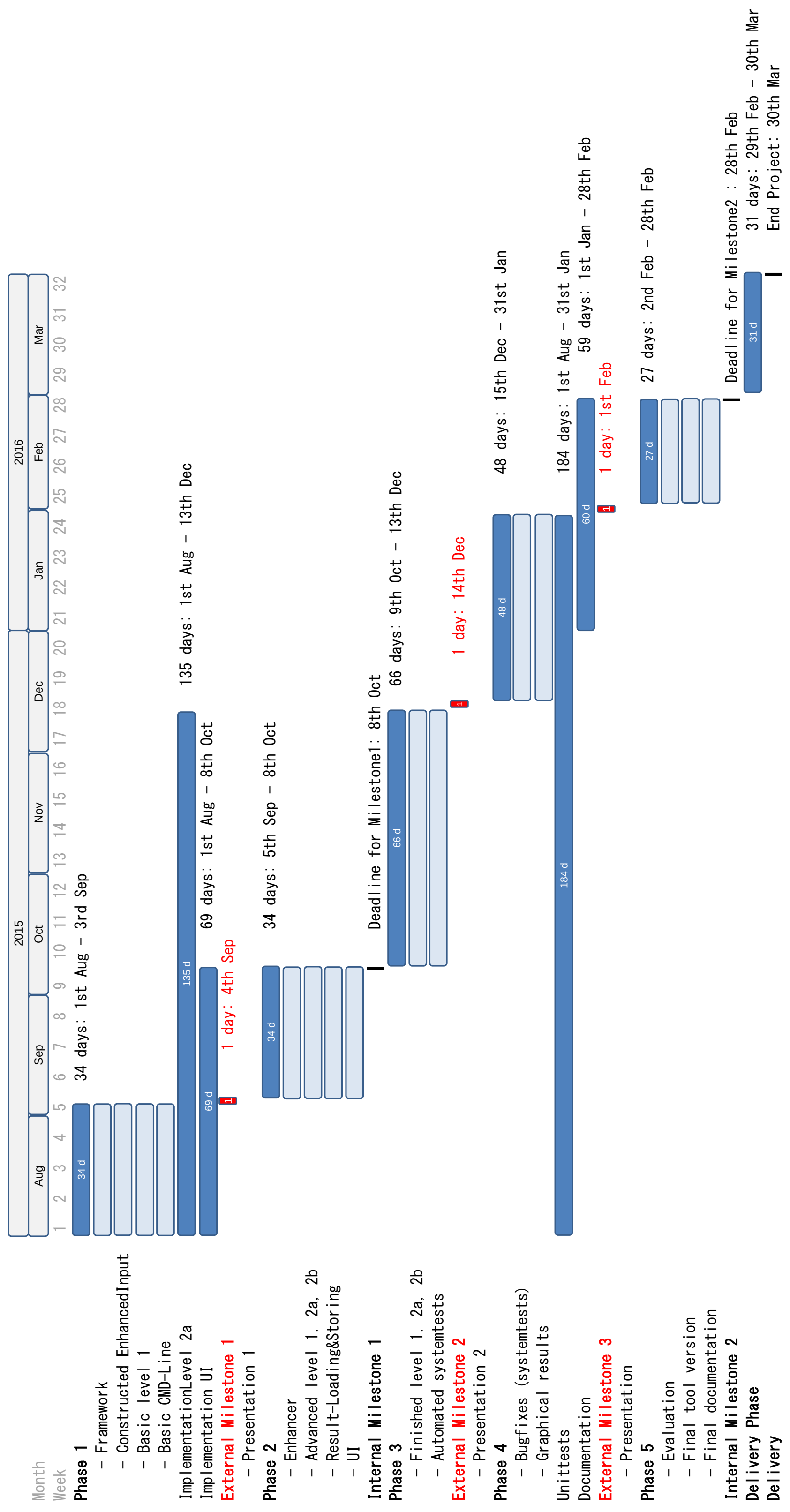


Figure 63: A3 Project Plan

The second sub-phase (Phase 2) then continues and lasts until the 8th of October. This phase focuses on the main processing of the application - the `Enhancer` whose responsibility is to generate the `EnhancedInput` from an `.apk` file. The implementation of SUMMARY mode corresponding to the three levels (Level 1, 2a and 2b) is carried out in this phase. In addition, the remaining components - `ResultStorer/ResultLoader` are also done. This phase ends with almost all features of GUI which already started at the beginning of the `Development Phase`. However the **Graphical Result Representation** function is left for the next phase. The end of current phase (**Phase 2**) is also the deadline for the **Internal Milestone 1**.

The third sub-phase (Phase 3) finishes all the implementations left. This phase has explicitly done the full analysis in SUMMARY and COMPARISON mode of the three levels. However the result is still only textual on CMD-Line and GUI. Automated System Tests are carried out also in this phase for all functions of the application. This phase takes 66 days to finish from the 9th of October to the 13th of December.

The second external milestone (External Milestone 2) takes place on 14th December. The application with full features is shown in the presentation. It is accepted for some bugs and the limitation of current implementation will be taken into concern for discussion. All bugs and limitation will be checked again in the next phase.

The fourth sub-phase (Phase 4) The **Graphical Result Representaion** function will be implemented. Concurrently, this phase also performs system tests and bug fixing (if any). This phase finishes in 48 days from 15th December 2015 to 31st January 2016. Later in the middle of this phase, the document implementation is started on the 1st of January in 2016.

The **UnitTests** start from the very beginning of `Development Phase` (from 1st August 2015). It is performed during the implementation of all features of the application and lasts until the end of the **Phase 4** on 31st January 2016.

The third external milestone (External Milestone 3) shows the complete application with full functionalities (including the `UnitTests`). The presentation takes place on the first of February in 2016.

The last sub-phase (Phase 5) consists of the evaluation of the application and the final documentation. This phase verifies the application to make sure that it is fully matching with the requirements in the `target-level-agreement` document. There are 27 days for this phase from 2nd to 28th February and it is also the last **Internal Milestone** for the `Development Phase`.

6.2 Delivery (March 2016)

From the beginning of March 2016 to the 31st of March, the final version of application as well as the final document will be delivered to the customers. Then the customer will give a specific day for final presentation. During this phase, feedback from the customer for the document will be taken into account. The final day for the whole deliveries is the 31st of March, namely the end of the project.

List of Figures