Introduction

StaticAnalyses is a subproject of Indus\(^1\) which provides static analyses that can be used in other program analyses and program transformations. At present it provides the following analyses and framework.

- **Value Flow Analysis (VFA) framework** provides a generic framework that can be used to implement analyses that have data flow flavour. **Object Flow Analysis (OFA)** is an instance of this framework that calculates points-to information.

- **Escape analysis** calculates "thread escaping" information for objects of a given system.

- **Dependence analysis** calculates the information pertaining to various notions of dependence for a given system.

The next section will outline the packages and their prime purpose. This will be followed by a detailed section on each of the above mentioned analyses.

Packages

The following is a list of packages in StaticAnalyses subproject along with their description. All the packages are rooted in a package called `edu.ksu.cis.indus`. All classes in the packages listed below with DOCUMENT ME! is better left unused as it does not have documentation and may be it is incomplete! Likewise any package not mentioned here is clearly already on its way out.

```
staticanalyses
```

This package contains classes that are common to various analyses. In general, if some class is used in more than one analysis and it is general then it will exist here before being moved into a
specific package. However, this movement will not be visible to the user.

`staticanalyses.cfg` This package will house the classes that contain logic and functionality pertaining to control flow analysis. This is usually uninteresting by themselves but may be interesting in more than one analysis more than once. For example, there are various information required by analysis that can be calculated just based the control flow graph of the method(s). This is provided by `CFGAnalysis` in this package.

`staticanalyses.concurrency` This package contains classes that analyze the system to discover properties/information about it pertaining to its concurrent nature. It can contain packages that are concerned with a particular concurrency aspect.

`staticanalyses.concurrency.escape` This package contains classes that deal with analyses that calculate escape information pertaining to objects in the system.

`staticanalyses.dependency` This contains analyses that calculate dependence information as required primarily for the purposes of program slicing. However, there may be more variants of these analyses that will be included for the purpose of customization and adaptation.

`staticanalyses.flow` This package contains a framework that can be used to implement value flow analysis in the same flavour as data flow analysis. All instances of this framework are housed as subpackages of this package.

`staticanalyses.flow.instances.ofa` As mentioned earlier, object flow analysis is a points-to analysis for Java implemented as an instance of value flow analysis framework provided by the parent package. This is used primarily to calculate a precise call graph and conservative thread graph.

`staticanalyses.flow.instances.ofa.fi` This package contains the specialized instances of flow framework classes to enable flow insensitive object flow analysis.

`staticanalyses.flow.instances.ofa.fs` This package contains the specialized instances of flow framework classes to enable flow sensitive object flow analysis.

`staticanalyses.flow.instances.ofa.processors` This package contains classes which process the raw information calculated by OFA to create more accessible information such as call graphs and thread graphs.

`staticanalyses.flow.modes.insensitive` This package contains classes that are specific to the flow framework which can be used to alter perform "context" unaware analysis. This basically means that these classes can be used to render a particular aspect of the flow analysis insensitive like flow insensitive.

`staticanalyses.flow.modes.sensitive` This package contains classes that are specific to the flow framework which can be used to alter perform "context" aware analysis. This basically means that these classes can be used to render a particular aspect of the flow analysis sensitive like flow sensitive.

`staticanalyses.flow.modes.sensitive.allocation` This package contains classes that are specific to the flow framework which can be used to perform allocation site sensitive analysis. This is useful in particular when the flow analysis instance differentiates the summary for a field or method based of the
value taken on by the receiver or the primary.

This package contains classes that are specific to the flow framework which can be used to perform flow sensitive analysis.

This package contains interfaces that should be used to expose various static analyses implementation; this way new implementation can be plugged into a system that has been using a static analysis.

This package is the counterpart of the processing package in Indus subproject. It provides filters, processors, and controllers that are

## Value Flow Analysis

### Idea

Data flow analysis is a common form of program analysis. Usually each developer rigs an implementation of a flow analysis or uses a framework such as one in Soot based on flow sets to implement a flow analysis. When developing VFA, the flow analysis framework in Soot was not available (pre Soot 1.2 era), hence we rolled out our own flow analysis framework. The reason we made it a framework rather than a single purpose implementation was that we had in mind other analyses which can be cast as a flow analysis problem, hence, the framework could be used to avoid common work that occurs in such implementation.

Although VFA framework is a data flow analysis implementation framework, the representation of the analysis instance and the constraints or equations is simple as it is graph based. Each summary set is denoted as a node in the graph and the edges connecting the nodes provide the path along which the values flow in the graph. This simplicity has its cost which will be addressed over time. The basic idea is to walk the system from a given set of entry points to create a flow graph as required. At the same time values are pushed along the edges to instrument the flow. This means the flow graph can grow. However, when the graph and the flow stabilizes, the analysis is done.

### Implementation

The framework provides classes that are required to construct the graph, map program points to nodes/summary sets, manage the key/index used to by the previous mapping in way particular to the "context", and abstract implementations that encapsulate common behavior required or exposed by all instances of the framework.

We use the term variant to represent the information/summary set/instance of a program point in a particular context. The framework provides managers to manage the variants. They just maintain mapping from a syntactic entity in a particular context to its variant. The managers are extended to handled various sorts of syntactic entities such as fields, AST chunks, methods, etc. Likewise, the variants are extended too. The managers rely on the index managers to manage the index of the program points in a particular context. This demarcation enables one to just vary the index managers in an instance and vary the sensitivity of the analysis. The framework also provides walkers/visitors that extend the Soot visitors by providing the common methods required by these visitor instances to create the flow graph and enable the flow in the framework. It also provides connectors to provide the developer control over the direction of the flow.

The actual instrumentation of the flow is done by a work list algorithm. Hence, the framework defines the interface of the work object to be used by its instances to instrument the flow. It has been our experience that due to the type system in Java the flow needs to be regulated based on types. To this end the
framework provides a filter that filters values based on types.

The framework itself is represented by the class FA and it requires a ModeFactory instance that provides the instances of various components of the framework to construct the flow graph and instrument the flow.

As for implementing an instance of the framework, the developer needs to provide classes of the nodes in the graph and the visitors which are used to walk the system and trigger the creation of the graph. In the visitors, the developer creates the nodes accordingly and delegates to the parent implementation to plug these nodes together. If the developer is interested in controlling the direction of the flow, he/she can provide a suitable connector implementation. As for instantiating an instance of the framework, the user needs to provide a mode factory instance that can create the required components according to the demands of the framework. That's it!

Please read the next section for more details on how to use the framework to implement an analysis.

Object Flow Analysis - an instance

OFA is an instance of VFA. It uses VFA framework to track the flow of objects in a system. Each object allocation site is treated as an abstract object and its flow through the system is tracked. This information is invaluable in constructing the call graph of the system. Please refer to [RanganathMSThesis2002] for nitty gritty details about OFA.

All classes refered in this section are rooted in the package edu.ksu.cis.indus.staticanalyses.flow.instances.ofa.

OFA provides 2 pair of walkers/visitors. An expression level visitor and statement level visitor to operate in flow insensitive/sensitive mode. These rely on the instance of a IFGNodeConnector to connect various nodes during the flow graph construction depending on the sensitivity setting. Hence, while instantiating the framework, the prototypes of the connectors are also provided. When the visitors create the nodes, they also may associate a work piece with them. This work piece is like a hook to be executed when a value flows into the node. These work pieces inject work into the frameworks work list to instrument the flow. For instance, the primary of a field expression is associated with a work piece that will be executed when new values arrive at the primary and this execution will connect the variant of field corresponding to the new abstract object to be connected with the variant of the ast chunk representing the field. In short, work pieces connect the nodes in the graph on a need basis.

As mentioned before the major chunk of work is done while post processing the information calculated by OFA. This is done by classes in processors package. These classes are driven by the processing infrastructure of Indus to visit parts of the system to create high-level information based on the OFA information. call graph, thread graph, and new-expression-to-init-site-mapper are the existing processors that calculated call graph information for the system, the call graph for each thread along with threading hierarchy for the system, and a mapping of which <init> invocation is associated to which new expres-
sion/allocation site.

**Escape analysis**

*Escape analysis* detects if an object escapes the method in which it is created and/or if it escapes the thread in which it is created. We have implemented Ruf's analysis and an extended version of Ruf's analysis. It is the latter we support and it is used to detect if an object is thread escaping. It is based on the processing infrastructure in Indus. Hence, before executing the analysis, it needs to walk the system via the processing infrastructure. After the execution, the client can query about the escaping nature of a soot value in a method.

Please refer to [RanganathCC04] for details about the extension to Ruf's analysis.

**Dependences**

**Concept**

The concept of a program point affecting/affected by another program point is captured as dependences. Dependence can be thought of as a relation between two program points x and y that indicates if x depends on y. In a dependence relation between x and y where x depends on y, we refer to x as the dependent and y as the dependee.

There are many notions of dependences and *data* and *control* dependence are the most common and simple notions of dependences that can occur even in simple non-procedural sequential programs. In a simple setting *data dependence* indicates if the variable being read at a program point is influenced by another program point at which the same variable is being written. Similarly, *control dependence* indicates if the flow of control to a program point is dependent on another program point.

An important restriction on the above stated dependence was that there should be a control flow path between the program points being considered. Hence, these simple notions need to be extended to be applicable in an inter-procedural setting and it is usually achieved by inlining the methods, constructing a program control flow graph for the entire program [HorwitzPLDI88]. The notions can again be extended in a conservative/pessimistic way when dealing with programs that use concepts such as dynamic memory allocation and reference variables. However, these notions fail when dealing with concurrent programs that use concepts such as dynamic memory allocation and reference variables which is a common case in programs written OO languages such as C++ and Java as there is not control flow between program points in different threads. If one were to attempt to patch a control flow graph (CFG) that captures all possible interleavings of the program then the size of such a CFG can be exponential as the number of interleavings in a concurrent program can be exponential (in the worst case.) This limitation was identified and addressed by many [HatcliffSAS99] [KrinkePASTE98] by introducing new notions of dependences that were applicable to concurrent programs.

*Interference Dependence* is one such dependence. It can be defined as "if a variable x is written at a program point m and x is read at a program point n and m and n occur in different threads then n is said to be interference dependent on m"\(^2\). This dependence is an extension of data dependence in which the restriction of the existence of a control flow path is relaxed to capture data dependence between program points in different threads. Refer to [cite John, Krinke] for more detail about this notion of dependence. [HatcliffSAS99] and [KrinkePASTE98] provide details about this dependence.

Monitor-related constructs such as `java.lang.wait()`/`java.lang.notify()`/`java.lang.notifyAll()` in Java introduce a new notion of dependence between program points in different threads as the completion of execution of a wait statement is dependent on a notify statement (ignoring exceptional completion). Similar situation occurs between `enter monitor` and `exit monitor` program points. This dependence is similar to control dependence except that it does not impose the restriction of existence of a control flow path and it is concerned with the control leaving the dependent program point rather than reaching it. This notion

\(^2\)Note that this definition is not precise.
of dependence is defined as *Ready Dependence*.

Another form of ready dependence that is more similar to control dependence (as it relies on a control flow path) relates the statements reachable in a CFG from statement containing `enter_monitor` or calls to `java.lang.Object.wait()`. The dependence captures the requirement of the completion of the dependee statement for the control to reach the dependent statement. This dependence captures divergence relation based on synchronization constructs unlike the conditional-based divergence dependence. These dependences are discussed in greater detail in [HatcliffSAS99]

### Implementation

All dependence analyses extend `DependencyAnalysis` class which provides the generic interface via which information from the analyses can be queried. Each analysis requires a set of information prior to starting execution. This is provided via a id to value mapping via `setup()` method. The id is based on notions of dependences and all implementations that provide info about a notion of dependence will have the same id. This is a system native to Indus project. This is followed by executing the analyses via an instance of `AnalysisController` as it considers any interdependencies between the analyses being executed to determine if all analyses have in fact completed. That's it, the analyses are ready to be queried.

### Closing Note

The XMLizing classes used by this project and it's parent and sibling projects use the xmlizing framework to drive the slicer. So, we urge you to peruse the source code of these classes before asking questions on the forum or the mailing list. We will be glad to answer any question you may have regarding the usage, but it probably would be faster if the user mocked an existing working piece of code while starting to use a new tool.

The reader is encouraged to use the modules as is or to extend them as required. In the due process, the users are urged to submit bug reports of any bugs uncovered with suitable information about the triggering input and configuration.

The interface of the modules are not fixed as the development team has not forseen all possible applications and tweaks to the slicer. Hence, the users are encouraged to raise change requests to the development team along with any feature requests they may have. However, please note that the development team may not be able to implement all requested features in which case they will assist by providing any information or alterations to enable the requested features.

Please refer to Indus [http://indus.projects.cis.ksu.edu] for more documentation, distribution, mailing list, forums, and links to other subprojects.

We hope you have a pleasant experience using our product.

### Bibliography


