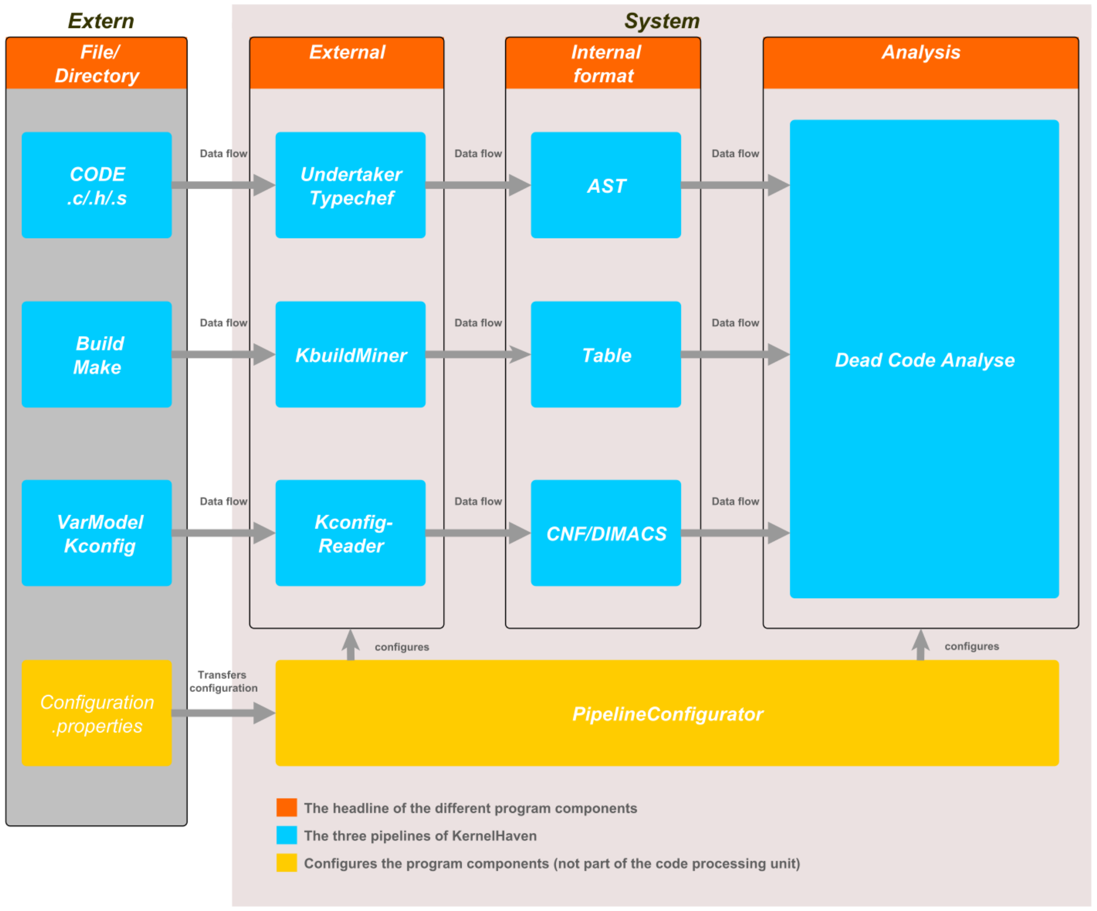
KernelHaven User Documentation

This document serves as a manual for users of KernelHaven. It will explain how to set up KernelHaven and execute an analysis. Refer to the developer documentation if you want to write new extractors or analyzes.

Concept

KernelHaven offers a generic infrastructure for performing different analyses on product lines. The architecture is shown in the figure below. The variability information from the product line is extracted via extractors. This is split into three pipelines: source code, build system and variability model. Each pipeline has an extractor that reads the necessary data from the product line, and a model in which the data is represented. Depending on the product line being analyzed, different extractors are used. This allows the infrastructure to adapt to different product lines, while still having a common data representation in the models.

KernelHaven has a plug-in system for easy replacement of extractors and analyses. Each plug-in is a Jar file placed in a central plugins folder. Based on the user configuration for an execution, different Java classes are instantiated at runtime, which serve as the extractors and analysis.



Setup and Execution

KernelHaven is written in Java, and thus does not depend on any specific operating system. The only requirement for executing KernelHaven is an installed Java VM (version 8 or greater). However, some extractors are not platform independent; they usually only run on Linux. Refer to the different plug-in descriptions for additional limitations and dependencies. KernelHaven and all extractors developed by us are tested to run at least on Ubuntu 16.04.

KernelHaven does not require any specific directory structure. All paths that are relevant for an execution can be configured. However, an installation of KernelHaven typically looks like this:

```
kernel_haven/
├── cache/
│   └── ...
├── log/
│   └── ...
├── output/
│   └── ...
├── plugins/
│   ├── cnfutils.jar
│   ├── kbuildminerextractor.jar
│   ├── kconfigreaderextractor.jar
│   ├── undeadanalyzer.jar
│   └── undertakerextractor.jar
├── res/
│   └── ...
├── kernel_haven.jar
└── dead_code.properties
```

The following list describes the usage of the different files and folders:

- The `kernel_haven.jar` file is an executable Jar file and contains the main infrastructure.
- The `dead_code.properties` file contains the configuration of a KernelHaven execution.
- The `cache/` folder will contain the cache files generated by the extractors. The user will usually not look into this directory (except maybe for manually clearing the cache).
- The `log/` folder will contain the log files generated by the infrastructure.
- The `output/` folder will contain the output files created by the analysis.
- The `plugins/` folder contains Jar files with plugins, that will be loaded by the infrastructure.
- The `res/` folder will contain resource files needed by the extractors. The extractors manage the contents of this themselves; the user does not have to look into this folder.

An example configuration file (`dead_code.properties`) for executing a dead code analysis on a Linux Kernel could look like this:

```
# Linux Source Tree
source_tree = /path/to/linux_source
arch = x86

# Analysis
analysis.class = net.ssehub.kernel_haven.default_analyses.DeadCodeAnalysis

# Code Extractor
code.provider.cache.read = true
code.provider.cache.write = true
code.extractor.class = net.ssehub.kernel_haven.undertaker.UndertakerExtractor
code.extractor.threads = 4

# Build Extractor
build.provider.cache.read = true
build.provider.cache.write = true
build.extractor.class = net.ssehub.kernel_haven.kbuildminer.KbuildMinerExtractor

# Variability Extractor
variability.provider.cache.read = true
variability.provider.cache.write = true
variability.extractor.class = net.ssehub.kernel_haven.kconfigreader.KconfigReaderExtractor

# Logging
log.console = true
log.file = true

# Directories
archive.dir = .
cache_dir = cache/
log.dir = log/
output_dir = output/
plugins_dir = plugins/

resource_dir = res/
```

This is a standard Java properties file. For a full list of possible configuration options, refer to the `config_template.properties` file in KernelHaven. Note that some extractors also have their own configuration options, in addition to the ones from the infrastructure. Relative paths are interpreted relative to the working directory from which KernelHaven is started; this can be different from the location of the `kernel_haven.jar` file.

To execute the infrastructure with the given configuration, start `kernel_haven.jar` and pass the configuration file as the only parameter:

```
java –jar kernel_haven.jar dead_code.properties
```

KernelHaven also supports archiving of an execution. If it is enabled, all relevant files are combined into a Zip archive after the execution finished. To enable archiving, either set the configuration option `archive` to `true` in the properties file, or start KernelHaven with the `--archive` parameter:

```
java –jar kernel_haven.jar --archive dead_code.properties
```

Note that it is possible to pass additional parameters to the Java call. For example, it is possible to set the maximum Java heap size, which may become necessary when dealing with a lot of data:

```
java –Xmx100g –jar kernel_haven.jar dead_code.properties
```