# Developer-Documentation KernelHaven

Arbeitsgruppe Software Systems Engineering • Institute for Information Technology
Universität Hildesheim • Universitätsplatz 1 • D-31141 Hildesheim

# Contents

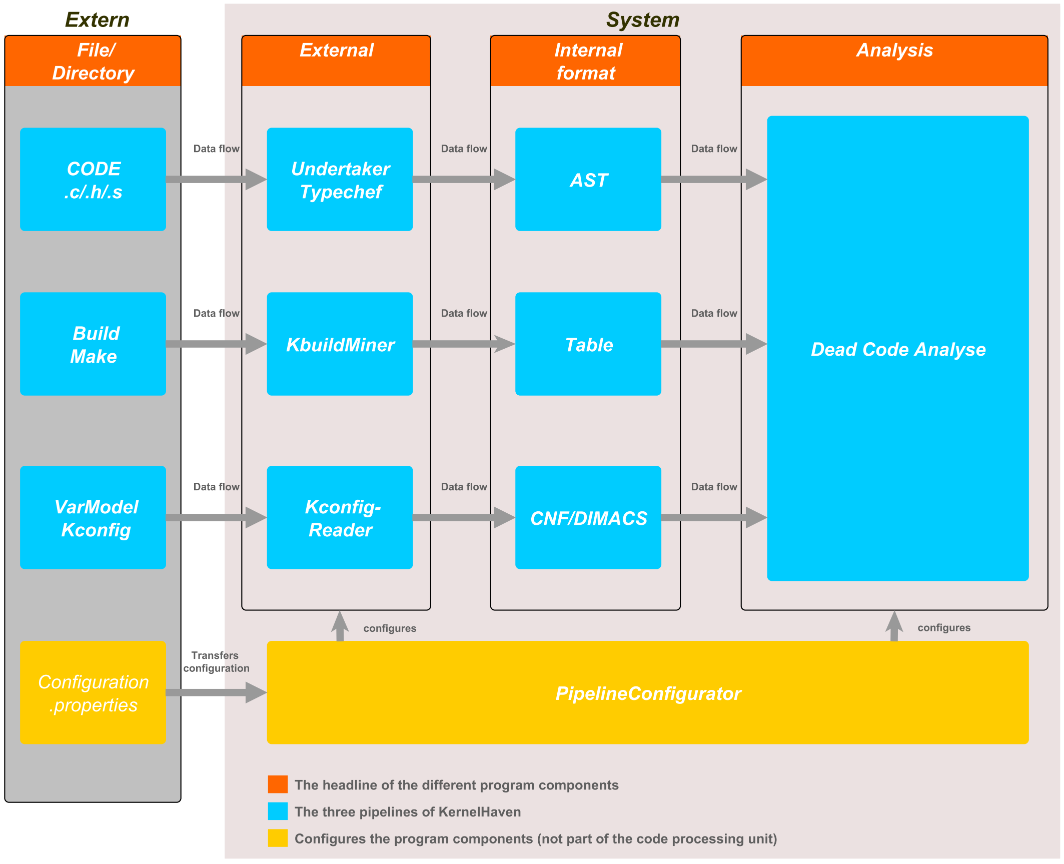
# List of Figures

# 1 Architecture Overview

This chapter gives a brief overview of the architecture.

## 1.1 Overview over the Architecture

Figure 1.1: Overview-Architecture of KernelHaven



In Figure 1.1 is an overview of the KernelHaven architecture, detailed information about how the execution works and about every component of the architecture are described below.

An execution works as follows:

After the pipeline is set up (see pipeline configuration below) , the analysis component starts the providers, which in its turn starts the extractors. Each extractor spawns a separate thread, so that the three pipelines and the analysis can run in parallel. When an extractor is finished, it tells its provider about the result. When the analysis queries the result, the provider waits until the result is present and returns it to the analysis. This way, the providers function as a mediator between the analysis and the extractor threads.

If an extractor encounters an exception, it passes an extractor exception to the provider. After that, each call from the analysis to the get function of the provider will throw this exception. This way the analysis is properly notified about exceptions.

If caching is enabled in the configuration, then the providers will first try to read the result from the cache. In this case, the extractors are only started if the cache does not contain the required data.

The pipeline is set up by the `PipelineConfigurator`. This is the component that is started when running `kernelhaven.jar` (i.e. it contains the main method). It reads the configuration file .properties, which is provided by the user. Every .jar file from a specified plugin folder is loaded into the JVM. This allows us to have the extractor and analysis components as pure runtime dependencies. The `PipelineConfigurator` then proceeds to instantiate the extractors, providers and the analysis based on this configuration. Each of these instantiated components also get the configuration passed into the constructor, so specific user configurations are available to them.

The extractors and analysis are instantiated via Java reflection, based on a fully qualified class name in the user configuration file. This, together with the dynamic plugin loading, allows for pure runtime dependencies: KernelHaven is completely agnostic to the concrete extractors and analysis that make up the pipeline at runtime.

An example of the working pipeline is shown in the Figure 1.2. The numbers in the red circles show the order of calling the methods. Please note that this figure is an example with the `kbuildminerextractor`. A wrapper[1] for the executable or a converter for the executable output is not necessary for all extractors. Also note that `iExtractor` is an shortcut for all interface extractors (e.g `iBuildModelExtractor`, see the next chapter for a list of all extractor interfaces )

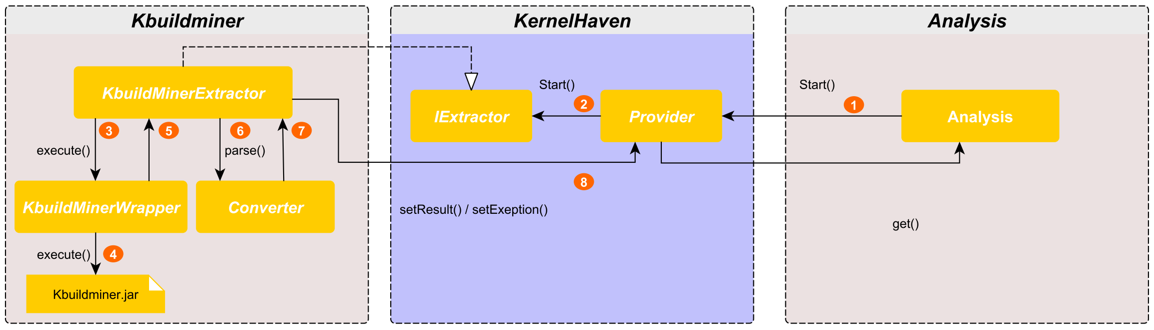


Figure 1.2: The process of the pipeline

[1] wrapper = java code that calls the external tool. This is used for command line tools and tools that are not written in java.

# 2 Writing a new Extractor

## 2.1 Overview

An extractor class has to implement the following Methods:

- Implement `IBuildModelExtractor`, `IVariabilityModellExtractor` or `ICodeModelExtractor` with the following methods:
  - **`setProvider()`** is called in the setup phase to tell the extractor which provider it should send its results to.
  - **`start()`** is called by the Provider to signal the extractor that it should start its extraction process. The extraction process should run in a seperate thread.
  - **`stop()`** is called by the Provider to signal that the extraction process took too long and the extractor should abort its extraction process.

The extractor has to use one of the following methods to signal the Provider its result. Those methods need to be called directly on the provider instance that was passed via the `setProvider()` method.:

- **`setResult()`** can be called to give the extractor the result of the extraction process. This must not be called after the provider called `stop()` (see above).
- **`setException()`** can be called instead of `setResult()`, to tell the Provider that the extraction process has terminated with an exception. The exception passed to it will be thrown in the next `getResult()` call that the analysis does on the provider.

If none of these methods is called within a user configured timeout, then the provider will kill the extractor via the `stop()` method.

There also needs to be a factory for creating an instance of the extractor for a given configuration. This factory needs to implement `IBuildExtractorFactory`, `ICodeExtractorFactory` or `IVariabilityExtractorFactory`. This interface defines the following method to be implemented:

- The **create()** method gets the configuration for the extractor and should return an instance of it. This is called via reflection when the analysis starts the extraction process. This method may throw a `SetupException` if it detects any invalid configuration options.

## 2.2 Example

Here is an example implementation of a build model extractor:

```java
import de.uni_hildesheim.sse.kernel_haven.SetUpException;
import de.uni_hildesheim.sse.kernel_haven.build_model.BuildModel;
import de.uni_hildesheim.sse.kernel_haven.build_model.BuildModelProvider;
import de.uni_hildesheim.sse.kernel_haven.build_model.IBuildModelExtractor;
import de.uni_hildesheim.sse.kernel_haven.config.BuildExtractorConfiguration;
import de.uni_hildesheim.sse.kernel_haven.util.ExtractorException;

/**
 * A simple dummy build extractor.
 *
 * This expects the build_extractor.throw_exception to decide whether to
     throw an exception
 * or return an empty build model.
 */
public class DummyBuildExtractor implements IBuildModelExtractor, Runnable {

    private boolean throwException;

    private BuildModelProvider provider;

    private boolean killed;

    public DummyBuildExtractor(BuildExtractorConfiguration config) throws
        SetUpException {
        String exceptionProp =
            config.getProperty("build_extractor.throw_exception");
        if (exceptionProp == null) {
            throw new SetUpException("No build_extractor.throw_exception
                setting");
        }

        throwException = Boolean.parseBoolean(exceptionProp);
    }

    @Override
    public void start() {
        Thread th = new Thread(this);
        th.setName("DummyBuildExtractor");
        th.start();
    }

    @Override
    public void stop() {
        killed = true;
    }

    @Override
    public void setProvider(BuildModelProvider provider) {
        this.provider = provider;
    }

    @Override
    public void run() {
        if (throwException) {
            if (!killed) { // make sure that we are not stop()ed yet
                // "throw" an exception
                provider.setException(new ExtractorException("Dummy exception
                    specified in config"));
            }


        } else {
            if (!killed) { // make sure that we are not stop()ed yet

                // just return an empty build model
                provider.setResult(new BuildModel());
            }
        }
    }

}
```

This extractor has a configuration property that either tells it to return an empty `BuildModel` or throw an `ExtracorException`.

The following factory can be used to instantiate this extractor:

```java
import de.uni_hildesheim.sse.kernel_haven.SetUpException;
import de.uni_hildesheim.sse.kernel_haven.build_model.IBuildExtractorFactory;
import de.uni_hildesheim.sse.kernel_haven.build_model.IBuildModelExtractor;
import de.uni_hildesheim.sse.kernel_haven.config.BuildExtractorConfiguration;

public class DummyBuildExtractorFactory implements IBuildExtractorFactory {

    @Override
    public IBuildModelExtractor create(BuildExtractorConfiguration config)
        throws SetUpException {
        return new DummyBuildExtractor(config);
    }

}
```

To compile this in eclipse:

- Create a new project
- Add the kernelhaven.jar with source attachments to the build path
- Insert the two classes from above
- Export the project as a jar archive (not as a runnable jar)

To run this extractor in the infrastructure, place the jar file in the plugins folder and set the factory at the build model extractor in the properties (see user documentation for details on this).

# 3 Writing a new Analysis

## 3.1 Overview

An analysis class has to implement the following Methods:

- Extend the abstract `AbstractAnalysis` class. The analysis has to implement the following abstract methods of it:
  - **`run()`** is called by the infrastructre after the pipeline is set up. This should execute the analysis.
- A **constructor** which takes a `Configuration` object. This is called via reflection in the setup phase to instantiate the analysis. The configuration object passed to it is the user configuration. This constructor may throw a `SetupException` if it detects any invalid configuration options.

The analysis can use the following API:

- The variables **`vmProvider`**, **`bmProvider`** and **`cmProvider`** give it access to the providers. These first need to be started via their `start()` methods. The configurations required by the `start()` methods can be derived from the configuration passed to the constructor.

  After the start methods have been called, the extractors run asynchronously. The result can be queried via the `getResult()` methods; this method blocks until the result is ready (or an `ExtractorException` is thrown).
- **`createResultStream()`** can be called to create a stream to a file to write the result of the analysis to. The infrastructure takes care that the result file is created in the correct output directory.
- The variable **`LOGGER`** can be used to log progress, etc.

To keep the analysis re-usable by other analyses, it should contain a public method that does the main analysis. The `run()` method should only start the providers and call this analysis method.

## 3.2 Example

The following example contains a `doAnalysis()` and a `run()` method which may seem redundant initially. However this setup can be beneficial when one analysis uses components of another analysis. While the `run()` method starts the providers that does not need to be done when called from another analysis as that part has already be taken care of by the calling analysis. Instead the calling analysis may use `doAnalysis()` and thereby only run the analysis-part without starting any further extractor-processes.

```java
import java.io.PrintStream;

import de.uni_hildesheim.sse.kernel_haven.SetUpException;
import de.uni_hildesheim.sse.kernel_haven.build_model.BuildModel;
import de.uni_hildesheim.sse.kernel_haven.config.Configuration;
import de.uni_hildesheim.sse.kernel_haven.util.ExtractorException;

public class DummyAnalysis extends AbstractAnalysis {

    public DummyAnalysis(Configuration config) {
        super(config);
    }


    public int doAnalysis(BuildModel bm) {
        return bm.getSize();
    }

    @Override
    public void run() {
        try {
            bmProvider.start(config.getBuildConfiguration());
            vmProvider.start(config.getVariabilityConfiguration());

            int result;
            result = doAnalysis(bmProvider.getResult());

            PrintStream out = createResultStream("result.txt");
            out.println("" + result);
            out.close();
        } catch (ExtractorException e) {
            LOGGER.logException("Error reading build model", e);
        } catch (SetUpException e) {
            LOGGER.logException("Error in configuration", e);
        }

    }

}
```

To compile this in eclipse:

- Create a new project
- Add the kernelhaven.jar with source attachments to the build path
- Optional: Add other dependencies (like CnfUtils) to the build path
- Insert the analysis class from above
- Export the project as a jar archive (not as a runnable jar)

To run this analysis in the infrastructure, place the jar file containing the analysis class in the plugins folder. Then set the analysis setting in the properties file to the fully qualified class name of your analysis (see the user documentation for details on this).

## 3.3 Utility Plugins for Analyses

CnfUtils is a project distributed with every release of KernelHaven as binary-jar as well as source code. It contains common operators for handling cnf. Refer to the javadoc/source-code for detailed information.

CnfUtils-Overview:

**Cnf**

Cnf is a storage class for storing boolean formulas in conjunctive normal form.

**SatSolver**
This class is used to check if an instance of `Cnf` is satisfiable.

**FormulaToCnfConverterFactory**
This factory can create converters which convert a boolean `Formula` to `Cnf`.

**VmToCnfConverter**
Converts a variablity model with a constraint model file in `DIMACS` to a `Cnf` object.