Commit Analysis Infrastructure (ComAnI)

Guide

November 17, 2018

Christian Kröher
kroeher@sse.uni-hildesheim.de

Implemented in: Java, Version 1.8.0_181
Used on: Windows, Ubuntu
Licensed under: Apache License, Version 2.0
Usage of external libraries (infrastructure): none
Usage of external libraries (non-third-party plug-ins): none

Software Systems Engineering • Institute of Computer Science
University of Hildesheim • Universitätsplatz 1 • 31134 Hildesheim • Germany

Acknowledgment

This work is partially supported by the ITEA3 project REVaMP[2], funded by the BMBF (German Ministry of Research and Education, https://www.bmbf.de/) under grant 01IS16042H. Any opinions expressed herein are solely by the author(s) and not by the BMBF.

A special thanks goes to the developers of KernelHaven [9, 3, 4]: Adam Krafczyk, Sascha El-Sharkawy, Moritz Flöter, Alice Schwarz, Kevin Stahr, Johannes Ude, Manuel Nedde, Malek Boukhari, and Marvin Forstreuter. Their architecture and core concepts significantly inspired the development of ComAnI. In particular, the mechanisms for file-based configuration of the infrastructure and the plug-ins as well as loading and executing individual plug-ins are adopted in this work.

Contents

List of Figures

Listings

List of Tables

1 Introduction

The Commit Analysis Infrastructure (ComAnI) is an open and configurable infrastructure for the extraction and analysis of commits from software repositories. For both tasks, individual plug-ins realize different extraction and analysis capabilities, which rely on the same data model provided by the infrastructure. Hence, any combination of extraction and analysis plug-ins is possible. For example, we could first conduct an analysis for a software hosted in a Git repository [2] and later conduct the same analysis for a different software hosted by SVN [1][1]. Another example is to use the same commit extractor, e.g., supporting the commit extraction from Git repositories, for different analyses. The definition of a particular ComAnI instance consists of a set of configuration parameters saved in a configuration file, which the infrastructure reads at start-up. Hence, there is no implementation effort needed. The infrastructure automatically performs its internal setup, loads and starts the desired plug-ins.

ComAnI represents a large increment of the ComAn toolset [10]. This toolset uses a single commit extraction script and a Java-based implementation of a particular commit analysis [6, 7, 5]. Further, the toolset is designed to be applied to the Linux kernel [8] and its Git repository [11]. This design of ComAn restricts its applicability to other software and repository types. While it is not completely impossible to adapt it to other inputs, this adaptation requires mayor implementation effort. Hence, we decided to create a complete infrastructure, which realizes a flexible and highly configurable ecosystem for conducting a variety of analyses by means of plug-ins for commit extraction and their analysis.

This guide consists of three parts. The first part in Section 2 introduces ComAnI in more detail and describes the concepts realizing the core features of the infrastructure. Section 3 represents the second part, which focuses on the end user of ComAnI. We describe how to download, install and execute the core infrastructure as well as the available plug-ins. As part of the execution, we also discuss the configuration parameters and the definition of particular ComAnI instances. The third part of this guide focuses on the developers. In Section 4, we discuss the development of new extraction and analysis plug-ins by examples.

[1] Assuming that the analysis is able to cope with the artifacts and their technologies of the new software under analysis.

2 Overview

ComAnI is designed to support the extraction of commits from different version control systems and various analyses of those extracted commits in any combination. For this purpose, it offers an open plug-in infrastructure implemented in Java. The core components of this infrastructure are the commit extractor, the internal data model, the commit analyzer, and the configuration file as illustrated in Figure 1. In this section, we will describe these components in detail.

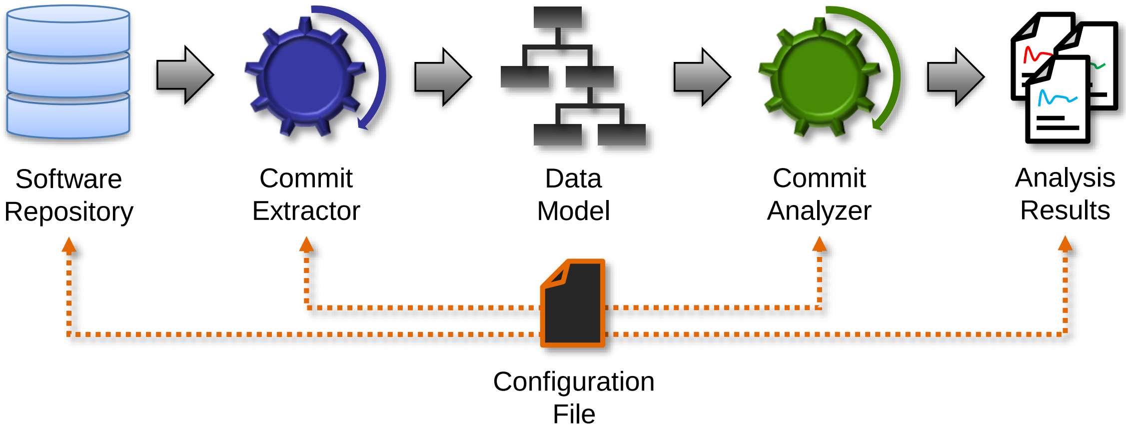

Figure 1: ComAnI overview

A **commit extractor** is a ComAnI plug-in, which is responsible for the extraction of commits and their provision for the commit analysis as elements of the internal data. A commit extractor typically supports three extraction variants[2] depending on the given sources from which the commits shall be extracted:

- *Full repository extraction*: this variant forces the extraction of all commits of a software repository. This requires the definition of the location of the target repository as part of the configuration file.
- *Partial repository extraction*: instead of extracting all commits of a software repository, this variant allows the extraction of a predefined set of commits. Besides the location of the target repository, this requires the specification of an additional file, which contains a list of unique commit numbers (or hashes). Each line of this commit list file must contain exactly one commit number. Further, the author of the commit list file must ensure that the commit numbers specify commits of the target repository.
- *Single commit extraction*: the third variant offers an interactive mode, in which the content of a single commit can be passed on the command line as an input.

The available extractors are introduced in Section 3.1. Section 3.2 describes their definition for a particular ComAnI instance and the usage of the different extraction variants above. Further, Section 4.2 explains the development of new extractor plug-ins for custom commit extraction capabilities.

The internal **data model** represents the conceptual interface between commit extractors and commit analyzers. It offers two main elements for representing commits: the `Commit` itself, which provides information, like its id (the commit number) or date, and the `ChangedArtifact` for storing the information about the artifacts changed by a specific commit. Hence, each commit typically contains a list of changed artifacts, which in

[2]While the respective methods need to be implemented by each commit extractor, developers are free to realize the required algorithms. Hence, we cannot guaranteed that all extractors support all extraction variants. We recommend reading the description of the desired commit extractor for more information.

turn contain information about their name and location as well as their content including the changed lines. A commit extractor creates instances of these elements based on the extracted commits from a target repository or the content of a commit passed as command line input. Section 4.1 provides further details about the internal data model and its elements.

The elements of the internal data model are input to a **commit analyzer**, which is a ComAnI plug-in similar to a commit extractor. Depending on the core algorithm of the respective analysis, a commit analyzer may either wait until all commits are available or directly start processing at the time a commit is available. The infrastructure neither imposes any restriction on the way of processing nor on the analysis results. Hence, each commit analyzer has full control over its result creation. The only input it receives is an output directory in which the results can be stored. While the available analyzers are introduced in Section 3.1, their definition for and usage in a ComAnI instance are described in Section 3.2. Section 4.3 explains the development of this type of plug-ins in detail.

The **configuration file** in the lower part of Figure 1 defines a particular setup of the commit extraction and analysis and, hence, a specific instance of ComAnI. It consists of a set of configuration parameters for preparing the infrastructure (input and output locations, etc.) as well as defining the desired commit extractor and analyzer plug-ins. The infrastructure reads these parameters to configure ComAnI prior to its actual execution. Section 3.2 introduces the available configuration parameters and their definition for a particular ComAnI instance.

3 User Guide

Some introduction...

3.1 Installation

3.2 Execution

```
# The path to the directory, which contains the ComAnI plug-ins,
# like the available extractors and analyzers.
#
# Type: mandatory
# Default value: none
# Related parameters: none
core.plugins_dir = <Path>

# The identifyer of the version control system (VCS), which the
# repository as the input for commit extraction relies on.
# Commit extractors and analyzers need to support the VCS. See
# the respective documentations of the desired plug-ins.
#
# Type: mandatory
# Default value: none
# Related parameters: none
core.version_control_system = <VCS_Id>

# The number defining a particular log-level and, hence, the
# amount of information the infrastructure as well as the plug-ins
# provide at runtime.
# Valid values are:
#     0 - SILENT: No information is provided and, hence, there will
#                 be no message at all except for initial setup
#                 errors
#     1 - STANDARD: Basic information, warnings, and errors are
#                   provided
#     2 - DEBUG: Similar to STANDARD, but additional debug
#                information is provided
#
# Type: optional
# Default value: 1
# Related parameters: none
core.log_level = <0|1|2>
```

Listing 1: ComAnI core configuration parameters

```
# The fully qualified main class name of the commit extractor
# to use in the particular ComAnI instance. Although being
# mandatory, the infrastructure will ignore this parameter, if
# reuse is enabled.
#
# Type: mandatory
# Default value: none
# Related parameters: none
extraction.extractor = <Extractor>

# The path to the directory denoting the root of a software
# repository from which the commit extractor will extract the
# commits. Although being mandatory, extractors will ignore
# this parameter in interactive mode.
#
# Type: mandatory
# Default value: none
# Related parameters: none
extraction.input = <Path>

# The path to and name of the file containing a list of commit
# numbers. Extractors will try to extract the corresponding
# commits from the specified repository exclusively.
#
# Type: optional
# Default value: none
# Related parameters: none
extraction.commit_list = <Path>

# The path to the directory for saving extracted commits.
# Defining this parameter enables the caching feature for the
# extraction, which allows saving extracted commits as individual
# files and reuse them in future analyses, while the current
# analysis processes the extracted commits as usual. This avoids
# repeating the extraction of the same commits for future analyses.
#
# IMPORTANT: the infrastructure deletes the content of this
# directory, if it is not emtpy.
#
# Type: optional
# Default value: none
# Related parameters: extraction.reuse
extraction.cache = <Path>

# The path to the directory containing cached commits. Defining
# this parameter enables the caching feature for the extraction,
# which leads to a reuse of previously extracted commits instead of
# executing the defined extractor. This avoids repeating the
# extraction of the same commits for future analyses.
#
# IMPORTANT: if caching and reusing is defined at the same time,
# caching is performed and the analysis uses extracted commits.
#
# Type: optional
# Default value: none
# Related parameters: none
extraction.reuse = <Path>
```

Listing 2: ComAnI extraction configuration parameters

```
# The fully qualified main class name of the commit analyzer
# to use in the particular ComAnI instance.
#
# Type: mandatory
# Default value: none
# Related parameters: none
analysis.analyzer = <Analyzer>

# The path to the directory for saving the analyis results.
# Each analysis will create its own sub-directory in this
# directory named by the name of the analyzer and a timestamp
# to avoid unintended overriding of previous results.
#
# Type: mandatory
# Default value: none
# Related parameters: none
analysis.output = <Path>
```
Listing 3: ComAnI analysis configuration parameters

4 Developer Guide

Some introduction...

4.1 Data Model

Figure 2: ComAnI data model

4.2 Commit Extractor Plug-ins

A commit extractor plug-in is responsible for extracting commit information from a software repository and providing this information for an analysis. Therefore it has to create instances of the `Commit` and `ChangedArtifact` classes described in Section 4.1 and add these instances to the internal `CommitQueue`. This queue represents the actual connection between commit extractors and analyzers. It is accessible through an attribute of the `AbstractCommitExtractor` class, which each commit extractor has to extend. Figure 3 presents this abstract class as well as the extractor-specific commit queue interface.

ıceholder

Figure 3: ComAnI `AbstractCommitExtractor` class

Each commit extractor inherits three attributes: the infrastructure-wide logger, the extraction properties, and the commit queue as shown in Figure 3. The `logger` provides multiple methods for logging general information about the extraction process, warning, error, and debug messages. The amount of information actually shown, e.g., on the command line, depends on the defined log-level in the configuration file (cf. Section 3.2). The `extractionProperties` include all configuration parameters, which start with the prefix "`extraction.`", a property providing the name of the operating system on which the extractor currently runs, and a property for the version control system as specified by the respective configuration parameter in the configuration file (cf. Section 3.2). The `commitQueue` enables the transfer of extracted commits to an analysis. It only provides a single method, which accepts a single `Commit` instance as a parameter. Hence, the extraction algorithms have to call this method for each extracted commit individually.

Figure 3 also shows that a commit extractor has to implement a constructor, which accepts a properties and a particular extraction queue instance as parameters, as well as a set of methods for extracting commits and checking whether it is executable in the current environment. Listing 4 introduces a blueprint of a commit extractor, which implements all these required elements.

This blueprint represents a starting point for each new extractor by implementing the necessary algorithms as follows:

1. *Constructor*: creates a new instance of the commit extractor and, hence, has to call its parent class' constructor by passing the constructor parameters of the new extractor. Further actions for setting up the particular commit extractor can be performed here as well, which may also throw `ExtractionSetupExceptions`, if the setup fails. Listing 4 shows the constructor in Lines 16 to 22 including the usage of the `logger`, which informs the user about its creation (Line 20).

2. *Extraction methods*: realize the three extraction variants as introduced in Section 2. Each method in the Lines 25 to 41 returns a Boolean value indicating whether the particular extraction variant was successful (`true`) or not (`false`). In the latter case, the user is automatically informed about an extraction error indicating that either

there are no analysis results or the results may potentially be incorrect. The individual methods have the following purpose:

(a) `extract (File repository)`: extracts all commits of the given software repository. The `repository` parameter identifies the directory specified as input (`extraction.input`) in the configuration file (cf. Section 3.2), which is typically the root directory of a software repository. The particular way of interacting with the supported type of repository depends on the commands and capabilities provides by the version control system.

(b) `extract (File repository, List<String> commitList)`: extracts only those commits of the given software repository, which are part of the given commit list. While the `repository` parameter provides the same information as for the method above, the `commitList` parameter contains a set of commit numbers (or hashes), which enable to extraction of the respective commits. However, this method is only called if a commit list is defined via the corresponding configuration parameter in the configuration file.

(c) `extract (String commit)`: transforms the given information representing the content of a particular commit into a commit of the internal data model. This method is only called in the interactive mode of ComAnI, in which the given string is passed directly as a command line argument.

3. *Support check methods*: realize the opportunity to restrict the application of a commit extractor to a particular operating system and version control system. In particular, this is important, if, for example, an extractor relies on a third-party library, which is only available for Windows, or if the extractor cannot process other commits than those of a Git repository. A missing support yields a `ExtractionSetupException` during the creation of an instance of the extractor by the infrastructure, which terminates the entire tool. Each method in Lines 44 to 54 returns a Boolean value indicating whether the extractor supports the respective system (`true`) or not (`false`):

(a) `operatingSystemSupported(String os)`: checks whether the extractor supports the given operating system. The `os` parameter provides the operating system in the format of `System.getProperty("os.name")`[3].

(b) `versionControlSystemSupported(String vcs)`: checks whether the extractor supports the given version control system. The `vcs` parameter provides the version control system as defined by the `core.version_control_system` configuration parameter in the configuration file.

4.3 Commit Analyzer Plug-ins

[3] `https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html`

```java
package core;

import java.io.File;
import java.util.List;
import java.util.Properties;

import net.ssehub.comani.data.IExtractionQueue;
import net.ssehub.comani.extraction.AbstractCommitExtractor;
import net.ssehub.comani.extraction.ExtractionSetupException;

public class CommitExtractor extends AbstractCommitExtractor {

    public CommitExtractor(Properties arg0, IExtractionQueue arg1) throws ExtractionSetupException {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    @Override
    public boolean extract(File arg0) {
        System.out.println("I am a new commit extractor doing nothing!");
        return false;
    }

    @Override
    public boolean extract(String arg0) {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean extract(File arg0, List<String> arg1) {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean operatingSystemSupported(String arg0) {
        // TODO Auto-generated method stub
        return true;
    }

    @Override
    public boolean versionControlSystemSupported(String arg0) {
        // TODO Auto-generated method stub
        return true;
    }

}
```

Listing 4: Blueprint of a ComAnI commit extractor main class

ıceholder

Figure 4: ComAnI `AbstractCommitAnalyzer` class

```java
package core;

import java.util.Properties;

import net.ssehub.comani.analysis.AbstractCommitAnalyzer;
import net.ssehub.comani.analysis.AnalysisSetupException;
import net.ssehub.comani.data.IAnalysisQueue;

public class CommitAnalyzer extends AbstractCommitAnalyzer {

    public CommitAnalyzer(Properties arg0, IAnalysisQueue arg1)
    throws AnalysisSetupException {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    @Override
    public boolean analyze() {
        System.out.println("
    Yeeeeeeeeeeeeeeeeeeeaaaaaaaaaaaaaaaaahhhhh, commits, yum yum");
        return false;
    }

    @Override
    public boolean operatingSystemSupported(String arg0) {
        // TODO Auto-generated method stub
        return true;
    }

    @Override
    public boolean versionControlSystemSupported(String arg0) {
        // TODO Auto-generated method stub
        return true;
    }

}
```

Listing 5: Blueprint of a ComAnI commit analyzer main class

References

[1] Apache Software Foundation. Apache subversion. `https://subversion.apache.org/`, 2018. Accessed 2018/10/11.

[2] Git. Git version control system. `https://git-scm.com/`, 2018. Accessed 2018/10/11.

[3] C. Kröher, S. El-Sharkawy, and K. Schmid. KernelHaven - an experimentation workbench for analyzing software product lines. In *40th International Conference on Software Engineering: Companion Proceedings*, pages 73–76, New York, NY, USA, 2018. ACM.

[4] C. Kröher, S. El-Sharkawy, and K. Schmid. KernelHaven - an open infrastructure for product line analysis. In *22nd International Systems and Software Product Line Conference*, volume 2, pages 5–10, New York, NY, USA, 2018. ACM.

[5] C. Kröher, L. Gerling, and K. Schmid. Identifying the intensity of variability changes in software product line evolution. In *22nd International Systems and Software Product Line Conference*, volume 1, pages 54–64, New York, NY, USA, 2018. ACM.

[6] C. Kröher and K. Schmid. A commit-based analysis of software product line evolution: Two case studies. Technical Report SSE 2/17/E, University of Hildesheim, 2017.

[7] C. Kröher and K. Schmid. Towards a better understanding of software product line evolution. In *Softwaretechnik-Trends*, volume 37:2, pages 40–41, Berlin, Germany, 2017. Gesellschaft für Informatik e.V., Fachgruppe PARS.

[8] Linux Kernel Organization, Inc. The Linux kernel archives. `https://www.kernel.org/`, 2018. Accessed 2018/10/15.

[9] Stiftung Unviversity of Hildesheim - Software Systems Engineering. KernelHaven. `https://github.com/KernelHaven`, 2018. Accessed 2018/10/12.

[10] Stiftung Unviversity of Hildesheim - Software Systems Engineering. Variability-centric commit extraction and analysis. `https://github.com/SSE-LinuxAnalysis/ComAn`, 2018. Accessed 2018/10/15.

[11] L. Torvalds. Linux kernel source tree. `https://github.com/torvalds/linux`, 2018. Accessed 2018/10/15.