



D2.3

First Version of Developer Tools

WP2 – Guidelines and Procedure for System and Software Security and Legal Compliance

SIFIS-HOME

Secure Interoperable Full-Stack Internet of Things for Smart Home

Due date of deliverable: 30/09/2022

Actual submission date: 30/09/2022

Responsible partner: POL

Editor: Luca Arditò

E-mail address: luca.ardito@polito.it

29/09/2022

Version 1.0

Project co-funded by the European Commission within the Horizon 2020 Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	



The SIFIS-HOME Project is supported by funding under the Horizon 2020 Framework Program of the European Commission SU-ICT-02-2020 GA 952652

Authors: Luca Ardito (POL), Luca Barbato (LUM), Marco Ciurcina (POL), Maurizio Morisio (POL), Marco Torchiano (POL), Marco Rasori (CNR), Michele Valsesia (POL)

Reviewers: Andrea Saracino (CNR), Valerio Frascolla (INT)

Revision History

Version	Date	Name	Partner	Section Affected	Comments
0.1	12/05/2022	Tentative ToC and contents	POL, LUM, CNR	All	
0.2	09/06/2022	Software Analysis in SIFIS-Home	POL	Section 2	
0.3	16/06/2022	Added Ontology	CNR	Section 4	
0.4	30/06/2022	Adding working example	LUM	Section 5	
0.5	08/07/2022	Software Lifecycle Tools Description	POL	Section 3	
0.6	29/07/2022	Document Review	POL, LUM, CNR	All	
0.7	23/08/2022	Reviewed Subsections	CNR	Section 4	
0.8	09/09/2022	Added executive summary	POL	Executive Summary	
0.9	19/09/2022	Document proofread before internal review	POL	All	
1.0	29/09/2022	Modifications after internal review	POL	All	

Executive Summary

This document is a companion to the release of the 'developer tools' code, developed in the context of Work Package (WP) 2, which is available on the SIFIS-Home project GitHub repository <https://github.com/sifis-home>. All the source code is released under the MIT licence and has documentation directly from the repository.

This document provides the reader with detailed information about the released code and the API labelling and provides some results obtained from using it.

D2.3 is a preliminary release; therefore, there may be bugs, inaccuracies, or not yet complete functionality.

The final version is planned for M30 (end of March 2023) with the release of D2.5.

Table of contents

Executive Summary	3
1 Introduction.....	5
2 Software Lifecycle Tools.....	6
2.1 sifis-generate	6
2.1.1 Creating projects	8
2.2 Testing projects	10
2.2.1 Unit and Integration tests	10
2.2.2 Code Coverage.....	10
2.3 Evaluating Code Quality	12
2.3.1 Additional quality checks	15
2.3.2 Evaluating Software Quality	16
2.4 Memory fault analysis.....	16
2.4.1 C/C++.....	17
2.4.2 Rust	17
2.5 Weighted Code Coverage	18
2.6 Project Deployment.....	20
3 API Labelling Tools	22
3.1 The SIFIS-Home Hazards Ontology	22
3.2 Ontology Translation into Different Programming Languages	24
3.3 Labelling for Application Developers.....	26
3.4 Labelling for Users.....	27
4 A Complete Working Example	31
4.1 WoT Implementation in Rust.....	31
4.2 Tools in Action.....	32
4.2.1 datta.....	33
5 Conclusions and Future Works.....	35
6 References.....	36
Glossary	37

1 Introduction

In software development, static and dynamic analysis tools enable diagnostics at various levels during implementation, testing, integration, and later stages, like the development of patches and update management. Static and dynamic analysis tools also play an important role in increasing software security, as they can spot in advance attacks that exploit defects and malfunctions in a programme. By guaranteeing the quality of development, automatic analysis tools increase the efficiency of code and user satisfaction but also help reduce software vulnerability. By using one of such tools, it is possible to streamline processes, lower debugging costs, and achieve more secure, functional, usable, available, efficient, and maintainable software.

Our focus is on making sure that the software that is used in connected devices for homes is trustworthy, and that means, on the one hand, providing tools to help the developers avoid known pitfalls and problems. Having analysis tools allows third parties to assess the quality of their software.

In the following table, we list the status of the released code. In the following chapters, we provide further information about its use.

Table 1. Status of released code.

Project name	Status
<u>wot-td</u>	90% to Release 0.2
<u>wot-serve</u>	90% to Release 0.2
<u>wot-discovery</u>	50% to Release 0.2
<u>sifis-generate</u>	Release 0.5
<u>weighted-code-coverage</u>	Ready for Release 0.2
<u>complex-code-spotter</u>	Ready for Release 0.1
<u>datta</u>	Release 0.1
<u>libsifis-rs</u>	Proof of Concept
<u>The SIFIS-Home Hazards Ontology</u>	90% to Release 1.0.3

2 Software Lifecycle Tools

Making software could follow various workflows, depending on the organization writing it.

Yet when we interviewed our project partners, few declared that they use CI systems in their projects, usually pointing out that while they all recognize its value, it is often too time-consuming to set CI up. It is resource intensive to run CI for every commit.

In SIFIS-Home, we selected the best practices that should be followed and prepared a list of suggested open-source tools that could be used.

We developed new tools when nothing would fit our specific needs and contributed to already established projects.

Among the tools we made two try to make more time efficient building and running a CI system:

- 1 **sifis-generate**: A project creation tool that automatically scaffolds a project equipped with an initial CI flow matching our suggested practices.
- 2 **complex-code-spotter** and **weighted-code-coverage**: Analysis tools that help focus on the most complex parts of a codebase, shaping the test suite to cover first the code with the higher odds of hiding defects.

Both these tools have been written in Rust because of the advantages of this language, which, considering our needs, can be listed as:

- Memory safety without the need of a garbage collection.
- Possibility to optimise software both in time and memory.
- Writing parallel code in an easier way.
- Package and deploy software in a few steps.
- Support the most known platforms, such as Linux, macOS and Windows and bare-metal targets

To better describe our tools, we have divided their features into different sections, so to have a better understanding of the rationale we used to implement them.

2.1 *sifis-generate*

sifis-generate primary purpose consists in reducing the conceptual effort requested by developers to set up a CI pipeline.

It generates a whole new project from scratch if an automatized project generator is not available or too complex to use. Otherwise, it adds a series of CI configuration files to an existing project. Figure 1 shows third-party components of the library and the ones associated to the cli. The contributions made to the components in light violet have been performed in the scope of the SIFIS-Home project.

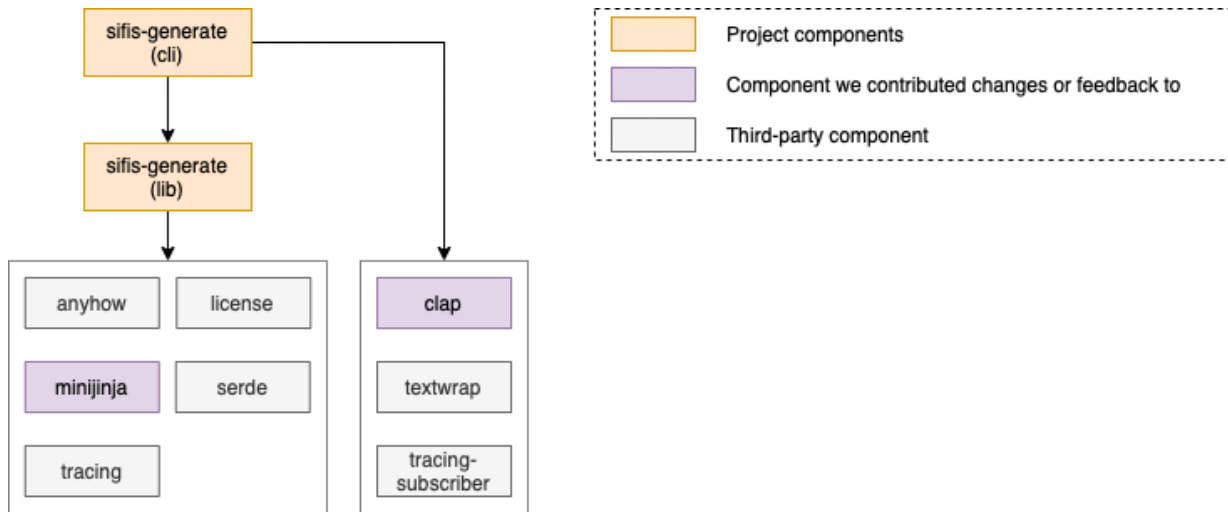


Figure 1: *sifis-generate cli and library with their third-party components*

This tool has been designed to cover some of the most common and used programming languages, with more emphasis on Rust for its advantages, as delineated above. Indeed, multiple changes applied to the tool source code are related to the CI developed for the Rust package manager: *cargo*.

sifis-generate produces a series of files from default templates associated and intended for that language by giving a programming language as input. Generated files can be of various kinds: source codes, build systems and CI configuration files, shell scripts, and Dockerfiles. All of them constitute and define a new project. Figure 2 illustrates a directory containing a new-generated project.

```

├── cli
│   ├── main.c
│   └── meson.build
├── docker-compose.yml
├── Dockerfile
├── lib
│   ├── foo.c
│   ├── foo.h
│   └── meson.build
├── meson.build
├── README.md
├── run_tests.sh
├── tests
│   ├── foo-lifecycle.c
│   └── meson.build
└── 3 directories, 12 files
    
```

Figure 2: *A simple C-demo project with its build configuration files*

Continuous Integration, better known by its initials CI, is composed of a series of steps which can perform lint, unit, and integration tests in addition to memory-hazards detection, all embodied with code and software quality checks. Each step runs one or more tools to obtain the desired result. Figure 3

shows the series of steps which might be run in a GitHub job for a specific configuration.

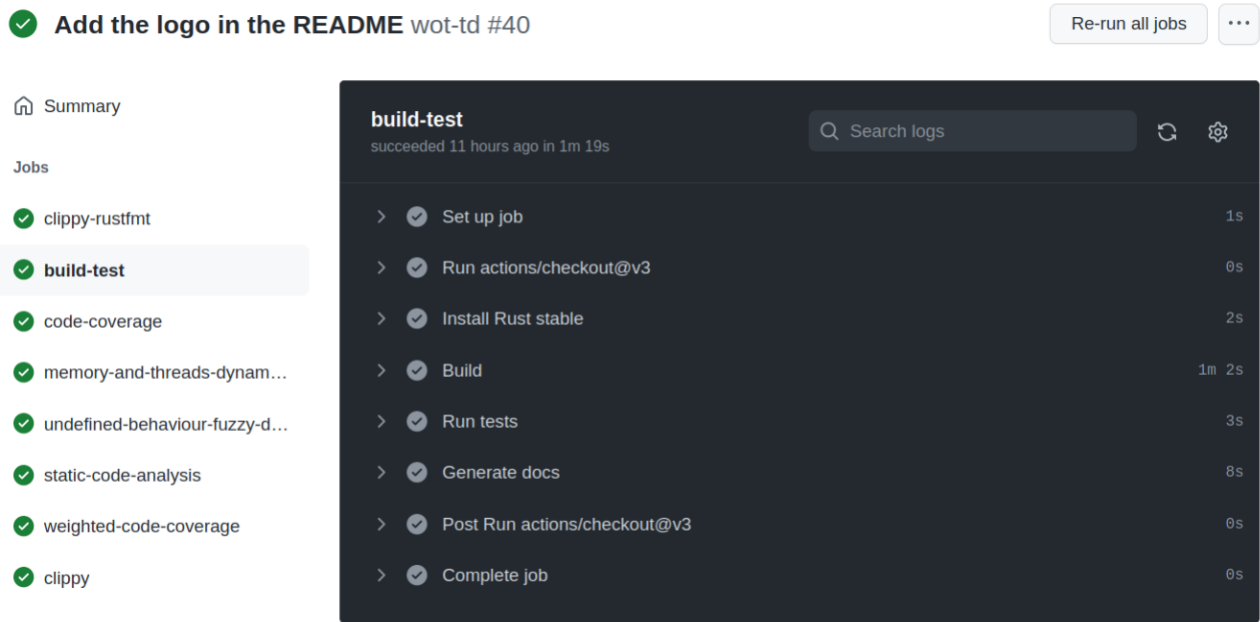


Figure 3: GitHub Actions CI service with its jobs and steps

2.1.1 Creating projects

Different programming languages need different configuration files to build and test software. Creating all these files could be cumbersome for a developer who must manually set up all of them. However, the newest languages provide a series of tools to create building environments automatically, but not all aspects are considered. For example, CI scripts are not produced at all. We have created the *sifis-generate* tool to overcome this problem: it generates new projects and CI scripts from scratch using templates.

Templates define the layout for a project and allow developers to insert data at runtime. Each template contains all necessary files to build a project for a determined programming language and CI and Docker scripts, which are used to run tests, deploy the software, and perform further checks.

sifis-generate must be considered as a starting point for a developer who is going to implement new software, so it defines default parameters and known patterns for each template.

As CI systems, GitHub Actions and GitLab CI/ Continuous Development (CD) have been taken into consideration because of their pervasiveness in the open-source world. Instead, Docker has been chosen as a solution to run software in an isolated environment.

Five templates have been defined, one for each programming language currently supported by the tool. Any template contains, by default, a README file with a series of badges to show some information about the quality of software in addition to the other details related to the new project.

C/C++

Meson is a new open-source build system meant to be extremely fast and, even more importantly, as user-friendly as possible. This tool has become popular in many C/C++ projects for its simplicity, so the decision to adopt it and generate its configuration files. In addition, a Dockerfile to build a Docker image and GitHub and GitLab CI scripts have been added.

Python

The innovative packager and dependency management *Poetry* has been used to build and deploy Python packages. GitHub and GitLab CI scripts have been provided even in this case.

Java

Apache Maven is a software project management and comprehension tool for Java. Configuration files for this software have been produced in addition to a GitHub CI script.

Rust

Cargo is the official package manager for Rust. It already implements a command to generate all building files: *cargo new*. This command also allows one to choose between a library or an executable as output. A series of GitHub CI scripts have been defined for this language because of Web of Things (WoT) Rust implementation demands. We have made available scripts for the most common architectures, Linux, macOS and Windows, in addition to a specific one for the deployment phase. A GitLab CI script has been also added. Figure 4 first shows the helper for *sifis-generate* and then the command to create a CI for Rust.

Javascript

Yarn is a JavaScript package manager. As Cargo, it already implements a command to generate a new project: *yarn init*. A GitHub and GitLab CI scripts are the only files defined for the JavaScript language.

```
[I] (master) ~/s/t/debug$ ./sifis-generate --help
sifis-generate

USAGE:
  sifis-generate [OPTIONS] <SUBCOMMAND>

OPTIONS:
  -h, --help      Print help information
  -v, --verbose   Output the generated paths as they are produced

SUBCOMMANDS:
  cargo      Generate a CI for a cargo project
  help      Print this message or the help of the given subcommand(s)
  maven     Generate a new maven project
  meson     Generate a new meson project
  poetry    Generate a new poetry project
  yarn      Generate a new yarn project
[I] (master) ~/s/t/debug$ ./sifis-generate -v cargo ./cargo-template
DEBUG sifis_generate: Creating ./cargo-template
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/deploy.yml
DEBUG sifis_generate: Creating ./cargo-template/README.md
DEBUG sifis_generate: Creating ./cargo-template/.gitlab-ci.yml
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/cargo-template.yml
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/cargo-template-ubuntu.yml
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/cargo-template-macos.yml
DEBUG sifis_generate: Creating ./cargo-template/.github/workflows/cargo-template-windows.yml
```

Figure 4: *sifis-generate* tool in action. At first, the usage of the tool is shown through the option `--help`; then, a Rust CI is created by using the “cargo” template.

2.2 Testing projects

Software testing checks the product's quality and finds bugs or features not covered and requested by the requirements, thus increasing the product's robustness.

The testing techniques are carried out to find bugs within the software and verify and certify that the software meets the requirements of the developers and customers, even under different climatic and electrical conditions.

Software testing can be performed either in parallel with the code development process or after the process is complete. Depending on when the testing is carried out, different methodologies are used.

In the context of the SIFIS-Home project, it is recommended that tests be performed during the software development phase through unit and integration tests, evaluating the percentage coverage that the written tests have over the software size (code coverage).

2.2.1 Unit and Integration tests

Software *must* be tested to verify and validate whether its behaviour is correct. In SIFIS-Home, software *should* implement at least unit tests and, optionally, integration tests.

A unit test verifies the functionality of a function or a subprocess, while an integration test analyses the interaction between the modules which compose a software. Thus, the latter focuses on a more general behaviour instead of a specific or local one.

Each programming language defines a different testing environment. Some tools are simpler to use than others and offer specific options to optimise the execution of the various tests. The most common SIFIS-Home frameworks and package managers for testing have been tackled, explaining their general structure and how developers can interact with them.

In Rust, a unit test is a simple function contained in the same source file of the function to verify, while an integration test is more complex and is defined as a file inside a reference directory called *tests*.

Both unit and integration tests are characterised by the attribute `#[test]` over their definition, which is used by *cargo* to create a series of small binaries for each test. Each binary runs in parallel since tests are independent of the other, and this approach reduces the overall testing time.

Even in *Meson*, it is possible to create unit and integration tests inside a reference directory called *tests*, but differently from *cargo*, though, a test needs to be defined as a binary directly by a developer inside a *meson.build* configuration file. So, the transformation of a test in binary is not performed automatically.

Instead, *Yarn* defines tests as scripts with a configuration file called *project.json*. It contains project metadata in addition to specific arguments and options for the JavaScript interpreter. It is necessary to launch the *yarn test* command to run tests.

Poetry makes use of *pytest*, a framework to write small and readable Python tests, to run unit and integration tests, launching the following command: *poetry run pytest*

For what concerns Java, the Maven package manager runs all tests defined in a library with the *mvn test* command. Group of tests are defined in the *src/test* package.

2.2.2 Code Coverage

Code coverage is a metric to determine how many lines of code have been covered by tests. The

percentage obtained from this metric represents a good indicator for software security since it permits tracking the most common execution flows a program might follow during its usage. The code coverage information is usually extracted by running instrumented builds and collecting their outputs in an aggregated report.

Some build systems have built-in subcommands to provide the code coverage nearly out of the box: Poetry with *poetry run coverage* or Meson with the *-Db_coverage=true* option to instrument the build and then *meson compile coverage-html* to aggregate the information.

The coverage information aggregators can directly upload the information to a third-party website to ease the processing and the analysis over time.

The most known sites for code coverage visualization are *codecov* and *coveralls*. Figure 5 shows the coverage for the *wot-td* repository on *coveralls*.

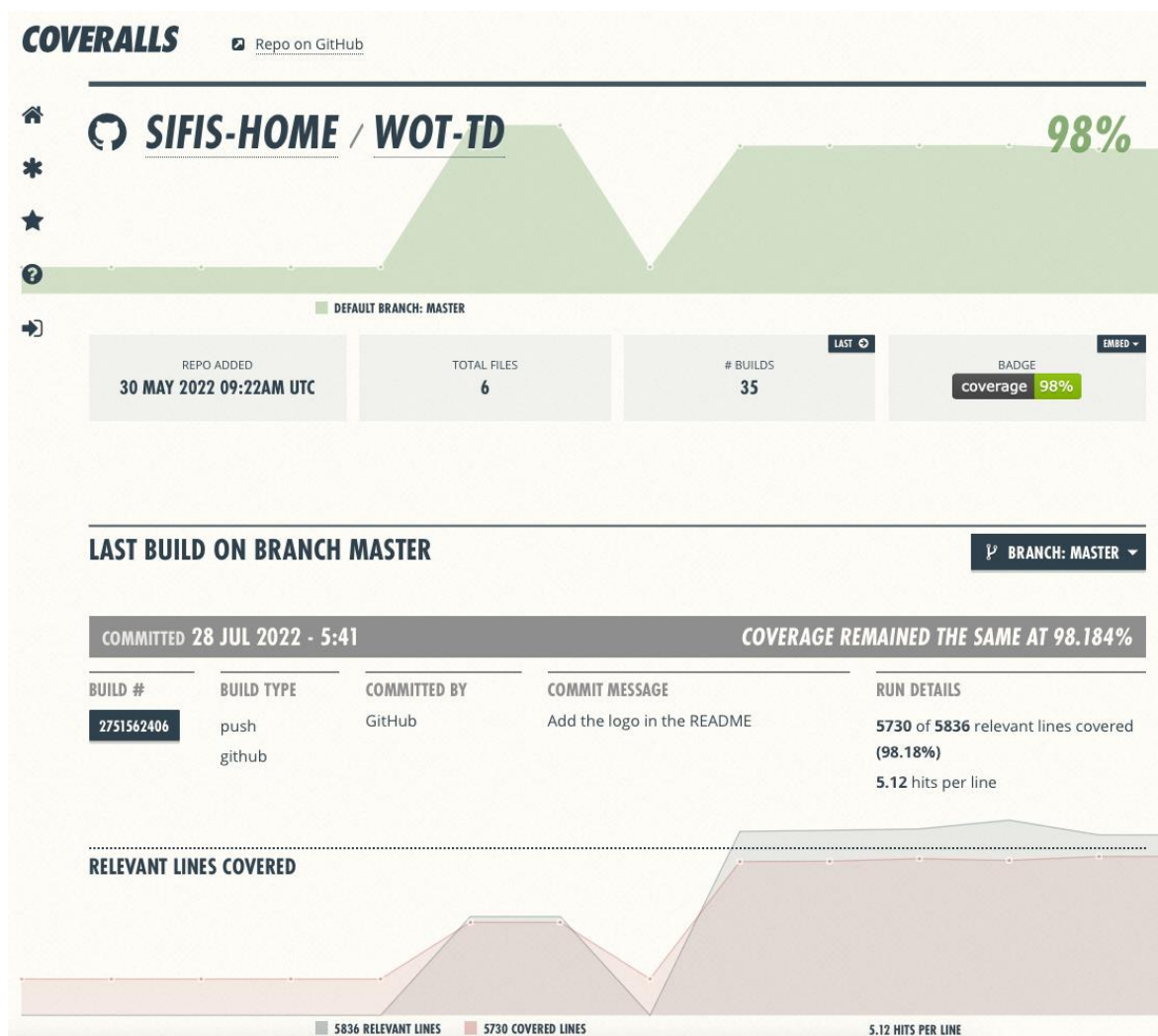


Figure 5: Code coverage visualization for *wot-td* repository on *Coveralls*

We implemented the traffic lights mechanism explained in D2.2 as part of our CI system to avoid reliance on those third-party services and provide a simpler quick check on the coverage health.

Code coverage with a percentage value greater than 80% is associated to **green** light, between 60% and 80% to **orange**, lower than 60% to **red**. If a code has been labelled as red, CI stops and exits with an error.

Among the various aggregation tools, we selected **grcov**, a tool developed by Mozilla in Rust, as the default tool for our *sifis-generate* CI. We also contributed a template system to customise the html reports (Figure 6).

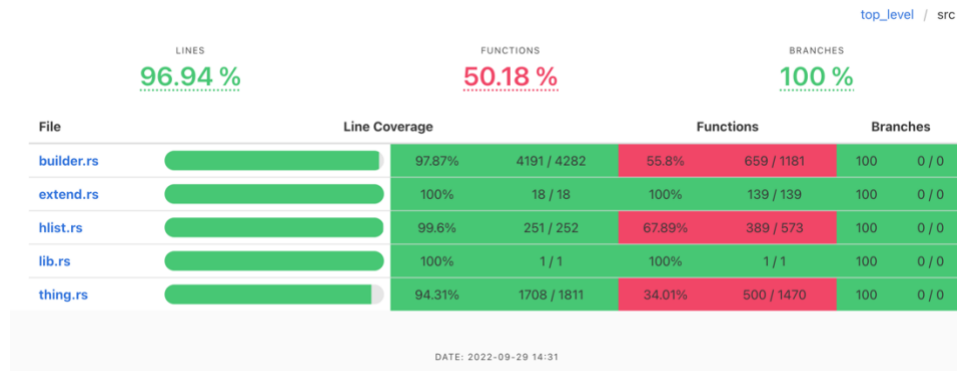


Figure 6 HTML report

The user can either rely on the third-party service or use the combination of the html output and the traffic-lights check to have the key features of coverage visualization and per build check.

2.3 Evaluating Code Quality

Code quality is an important aspect of the maintainability of software. Not only the structure of a piece of code should be considered, but also the libraries and the dependencies which contribute to reducing code boilerplate, avoiding the reinvention of the wheel every time.

In SIFIS-Home, we have contributed to the development of a software by Mozilla called [rust-code-analysis](#), in short *rca*, which implements a series of *Source Code Metrics* for the most used programming languages in the Firefox codebase. Those metrics focus on measuring the properties of a source code, mapping them into numerical values. In addition, they allow identifying parts of the software that need a refactor or structural changes.

This tool computes the metrics in parallel, distributing each file to the threads available on the system. For each file, an AST has built through a third-part dependency called *tree-sitter*, which also provides some functions to interact with its nodes. So, visiting the nodes of this tree, it is possible to identify the token and the constructs of the analysed programming language and then compute the metrics starting from there.

rust-code-analysis also contains the code complexity metrics necessary for:

- Extracting snippets of complex code from a project.
- Computing the four algorithms associated with the new code coverage concept described in D2.2.

Detecting pieces of complex code is the main goal of *complex-code-spotter*, a new tool developed in SIFIS-Home. Figure 7 shows third-party components of the library and the ones associated to the cli.

The contributions made to the components in light violet have been performed in the scope of the SIFIS-Home project.

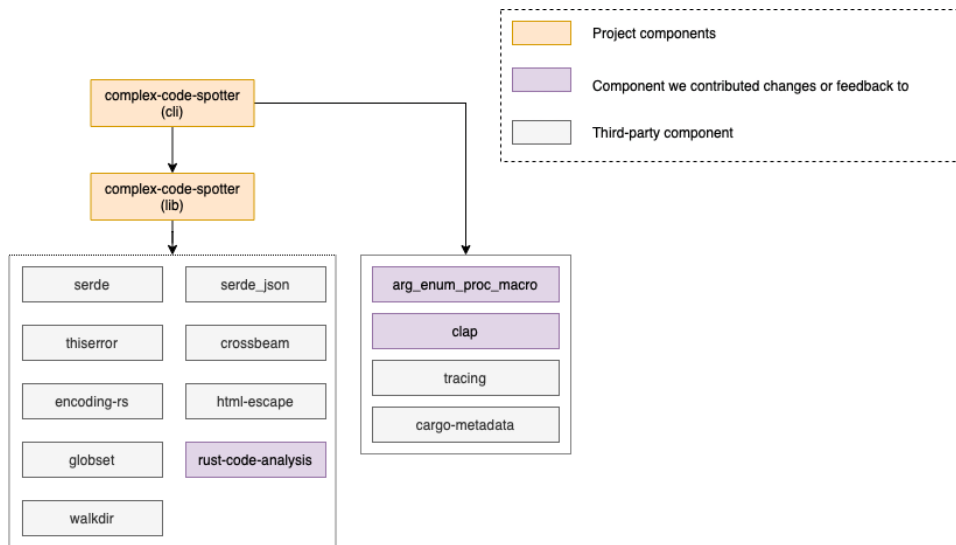


Figure 7: `complex-code-spotter` cli and library with their third-party components

It requires as input:

- The path to the source code, which is parsed in search of complex snippets.
- The output directory containing all code snippets which exceed the thresholds of the implemented code complexity metrics.

A snippet is any piece of code identified by a `rust-code-analysis` space. This approach leads to a more defined granularity because retrieving functions and closures other than whole files is possible. It supports the same programming languages defined in **rca**.

Concerning its usage, this tool uses **rca** as a dependency for computing code complexity metrics, extracts a code snippet when the value of a metric exceeds a determined threshold, and then saves those snippets in one of the supported formats, which can be: *markdown*, *html*, and *JSON*.

The default configuration extracts code snippets for *cyclomatic* and *cognitive* metrics, with an empirical threshold of *15*, and defines *markdown* as the default output format.

Other optional arguments are:

- The kinds of code complexity metrics with their own thresholds.
- The number of computational threads.
- An option to visualize the operations performed by the tool.

It is also possible to filter input source files using *I* and *X* options. The *input -I* option is a glob filter that considers **only** the files with a determined file extension. The *exclude -X* option is a glob filter that **does not consider only** the files with a determined file extension.

For example, to consider only *Rust *.rs* files:

```
complex-code-spotter -I "*.rs" /path/to/file/or/directory /output/path
```

To exclude only *Rust* **.rs* files instead:

```
complex-code-spotter -X "*.rs" /path/to/file/or/directory /output/path
```

Both these options can be used more than once.

The main workflow of *complex-code-spotter* is the following: each source code file is assigned to one of the threads available on the system, which computes the complexity metrics for that file and then extracts the relative snippets.

The results produced by each thread are then sent to a collector thread which, as the first step, creates the structure of the output directory and then writes every snippet of a file from memory into the output format file. In Figure 8, the list of arguments and options for *complex-code-spotter*.

```
complex-code-spotter
USAGE:
  complex-code-spotter [OPTIONS] <SOURCE_PATH> <OUTPUT_PATH>
ARGS:
  <SOURCE_PATH>
    Path to the source files to be analyzed

  <OUTPUT_PATH>
    Output path containing the snippets of complex code for each file
OPTIONS:
  -c, --complexities <COMPLEXITIES>
    Threshold 0 is minimum value, thus no threshold at all.
    Threshold 100 is maximum value, thus each complexity value is not accepted.

    Thresholds 0 and 100 are extremes and are generally not recommended

    [default: cyclomatic:15 cognitive:15]

  -h, --help
    Print help information

  -I, --include <INCLUDE>
    Glob to include files

  -O, --output-format <OUTPUT_FORMAT>
    Output format

    [default: markdown]
    [possible values: markdown, html, json, all]

  -v, --verbose
    Output the generated paths as they are produced

  -X, --exclude <EXCLUDE>
    Glob to exclude files
```

Figure 8: *complex-code-spotter* interface

We have also added a *cargo* subcommand to extract snippets using *cargo* itself. Figure 9 shows the list of arguments and options for *cargo* subcommand.

```

complex-code-spotter
Complex Code Spotter cargo applet

USAGE:
  cargo-ccs <SUBCOMMAND>

OPTIONS:
  -h, --help    Print help information

SUBCOMMANDS:
  ccs    Complex Code Spotter cargo subcommand
  help   Print this message or the help of the given subcommand(s)

```

Figure 9: cargo subcommands

Below a complex code snippet with a cyclomatic complexity of 26 written in Rust code.

```

loop {
  if let Some(subtype) = subtype.as_ref() {
    match subtype {
      Array(array) => {
        match (array.min_items, array.max_items) {
          (Some(min), Some(max)) if min > max => return
Err(Error::InvalidMinMax),
          _ => {}
        };

        if let Some(items) = array.items.as_deref() {
          stack.extend(items.iter());
        }
      }
      Number(number) => {
        match (number.minimum, number.maximum) {
          (Some(x), _) | (_, Some(x)) if x.is_nan() => return
Err(Error::NanMinMax),
          (Some(min), Some(max)) if min > max => return
Err(Error::InvalidMinMax),
          _ => {}
        }

        match number.multiple_of {
          Some(multiple_of) if multiple_of <= 0. => {
            return Err(Error::InvalidMultipleOf)
          }
          _ => {}
        }
      }
    } // match
  } // if let
} // loop

```

2.3.1 Additional quality checks

Specifying dependencies during software development and not using them afterwards is a typical pattern

that could lead to confusion for new contributors to the code. To solve this problem for Rust, we have defined a CI step which detects unused crates running a command with the nightly toolchain of the language:

```
cargo +nightly udeps -all-targets
```

The *all-targets* option runs this check for tests and benchmarks crates too.

A developer might require specific and acceptable license terms for the dependencies of the software. To check whether these requirements have been satisfied, we have added a Rust tool called *cargo deny* to our CI:

```
cargo deny check licenses
```

it exits with an error when a dependency license is not contained in the ones specified for the codebase. This tool can also deny (or allow) specific crates, as well as detect and handle multiple versions of the same crate with:

```
cargo deny check bans
```

Running the *cargo audit* command can identify some dependencies containing security vulnerabilities. It searches for crate vulnerabilities inside a database; if one or more crates are affected, it returns an error. This software also offers the possibility to update or replace dangerous dependency requirements using this command:

```
cargo audit fix
```

2.3.2 Evaluating Software Quality

Software quality evaluation consists of a series of checks performed at runtime. One example is the code coverage metric explained in previous sections, which needs to know execution flows to determine the percentage of covered lines. So, if a test does not undertake a specific flow, some lines *might* not be covered.

Code coverage can be applied to any programming language, but there are some languages, mainly compiled ones, which could incur memory management and threading bugs, so their behaviour should be profiled in detail at runtime.

2.4 Memory fault analysis

Dynamic analysis (or dynamic code analysis) methods analyse the software running. The dynamic analysis techniques aim to find errors in a program while executing (instead of examining the code itself) and can identify:

- Lack of code coverage.
- Memory allocation and leaks errors.
- Fault localization according to failing and passing test cases.

- Concurrency errors (race conditions, exceptions, resource & memory leaks, and security attack vulnerabilities).
- Performance bottlenecks and security vulnerabilities.

2.4.1 C/C++

For C/C++ we run a fast memory error detector, a runtime library part of the *clang* and *GCC* suites called **AddressSanitizer**, better known as *asan*. It detects out-of-bounds accesses to heap, stack and globals, use-after-free, double-free, invalid-free, and more generics memory leaks. This kind of verification is preminent for security reasons since an attacker could exploit a memory hazard to insert a malevolent piece of code inside the software. The meson command to produce an asan-instrumented build is:

```
meson setup --buildtype release -Db_sanitize=address -Db_lundef=false .build-  
directory-asan
```

which builds up the binary, runs it in search of memory problems and then saves the results, if there are any, in a directory called *.build-directory-asan*.

After that, the AddressSanitizer runs tests and inspects them through this command:

```
meson test -C .build-directory-asan
```

However, some platforms are not yet supported by AddressSanitizer, so different tools need to be run: **Valgrind** and **Memcheck**.

Valgrind is a suite of analysis tools that rely on a CPU emulator to run instrumented binaries and insert profiling/instrumentation information on the fly. It provides a plethora of tools not limited to fault analysis (memory access, thread usage) but also profiling ones (memory allocation, cache usage, per function execution time, etc.).

```
meson test --wrap='valgrind --leak-check=full --error-exitcode=1' <test>
```

Instead, *Memcheck* has a higher overhead compared to asan and should be used mainly when asan does not support the architecture, or an in-depth analysis is required.

2.4.2 Rust

The Rust language compiler can instrument the code with AddressSanitiser passing the compilation flag through the *RUSTFLAGS* environment variable:

```
export RUSTFLAGS=-Zsanitizer=address  
cargo test
```

For the platforms that do not support AddressSanitizer yet, we rely on the *cargo valgrind* subcommand, which invokes *valgrind memcheck* under the hood.

```
cargo valgrind run -- --command_and_options_to_be_analysed
```

Miri is an experimental interpreter of the Rust Mid-level Intermediate Language (MIR). It is used to detect Undefined Behaviours (UB).

Currently, it covers the following aspects:

- Out-of-bounds memory accesses and use-after-free.
- Invalid use of uninitialized data.
- Violation of intrinsic preconditions.
- Not sufficiently aligned memory accesses and references.
- Violation of *some* basic type invariants (a boolean which is not 0 or 1, for example, or an invalid enum discriminant).

The possibility of detecting memory leaks is one of the upcoming features currently under development in *Miri*. Once it reaches an adequate level of maturity, it might replace *Memcheck*.

In our CI, we have set up some of these options, creating a default configuration which covers:

- A lot of extra UB checks relating to raw pointer aliasing rules.
- A stricter alignment checks.
- Validity rules for integer and float values, like forced initialisations.

By default, *Miri* runs a binary in an isolation environment not to be affected by any architectures-dependent instructions. Still, it is always possible to disable this behaviour, allowing software to access resources such as environment variables, file systems, and randomness.

2.5 *Weighted Code Coverage*

Running lots of tests to achieve an adequate code coverage can be time and resource-intensive, even more so if the tests are built multiple times for instrumenting them according to different purposes (e.g., code coverage and fault analysis).

In D 2.2, we have introduced two algorithms that combine code complexity and code coverage into a new metric called *weighted code coverage*, which aims to suggest to the developer which areas require extra care, and which are less likely to hide bugs.

Those algorithms have been implemented in the *weighted-code-coverage* software that also contains two additional algorithms from the Ruby language, called *Skunk* and *Crap*, based on a weighted code coverage concept. Figure 10 shows third-party components of the library and the ones associated to the cli. The contributions made to the components in light violet have been performed in

the scope of the SIFIS-Home project.

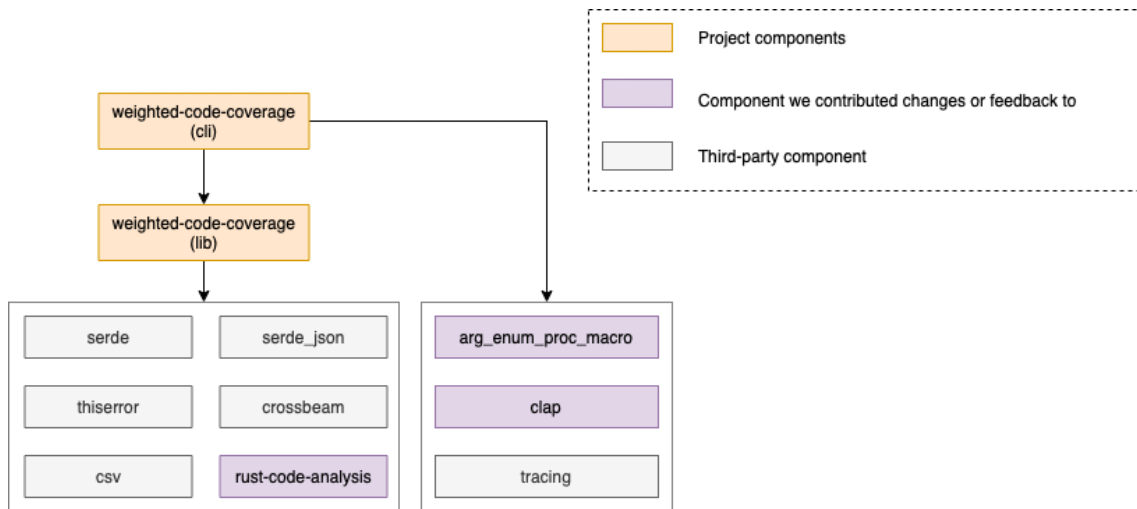


Figure 10: `weighted-code-coverage cli` and `library` with their third-party components

The tool requires only two mandatory arguments as input:

- The path to the source code's directory to compute code complexity metrics.
- The JSON file produced by `grcov` that contains covered and uncovered lines of information for each file.

It is possible to export the results of all four algorithms in two different formats: JSON and CSV. If an output path has not been specified, results are printed on the terminal.

Other optional arguments are:

- The kinds of code complexity metrics with their thresholds.
- The number of computational threads.
- An option to visualize the operations performed by the tool.

The execution flow is the following: source code files are grouped in chunks, and then each chunk is assigned to a thread that computes all four algorithms on each chunk file. The results produced by each thread are then sent to a collector thread which merges all of them into the output file.

In addition to computation for single files, the tool can be more granular and consider functions too. In this way, it is possible to retrieve which functions are uncovered, and which ones contain a code difficult to comprehend at first glance.

Currently, `weighted-code-coverage` analyses only Rust files, but it might be expanded to other programming languages in the future. Figure 11 shows the list of arguments and options for `weighted-code-coverage`.

```
weighted-code-coverage 0.2.0
USAGE:
  weighted-code-coverage [OPTIONS] --path_file <PATH_FILE> --path_json <PATH_JSON>
OPTIONS:
  -c, --complexity <COMPLEXITY>
      Choose complexity metric to use

      [default: cyclomatic]
      [possible values: cyclomatic, cognitive]

  --csv <PATH_CSV>
      Path where to save the output of the csv file

  -f, --json-format <JSON_FORMAT>
      Specify the type of format used between coveralls and covdir

      [default: coveralls]
      [possible values: covdir, coveralls]

  -h, --help
      Print help information
```

Figure 11: A portion of weighted-code-coverage command line interface

2.6 Project Deployment

Software lifecycle tools are deployed as binaries on GitHub to distribute effortlessly among the supported architectures, in our case, Linux, macOS and Windows.

The deployment procedure is performed by a CI script added to *sifis-generate*. This script runs in parallel a task for each architecture with the purpose of building and packaging binaries. Subsequently, another task starts up and collects all produced packages, uploading them on GitHub. The task sequence is triggered whenever a new tag is added. Figure 12 shows the deployment procedure in action, with the relative produced artifacts.

Prepare for release ⋮
deploy #15

Summary

Jobs

- create-windows-binaries
- create-unix-binaries (u...
- create-unix-binaries (...)
- deploy

Triggered via push 3 months ago

lu-zero pushed → 490f6b1 v0.3.1	Status	Total duration	Artifacts
	Success	8m 21s	3

deploy.yml
on: push

Matrix: create-unix-binaries

2 jobs completed
[Show all jobs](#)

deploy 52s

create-windows-binaries 6m 53s

deploy 52s

🗂 - +

Artifacts
Produced during runtime

Name	Size	
sifis-generate-0.3.1-x86_64-apple-darwin.tar.gz	2 MB	🗑
sifis-generate-0.3.1-x86_64-pc-windows-msvc.zip	1.65 MB	🗑
sifis-generate-0.3.1-x86_64-unknown-linux-musl.tar.gz	2.13 MB	🗑

Figure 12: Deployment of sifis-generate

3 API Labelling Tools

The tools we show in this section use the concepts of API Label and App Label described in Section 3 of D2.2 (Preliminary Developer Guidelines). The term "tag" used in D2.2 is replaced herewith by the term "hazard".

We recall that an API Label is associated with a SIFIS-Home developer API to describe possible risks deriving from its execution. The API Label consists of a list of hazards, each identifying a risk. On the other hand, the App Label is a label associated with an application written by a third-party developer; it is derived from the combination of the API Labels related to the SIFIS-Home developer APIs invoked by the application's source code.

The list of hazards presented in D2.2 has been formally defined in an ontology introduced in the following section.

3.1 *The SIFIS-Home Hazards Ontology*

An ontology called *SIFIS-Home Hazards Ontology* (SHO) has been created to formally define the hazards we identified for the smart home environment. Notably, the SHO can be used to extend the representation of WoT's smart devices, called Thing Descriptions (TDs).

A TD is a JSON-LD representation of a connected device called Thing. The TD ideally provides all the information to control the device in a structured way.

Every possible interaction is mapped through 3 categories: Properties, Actions, and Events. A client consuming the description can set or read a Property, subscribe/unsubscribe for future Events or issue a complex order and then wait for it to happen (Action).

Since the TD is a JSON-LD, it is possible to extend it and add semantic meaning to every element of it; the SHO is used to bind the risk information to every interaction described.

For example, a Property of an oven's TD could be "on". Of course, turning an oven on may pose some risks, especially if this function is called remotely and the home is unattended. Within a TD, such risks can be expressed precisely by mapping the related hazards to the "On" state, i.e., when the "on" property value is "true".

```
"property" : {
  "on" : {
    "@type" : "OnOffProperty",
    "type" : "boolean",
    "hazards" : [
      {
        "@id" : "sho:ElectricEnergyConsumption",
        "title" : "Electric energy consumption",
        "description" : "The execution enables a device that consumes electricity",
        "riskScore" : 5,
        "type" : "boolean",
        "const" : true
      },
      {
```

```

    "@id": "sho:FireHazard",
    "title": "Fire hazard",
    "description" : "The execution might cause fire",
    "riskScore" : 8,
    "type" : "boolean",
    "const" : true
  }
]
}
}

```

Figure 13 shows a graphical representation of the SHO by means of a force-directed graph layout.

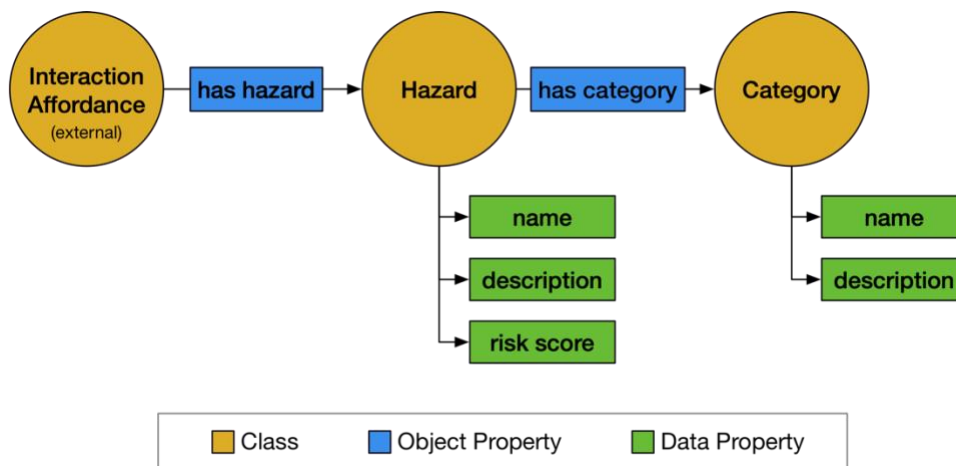


Figure 13: Visualization of the SIFIS-Home Hazards Ontology

The SHO defines the classes Hazard and Category, and it uses the external class Interaction Affordance (defined in the Thing Description Ontology [TD, 2022]) to link to the WoT world. In particular, the object property hasHazard connects the Interaction Affordance class to the Hazard class with a one-to-many relationship, meaning that a Property, an Action, or an Event can be associated with one or more objects of class Hazard.

The Hazard class is characterized by the data properties "name" and "description" of type string, and, optionally, "risk score" of type level, defined as an integer in the range from 0 to 10; Also, the Hazard class is connected to the Category class through an object property named hasCategory, indicating that a hazard belongs to some category.

The Category class is characterized by the data properties "name" and "description" of type string. Moreover, the ontology defines three Named Individuals for the Category class:

- Safety: category identifying hazards that may lead to physical harm to people and/or assets.
- Privacy: category identifying hazards that may compromise privacy.
- Financial: category identifying hazards that lead to an expense.

Currently, the ontology defines also 22 Named Individuals for the Hazard class. As an example, we report in Figure 14 the complete description of a hazard named ElectricEnergyConsumption.

The screenshot shows a web page for the Named Individual **ElectricEnergyConsumptionⁿⁱ**. At the top right, there is a link: [back to ToC or Named Individual ToC](#). Below the title, the IRI is listed as `https://purl.org/sifis/hazards#ElectricEnergyConsumption`. A description follows: "A type of finance-related hazard". A dashed line separates this from the "belongs to" section, which lists [Hazard^c](#). Below that is the "has facts" section, containing:

- [has category^{op}](#) [Financial](#)
- [description^{dp}](#) *"The execution enables a device that consumes electricity"*
- [name^{dp}](#) *"Electric energy consumption"*
- [risk score^{dp}](#) *^^^level*

Figure 14: Example of hazard defined as Named Individual within the SHO.

The screenshot above has been taken from the SIFIS-Home Hazards Ontology specification webpage (SHO, 2022), where the whole ontology is described and can be downloaded in various serialized formats, among which JSON-LD.

3.2 Ontology Translation into Different Programming Languages

A serialized version of the SIFIS-Home Hazards Ontology has to be translated into specific programming languages to be used for programming purposes. It is needed in many cases, such as for creating the API Labels and the App Label and deserializing Thing Descriptions.

To this aim, we created a tool called **generate-sifis-hazards** which reads the SHO's JSON-LD serialized file and translates it into a programming-language-specific file containing the structures to represent the ontology. Currently, the only programming language supported is Rust.

This tool fills a given template, different for each programming language, with information extracted from the ontology, generating in the output the programming-language-specific file, which allows to interact with hazards and get their data. A template is composed of a series of APIs and structures representing concepts and information contained in an ontology, so the same goes for the output file produced by the tool.

The `generate-sifis-hazards` is run from the command-line interface, as shown in Figure 15, and its inputs are:

- `<ONTOLOGY_PATH>`: the path to the input JSON-LD file containing the ontology.
- `<TEMPLATE>`: the template to be used, which determines the programming language the ontology will be translated into.
- `<OUTPUT_PATH>`: the path where the output programming-language-specific file will be stored.


```

generate-sifis-hazards

USAGE:
  generate-sifis-hazards [OPTIONS] --template <TEMPLATE> <ONTOLOGY_PATH> <OUTPUT_PATH>

ARGS:
  <ONTOLOGY_PATH>  Path to the ontology file
  <OUTPUT_PATH>    Path to the generated API

OPTIONS:
  -h, --help          Print help information
  -t, --template <TEMPLATE> Name of a builtin template [possible values: rust]
  -v, --verbose       Output the generated paths as they are produced

```

Figure 15: generate-sifis-hazards command line interface

Below we show a portion of the output produced by generate-sifis-hazards representing a Rust enumerator, including all the hazards extracted from the ontology.

```

/// Hazards type.
pub enum Hazard {
  /// The execution may release toxic gases
  AirPoisoning,
  /// The execution may cause oxygen deficiency by gaseous substances
  Asphyxia,
  /// The execution authorises the app to record and save a video with audio on
  persistent storage
  AudioVideoRecordAndStore,
  /// The execution authorises the app to obtain a video stream with audio
  AudioVideoStream,
  /// The execution enables a device that consumes electricity
  ElectricEnergyConsumption,
  /// The execution may cause an explosion
  Explosion,
  /// The execution may cause fire
  FireHazard,
  /// The execution enables a device that consumes gas
  GasConsumption,
  /// The execution authorises the app to get and save information about the
  app's energy impact on the device the app runs on
  LogEnergyConsumption,
  /// The execution authorises the app to get and save information about the
  app's duration of use
  LogUsageTime,
  /// The execution authorises the app to use payment information and make a
  periodic payment
  PaySubscriptionFee,
  /// The execution may cause an interruption in the supply of electricity
  PowerOutage,
  /// The execution may lead to exposure to high voltages
  PowerSurge,
  /// The execution authorises the app to get and save user inputs
  RecordIssuedCommands,
  /// The execution authorises the app to get and save information about the
  user's preferences
  RecordUserPreferences,

```

```

    /// The execution authorises the app to use payment information and make a
payment transaction
    SpendMoney,
    /// The execution may lead to rotten food
    SpoiledFood,
    /// The execution authorises the app to read the display output and take
screenshots of it
    TakeDeviceScreenshots,
    /// The execution authorises the app to use a camera and take photos
    TakePictures,
    /// The execution disables a protection mechanism and unauthorised
individuals may physically enter home
    UnauthorisedPhysicalAccess,
    /// The execution enables a device that consumes water
    WaterConsumption,
    /// The execution allows water usage which may lead to flood
    WaterFlooding,
}

```

3.3 Labelling for Application Developers

Third-party application developers are provided with a high-level API (SIFIS-Home Developer API) that abstracts away the Web of Thing details and forces some constraints on what can be represented:

- The device must provide only the set of interaction affordances linked to their category (e.g., a Lamp must provide an OnOff property)
- The devices are exposed using concrete classes with self-explanatory methods (e.g., Lamp has a *turn_light_on* method)

The SIFIS-Home Developer APIs provide a means to search for devices filtering it according to the risk level and the kind of hazards. A SIFIS-Home Developer API expresses in its documentation the inherent risks, i.e., the API Label.

```

/// Turns a light on.
///
/// # Hazards
///
/// * Fire hazard\
///   The execution may cause fire
pub fn turn_light_on(&mut self, brightness: Percentage, color: Rgb) -> Result<()>
{ ... }

```

Since API Labels are code comments, they can be represented as popup notes that appear as part of an autocomplete feature in a development tool. This task is nowadays performed by an implementation of a Language Server Protocol.

Autocomplete, go-to definition or documentation on hover features are usually implemented similarly for each development tool providing different APIs for the same goal. A Language Server Protocol has been invented to reduce this attitude by defining a standardized protocol to interact with these development tools through inter-process communication. The protocol delineates a series of guidelines to build a server that can be reused in multiple development tools, providing the features described

above and supporting various programming languages with minimal effort. Many development tools implement this protocol, particularly code editors like VS Code, Emacs and Vim.

In SIFIS-Home, an overview of a typical implementation session conducted by a third-party developer has been performed using *rust-analyzer*, a Language Server Protocol implementation for the Rust language, and VS Code as a code editor. Figure 16 shows the autocompletion plug-in in action within VS Code.

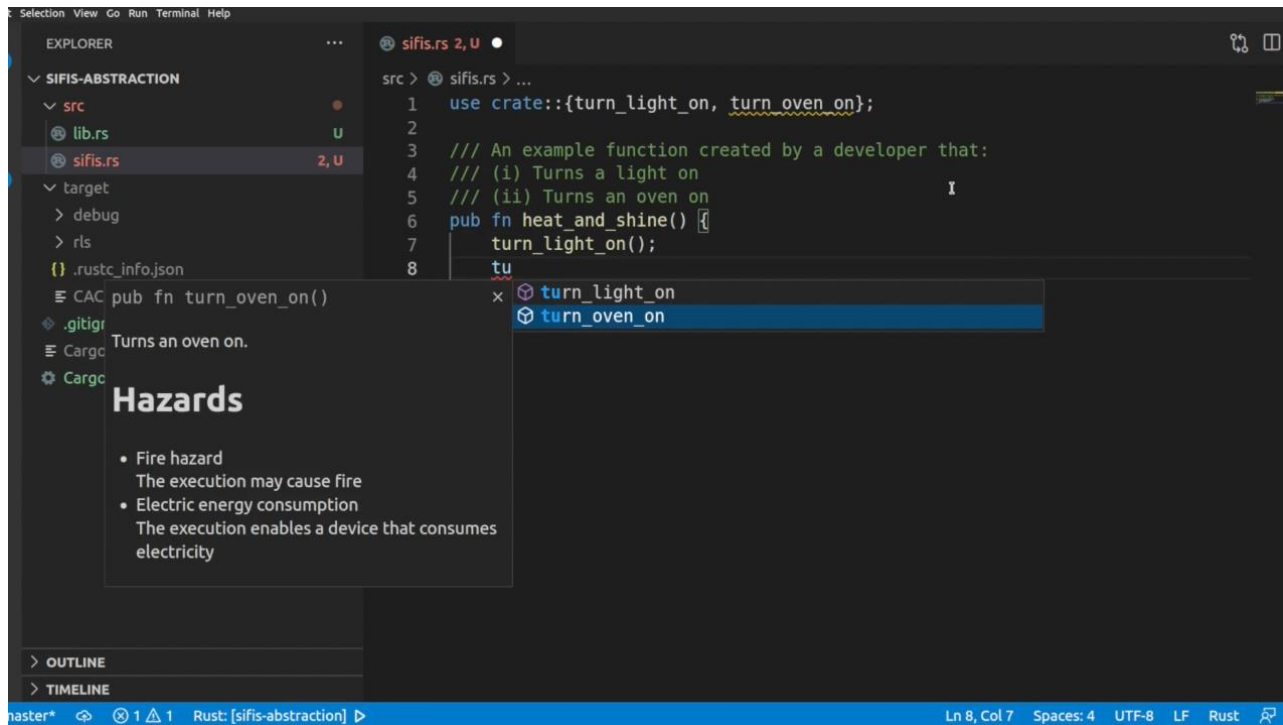


Figure 16: The VS Code plug-in in action. It allows autocompletion of SIFIS-Home Developer APIs and shows the description of a SIFIS-Home Developer API and its related hazards.

3.4 Labelling for Users

The usefulness of security labels regarding users becomes clear when we reason about applications users wish to install onto their smart devices. Indeed, every third-party application includes an App Label, which is used to notify users of all possible hazards that may arise during execution. Before an application is installed, a series of application permissions are presented to the user, who can choose which hazards are permitted and which ones are prohibited.

When a permission is denied, the SIFIS-Home Developer APIs associated with that hazard are disabled and calling those methods would either request a user interaction, silently fail, or terminate the application depending on the execution context.

To extract every hazard contained in an application, and thus to create the App Label, a tool called *manifest* has been developed in Rust. Figure 17 shows its usage and options printed at the command-line interface. The *manifest* tool analyses the binary format of an application to find all SIFIS-Home Developer APIs contained in it and subsequently discover which are the hazards paired with each retrieved API. The tool needs only two input parameters: the application binary path and the Library API labels path.

```
manifest
USAGE:
  manifest [OPTIONS] -b <BINARY_PATH> -l <LIBRARY_API_LABELS_PATH>
OPTIONS:
  -b <BINARY_PATH>          Path to the binary to be analyzed
  -h, --help                Print help information
  -l <LIBRARY_API_LABELS_PATH> Path to the SIFIS-Home library API labels
  -o <OUTPUT_PATH>         Output path of the produced manifest
  -v, --verbose             Enable additional information about the underlying process
```

Figure 17: manifest command line interface

In particular, the Library API labels path is a JSON file formed by an array of API Labels, including all the SIFIS-Home Developer APIs contained in a specific version of a SIFIS-Home library. Each API Label within this file is defined according to the API Label Schema introduced in Section 3.5.1.1 of D2.2. Note that since new APIs can be added or removed over time, this file is likely to be different for each version of the SIFIS-Home library.

Below, the content of a Library API labels file is shown. Within the file, an API Label is included for each SIFIS-Home Developer API. In this oversimplified example, the number of all the SIFIS-Home Developer APIs defined within the SIFIS-Home library is two.

```
{
  "version": "0.1",
  "api_labels": [
    {
      "api_name": "turn_light_on",
      "description": "Turns on a lamp.",
      "security_label": {
        "safety": [
          {
            "name": "FIRE_HAZARD",
            "description": "The execution may cause fire."
          }
        ],
        "privacy": [
          {
            "name": "LOG_ENERGY_CONSUMPTION",
            "description": "The execution allows the app to register information
about energy consumption."
          }
        ],
        "financial": [
          {
            "name": "ELECTRIC_ENERGY_CONSUMPTION",
            "description": "The execution enables the device to consume further
electricity.",
            "risk_score": 0.8
          }
        ]
      }
    }
  ],
  "risk_score": 0.8
}
```

```

    "api_name": "turn_oven_on",
    "description": "Turns on an oven at the last selected temperature.",
    "security_label": {
      "safety": [
        {
          "name": "FIRE_HAZARD",
          "description": "The execution may cause fire."
        },
        {
          "name": "AUDIO_VIDEO_STREAM",
          "description": "The execution authorises the app to obtain a video
stream with audio."
        },
        {
          "name": "POWER_OUTAGE",
          "description": "High instantaneous power. The execution may cause
power outage.",
          "risk_score": 0.8
        }
      ],
      "privacy": [
        {
          "name": "LOG_ENERGY_CONSUMPTION",
          "description": "The execution allows the app to register information
about energy consumption."
        }
      ],
      "financial": [
        {
          "name": "ELECTRIC_ENERGY_CONSUMPTION",
          "description": "The execution enables the device to consume further
electricity.",
          "risk_score": 0.8
        }
      ]
    }
  }
]
}

```

The tool produces in output a JSON file composed of the array of hazards which might be raised by the analysed application. Below is an example of an App Label extracted from a binary.

```

{
  "name": "app_name",
  "description": "app_description",
  "sifis_version": "0.1",
  "api_hazards": [
    {
      "api_name": "turn_light_on",
      "description": "Turns on a lamp.",
      "security_label": {
        "safety": [

```

```
{
  {
    "name": "FIRE_HAZARD",
    "description": "The execution may cause fire.",
    "risk_score": null
  }
],
"privacy": [
  {
    "name": "LOG_ENERGY_CONSUMPTION",
    "description": "The execution allows the app to register information
about energy consumption.",
    "risk_score": null
  }
],
"financial": [
  {
    "name": "ELECTRIC_ENERGY_CONSUMPTION",
    "description": "The execution enables the device to consume further
electricity.",
    "risk_score": 0.8
  }
]
}
]
```

As any other software developed in WP2, even this one is subject to the CI checks described in Section 2, in addition to a script for deployment and release.

4 A Complete Working Example

We focused on WoT and started working with the Webthings.io community, extending their `webthings-rust` and `webthings-arduino` implementations by writing an initial proof of concept of Thing extended with the SIFIS-Home Hazards ontology.

From this initial experience, we developed a set of separate crates to support the incoming WoT 1.1 standard. We also interacted with the more extensive W3C Web Of Thing community to keep track of the spec evolution and provide feedback on a few issues we found along the way.

4.1 *WoT Implementation in Rust*

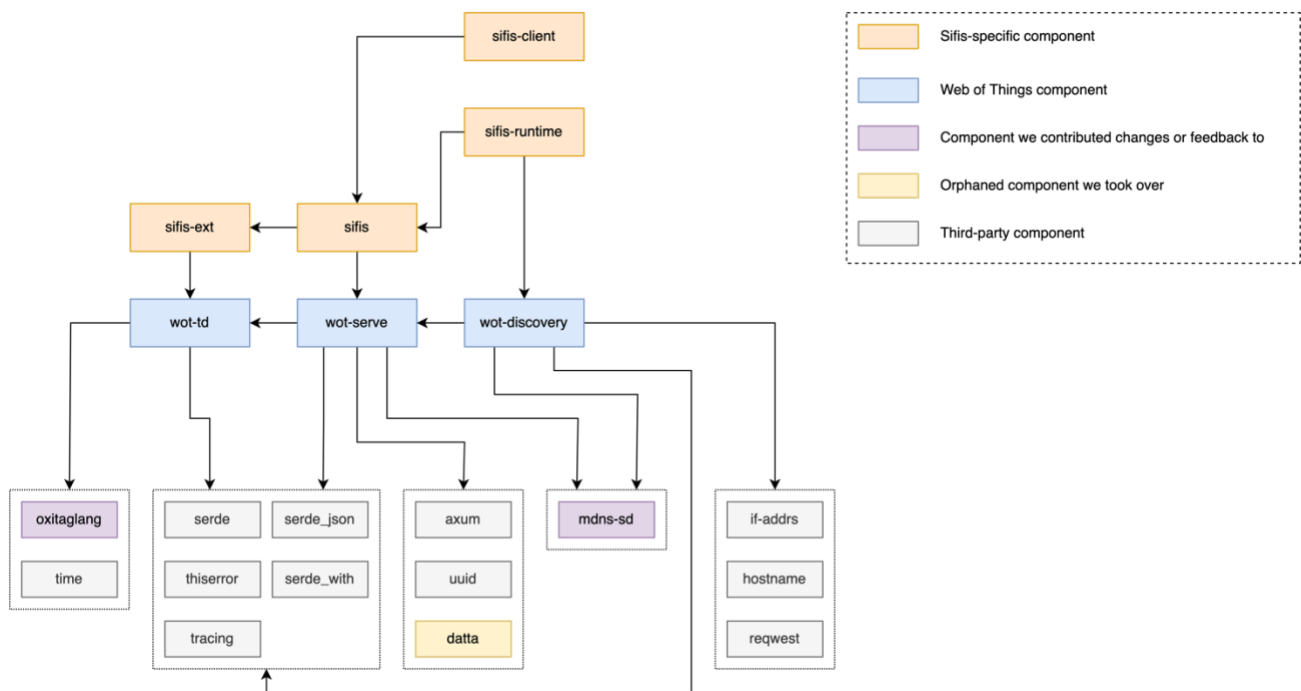


Figure 18: dependency diagram for the SIFIS-Home Rust framework

We use WoT as a foundation layer and build on top of it an implementation of the SIFIS-Home framework.

We split the WoT implementation into the following crates:

- **wot-td** that works on serializing, deserializing and extending the **Thing Description** in a type-safe way.
- **wot-serve** provides the building blocks to implement **Servients**, initially supporting building HTTP servients via **axum** and advertising its existence via multicast dns-sd.
- **wot-discovery** that provides the components to discover Things in the local network and build a directory.

Using them, we will reimplement our SIFIS-Home Hazards ontology as an Extension to the Thing

Description and build on top of it the high-level SIFIS-Home API described. The full dependency graph is shown in Figure 18.

During the initial development, we contributed to the following crates, besides the ones mentioned in Sections 2.4 and 2.5:

- **cargo-c** to make available the rust crates to other languages so that the *sifis crate* will be available as *libsifis* for the C/C++ consumers.
- **mdns-sd** used inside *wot-discovery* and *wot-serve* to advertise and discover Things.
- **derive_bounded** to reduce the boilerplate in the *wot-td* implementation.
- **datta** an implementation [rfc6570](https://www.rfc-editor.org/rfc/rfc6570) (<https://www.rfc-editor.org/rfc/rfc6570>).

4.2 Tools in Action

We used **sifis-generate** to create all the WoT projects, and we refined its Rust template from the experience of using it in this scenario.

We tried to keep the code coverage above 85%, aiming to stay well above 90% and ensure every pull request landing is clean of **clippy lints** (as described in D2.2). All the dependencies do not bring problems thanks to **cargo audit** and ensure that our implementation is spec-compliant as much as possible.

When we had to introduce **uritemplate** as a dependency, it triggered a good number of warnings, as shown in Figure 19.

```

> cargo audit
  Fetching advisory database from `https://github.com/RustSec/advisory-db.git`
  Loaded 457 security advisories (from /Users/lu_zero/.cargo/advisory-db)
  Updating crates.io index
  Scanning Cargo.lock for vulnerabilities (125 crate dependencies)
Crate:      regex
Version:    0.1.80
Title:      Regexes with large repetitions on empty sub-expressions take a very long time to parse
Date:      2022-03-08
ID:         RUSTSEC-2022-0013
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0013
Solution:  Upgrade to >=1.5.5
Dependency tree:
regex 0.1.80

Crate:      thread_local
Version:    0.2.7
Title:      Data race in `Iter` and `IterMut`
Date:      2022-01-23
ID:         RUSTSEC-2022-0006
URL:       https://rustsec.org/advisories/RUSTSEC-2022-0006
Solution:  Upgrade to >=1.1.4
Dependency tree:
thread_local 0.2.7
└─ regex 0.1.80

error: 2 vulnerabilities found!

```

Figure 19: cargo-audit output for uritemplate

4.2.1 datta

To implement a part of wot-serve, we had to implement a mapping between the [uritemplates](#) used in the WoT [Forms](#) and the axum [Paths](#).

The [uritemplate](#) would fit our needs, but it is not updated since six years ago with short-winded tries to update it by other parties, and as shown above, it is showing its age.

4.2.1.1 Shortcomings

- Pre-2018 codebase.
- Plenty of [clippy](#) triggers.
- Missing CI.
- Stale dependencies:
 - Regex and thread_local faults were caught by cargo audit.
 - Since we wanted to test how our sifis-generated CI behaved, we had [Miri](#) catch at least one problem while running the test suite.

4.2.1.2 Mitigation

- Since the original developer is not responsive and the crate is left in full neglect, we created a full fork of it.
- We made sure to update it to the Rust edition 2021.
- We addressed all the lints clippy found.
- We updated the dependencies, so the cargo audit report is clear.
- We set up the CI using the sifis-generate, as shown in Figure 20.

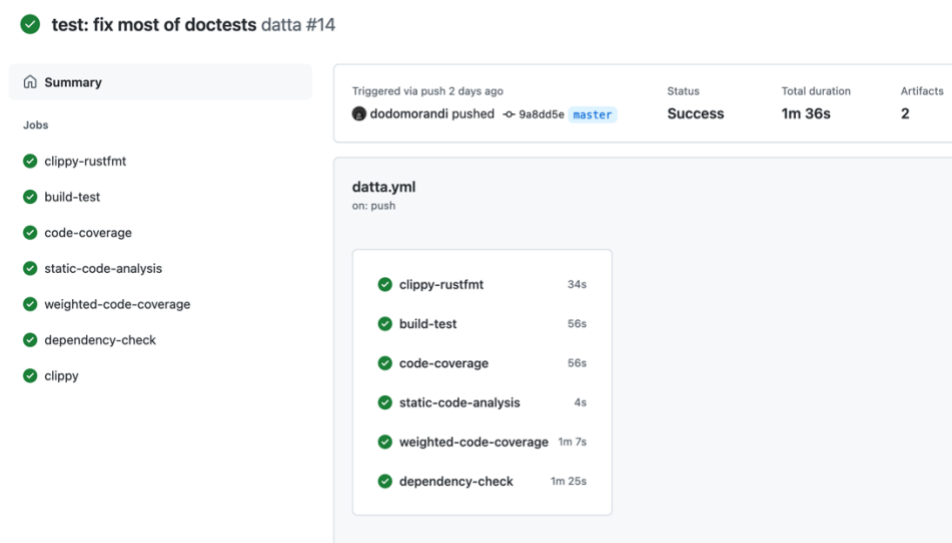
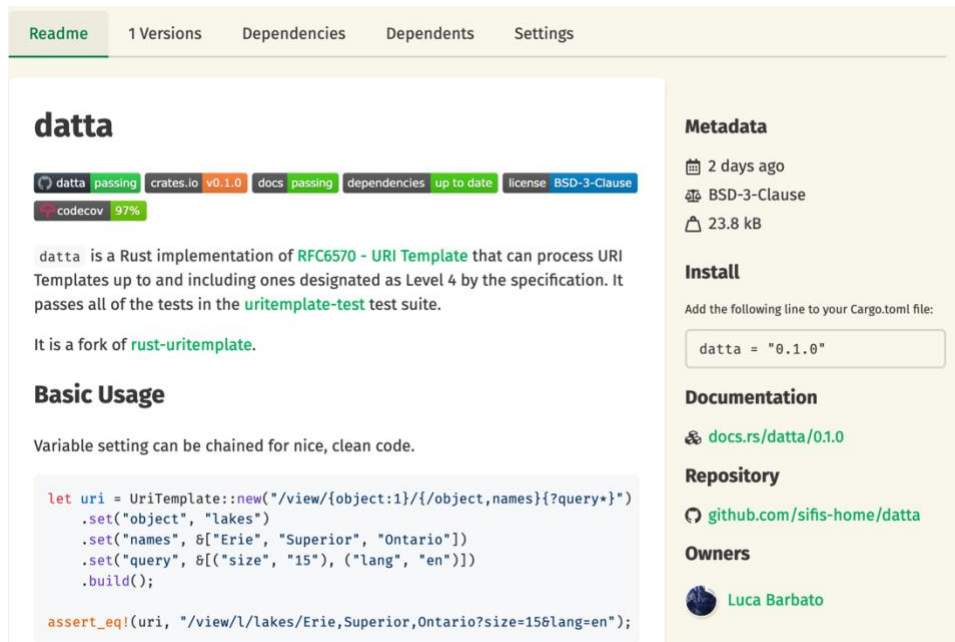


Figure 20: CI for datta

4.2.1.3 Updates and Release

We extended datta API to fit the wot-serve needs, updated documentation and released it with a new name, as shown in Figure 21.



datta

data passing crates.io v0.1.0 docs passing dependencies up to date license BSD-3-Clause

codecov 97%

datta is a Rust implementation of [RFC6570 - URI Template](#) that can process URI Templates up to and including ones designated as Level 4 by the specification. It passes all of the tests in the [uritemplate-test](#) test suite.

It is a fork of [rust-uritemplate](#).

Basic Usage

Variable setting can be chained for nice, clean code.

```
let uri = UriTemplate::new("/view/{object:1}/{/object,names}{?query*}")
    .set("object", "lakes")
    .set("names", &["Erie", "Superior", "Ontario"])
    .set("query", &[("size", "15"), ("lang", "en")])
    .build();

assert_eq!(uri, "/view/l/lakes/Erie,Superior,Ontario?size=15&lang=en");
```

Metadata

- 2 days ago
- BSD-3-Clause
- 23.8 kB

Install

Add the following line to your Cargo.toml file:

```
datta = "0.1.0"
```

Documentation

- [docs.rs/datta/0.1.0](#)

Repository

- [github.com/sifis-home/datta](#)

Owners

- [Luca Barbato](#)

Figure 21: crates.io for datta (<https://crates.io/crates/datta>)

5 Conclusions and Future Works

In this document, we described the tool released by WP2 at M24. We also provided the reader with additional information regarding the context in which these tools operate and some practical examples. All the described tools are released under the MIT license.

This document will also serve as an input for the next deliverable called D2.5 and planned to be delivered at M30, which will describe the development devoted to:

- Improving the tools described in this document.
- Implementing and releasing the dashboard for Agents that develop SIFIS-Home technologies described in D2.6 section 7.1.
- Implementing and releasing the license Conflict Mapper that will show possible conflicts among the software licenses of the software reused in the source code development process.
- Implementing and releasing the software bridges between the SIFIS-Home DHT and the WoT implementation and continuously refining its components in collaboration with WP5.

6 References

[TD, 2022] Thing Description (TD) Ontology. URL: <https://www.w3.org/2019/wot/td>

[SHO, 2022] The SIFIS-Home Hazards Ontology Specification. URL: <https://purl.org/sifis/hazards>

Glossary

Acronym	Definition
API	Application Programming Interface
CI	Continuous Integration
CPU	Central Processing Unit
GCC	GNU Compiler Collection
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
SHO	SIFIS-Home Hazards Ontology
SIFIS-Home	Secure Interoperable Full-Stack Internet of Things for Smart Home
TD	Thing Description
WoT	Web of Things
WP	Work Package