



D3.4 – INITIAL CPSWARM WORKBENCH AND ASSOCIATED TOOLS

Deliverable ID	D3.4
Deliverable Title	Initial CPSwarm Workbench and associated tools
Work Package	WP3 – Architecture design and Component Integration
Dissemination Level	PUBLIC
Version	1.0
Date	2017-11-30
Status	Final
Lead Editor	FRAUNHOFER
Main Contributors	Junhong Liang (FRAUNHOFER), Farshid Tavakolizadeh (FRAUNHOFER), Sisay Adugna Chala (FRAUNHOFER)

Published by the CSPwarm Consortium



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731946.

Document History

Version	Date	Author(s)	Description
0.01	2017-11-02	Junhong Liang (FRAUNHOFER)	First Draft with TOC and initial content
0.02	2017-11-22	Sisay Chala (FRAUNHOFER)	Added Introduction and Scope
0.03	2017-11-22	Junhong Liang (FRAUNHOFER)	Fixed typos
0.03	2017-11-24	Sisay Chala (FRAUNHOFER)	Added Section 3.3.4 and Figure 9, modified first sentence of Section 1.1 and fixed some typos
0.04	2017-11-26	Farshid Tavakolizadeh (FRAUNHOFER)	Added content to section 3.2
0.05	2017-11-29	Junhong Liang (FRAUNHOFER)	Added content to section 3.4
0.10	2017-11-30	Junhong Liang (FRAUNHOFER)	Minor modification; Fixed misspells, typos and formats; Released for internal review;
0.11	2017-12-04	Sisay Chal (FRAUNHOFER)	Modifications inserted to address few of the issues highlighted in the review from LAKE
0.12	2017-12-05	Farshid Tavakolizadeh (FRAUNHOFER)	Additional modifications inserted to address few of the issues highlighted in the review from LAKE
0.8	2017-12-06	Junhong Liang (FRAUNHOFER)	Modifications inserted to address remaining issues highlighted from LAKE and issues pointed out by ROBOTNIK
0.9	2017-12-08	Farshid Tavakolizadeh (FRAUNHOFER)	Fixed typo in figure
1.0	2017-12-08	Junhong Liang (FRAUNHOFER)	Final version to be submitted to the EC

Internal Review History

Review Date	Reviewer	Summary of Comments
2017-12-01 (v 0.10)	Micha Rappaport (LAKE)	Modifications inserted and comments
2017-12-05 (v 0.10)	Angel Soriano (ROBOTNIK)	Modifications inserted and comments

Table of Contents

Document History	2
Internal Review History	2
Table of Contents	3
1 Introduction.....	4
1.1 Scope	4
1.2 Related documents.....	4
2 CPSwarm Initial Components.....	5
2.1 Initial Modelling Tool.....	5
2.2 Initial CPS Modelling Library.....	7
2.3 Initial Swarm Modelling Library.....	8
2.4 Initial Simulation Environment.....	9
3 CPSwarm Initial Components Integration	12
3.1 Component Integration and Interfaces.....	13
3.2 Continuous Integration (CI) Platform.....	14
3.3 Integration Test Setup.....	18
4 Conclusions.....	27
Acronyms	28
List of figures.....	28
References.....	29
Annex A: Dockerfile for Modelio	30
Annex B: Dockerfile for FREVO.....	31
Annex C: Dockerfile for Minisim	32

1 Introduction

This document is prepared to support the CPSwarm project in defining, documenting, and implementing a software testing methodology and test management governance framework. It can be adopted in order to achieve improvement in overall software quality.

This document provides the initial workbench and associated tools under which CPSwarm project can operate to deliver testing services to the various system components. This ensures that all products satisfy the requirements of design specifications and performance criteria based on the requirements specified in D2.3.

The overall software testing goal is to produce high-quality systems using a set of managed and controlled processes, which meet the requirements and expectations. This document establishes a Continuous Integration (CI) software testing methodology and associated tools for CPSwarm.

1.1 Scope

This document elaborates the requirements described in the Initial Requirements Report (D2.3) and the CPSwarm Test and Integration Plan (D3.7). It furthermore provides the required components, i.e., Initial CPSwarm Modeling Library (D4.1), Initial Swarm Modeling Library (D4.4), Initial Swarm Modeling Tool (D5.2), Initial Simulation Environment (D6.1), and their interactions for the integration and system tests.

A public prototype deliverable is produced by Task T3.3, which documents the integration results of the Initial CPSwarm Workbench and associated tools. In order to show the context of this deliverable, the input-process-output relationship between this Integration test document and the related deliverables and tasks are shown in the Figure 1.

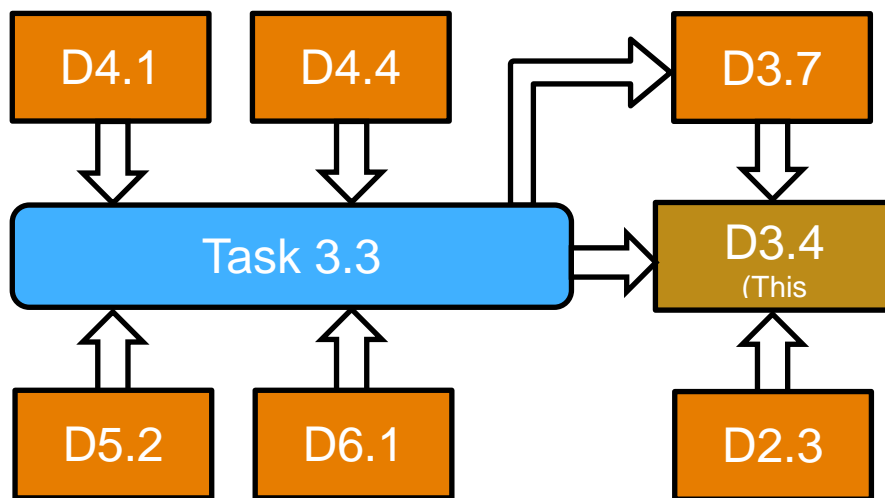


Figure 1 - Relationship of this document with other deliverables and Tasks

1.2 Related documents

ID	Title	Reference	Version	Date
[RD.1]	Initial Requirements Report	D2.3	1.0	2017-06-30
[RD.2]	Test and Integration Plan	D3.7	1.0	2017-09-30
[RD.3]	Initial CPSwarm Modeling Library	D4.1	1.0	2017-09-30
[RD.4]	Initial Swarm Modeling Library	D4.4	1.0	2017-10-31
[RD.5]	Initial Swarm Modeling Tool	D5.2	1.0	2017-09-30
[RD.6]	Initial Simulation Environment	D6.1	1.0	2017-09-30

2 CPSwarm Initial Components

In the first integration phase of CPSwarm, the project team focused on incorporating initial components regarding the modelling and optimization aspects. According to the document of action, components developed in D4.1, D4.4, D5.2, and D6.1 should be integrated until the end of M11, which include the initial CPS Modelling Library, the initial Swarm Modelling Library, the initial Modelling Tool, as well as the initial Simulation Environment. They are highlighted in Figure 2. It is important to notice that the components defined in the deliverables do not have a one-to-one match to the architecture diagram. In fact, the Modelling Library in the architecture diagram includes both the initial CPS Modelling Library (D4.1) and the initial Swarm Modelling Library (D4.4). While the Modelling Tool in the diagram match the initial Modelling Tool defined in D5.2, the Algorithm Optimization Environment in the diagram matches the Simulation Environment defined in D6.1 (see notations on Figure 2).

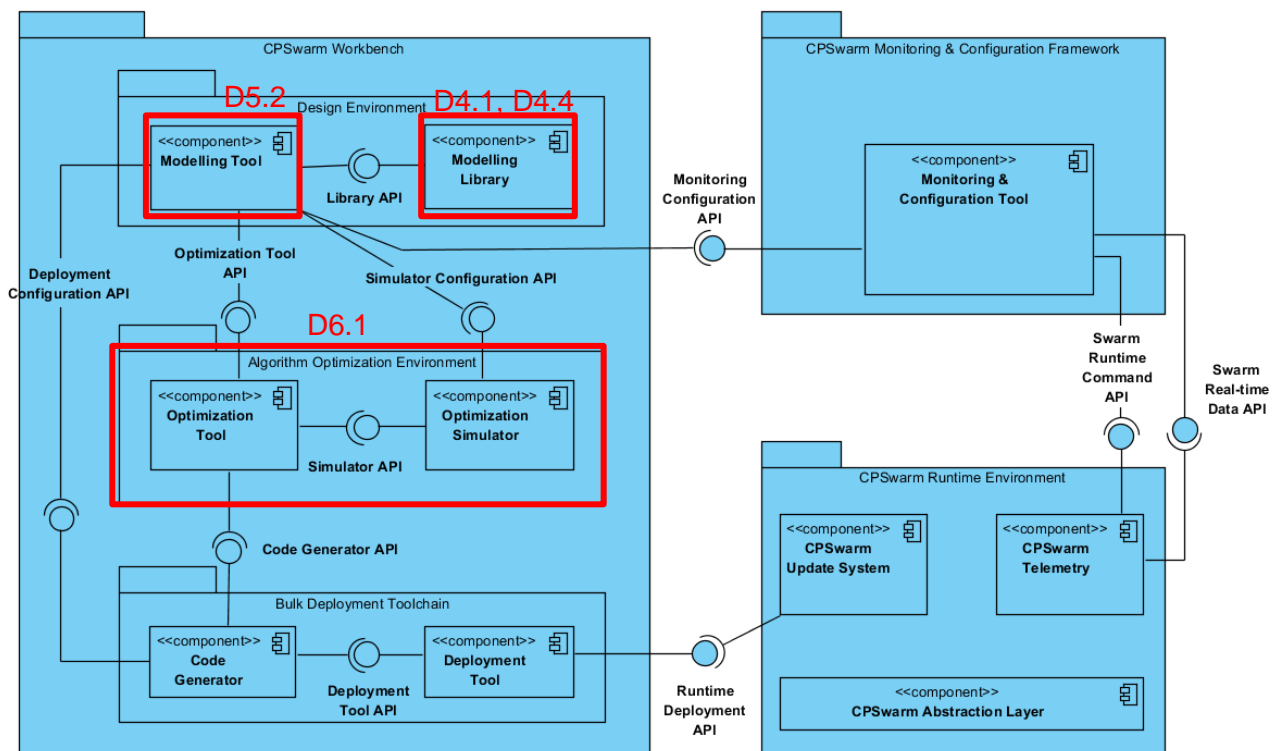


Figure 2 - CPSwarm system architecture extracted from D3.1 (Components to be integrated are highlighted in red rectangles. The annotations show the mapping between components in the diagram to the components defined in deliverables)

These components are chosen for the first integration phase because they are the crucial components for the beginning of the whole CPSwarm system workflow, namely modelling and optimization. The integration of these components builds a sound foundation for later development.

This chapter provides a brief overview of the aforementioned components' implementations, so that readers can conveniently have a big picture of the current development progress. Further implementation details are documented in the respective deliverables.

2.1 Initial Modelling Tool

The CPSwarm Modelling Tool is built on top of the open source graphical modelling environment named Modelio¹, whose graphical user interface is shown in Figure 3. Modelio is developed by SOFTEAM and it delivers a broad range of standards functionalities and is capable of modelling standard diagrams such as

¹ <https://www.modelio.org/>

UML2, BPMN², MARTE³, SysML⁴, etc. Modelio's architecture is built on a plugin-based framework, around which extensions are defined to fulfil different functionalities. This architecture allows the Modelio modelling environment to be flexible and configurable simply by adding the desired extension and related functionalities.

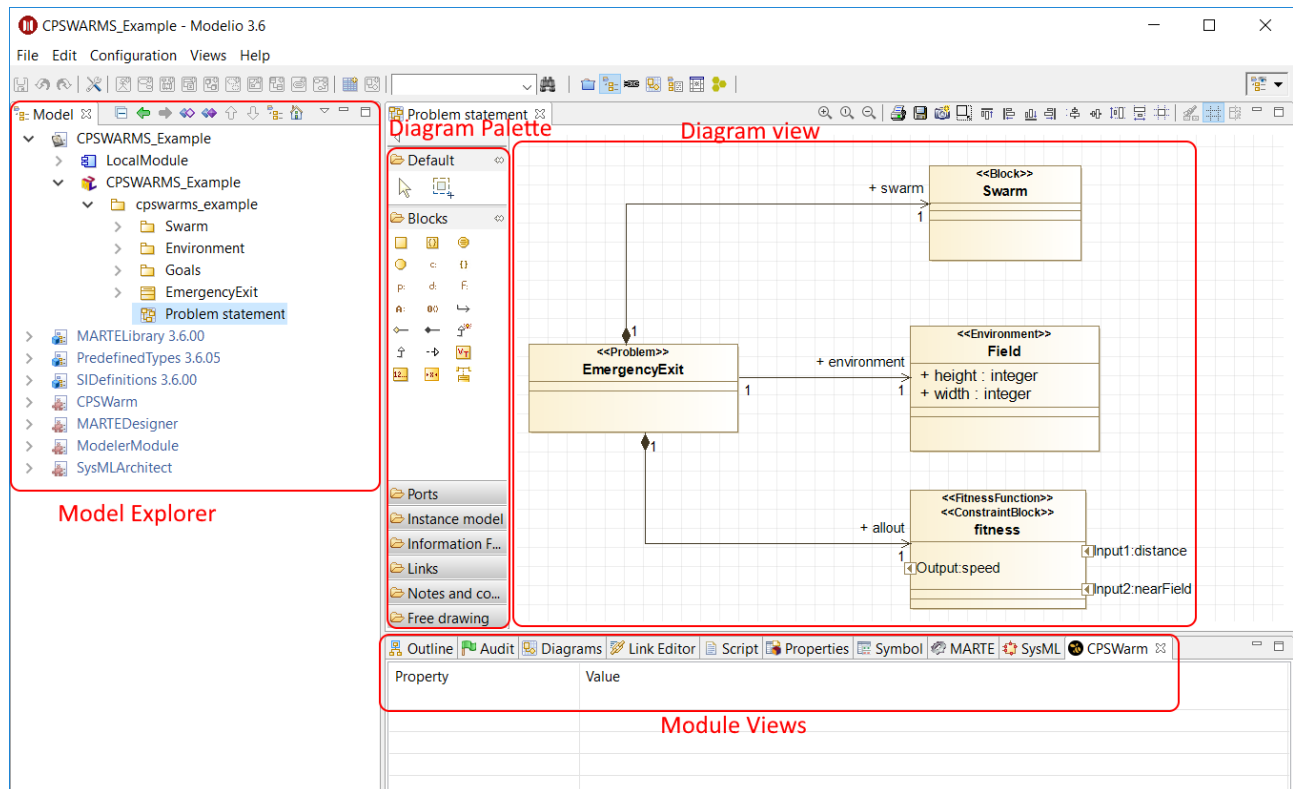


Figure 3 - Modelio graphical user interface

As depicted in Figure 4, the CPSwarm Modelling tool is composed of Modelio itself, a dedicated CPSwarm extension to provide the functionalities related to CPS swarm design, and a set of pre-existing extensions to reuse their relevant functionalities in the CPSwarm context. At M11 of CPSwarm, only the SysML extension has been chosen for its functionalities related to system modelling.

² <http://www.bpmn.org/>

³ <http://www.omg.org/omgmarte/>

⁴ <http://www.omg.sysml.org/>

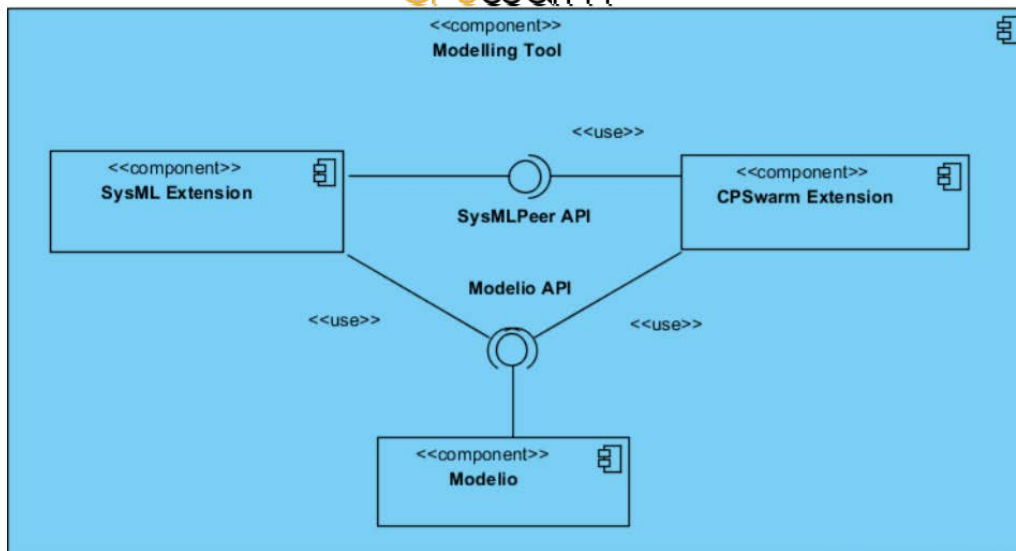


Figure 4 - Subcomponent diagram of the Modelling Tool

2.2 Initial CPS Modelling Library

When designing a swarm consisting of CPSs, modelling the individual CPSs (i.e. swarm members) in the swarm is necessary so that the designer can define the composition of the swarm. For this purpose, the CPS Modelling Library is defined to help easing development and integration of complex swarms of heterogeneous CPSs by providing existing and reusable components and models to the modeller. The overall idea is to have a library for the modeller that contains certain predefined models. These models can be reused, changed, or added by the modeller.

At M11, three groups of libraries are defined in the CPS Modelling Library: swarm member, environment, and goal libraries.

- **Swarm member library:** The swarm member library describes an individual CPS used in swarm applications. Following sub-libraries are available:
 1. local memory
 2. behaviour
 3. physical aspects
 4. security (optional)
 5. human interaction (optional)
- **Environment library:** The environment library describes the environment in which the swarm of CPSs is acting. Several models express an environment, whereby the following ones are indispensable (further ones can be added if necessary):
 1. 2D/3D map of the environment
 2. Size of the environment
 3. Resolution
- **Goal library:** The goal library describes the goal that the swarm of CPSs wants to reach. The goal is expressed by a fitness value and a calculation specification. The calculation is done by incorporating parameters from other models. If the application asks for it, multiple fitness values can be modelled, possibly related to each other.

Currently these libraries are built-in into the CPSwarm extension for Modelio so that they have seamless integration with the Modelling Tool. When the user installs the CPSwarm extension, the libraries are available for use in Modelio.

2.3 Initial Swarm Modelling Library

Another aspect of modelling a swarm of CPSs is to model the intelligent behaviour of the swarm as a whole. In CPSwarm a common modelling standard for swarm behaviour has been introduced. The main idea is to have a formulation and definition of models in SysML visually representing:

- swarm intelligence algorithms
- individual behaviours of existing swarm intelligence algorithms for customization

The Swarm Modelling Library is defined which helps the designer to model the behaviour of a swarm. This library comprises existing swarm intelligence algorithms such as the honeybee algorithm [1], the BEECLUST algorithm [2], or the firefly algorithm [3] from a high-level view. Figure 5 shows a model to represent a swarm member with BEECLUST algorithm.

At M11, the intelligent models found in the Swarm Modelling Library include the following aspects:

- a description of their functionality
- defined inputs and outputs
- defined local states (if necessary)
- deposited Java and C++ code (in a first version: pseudocode)

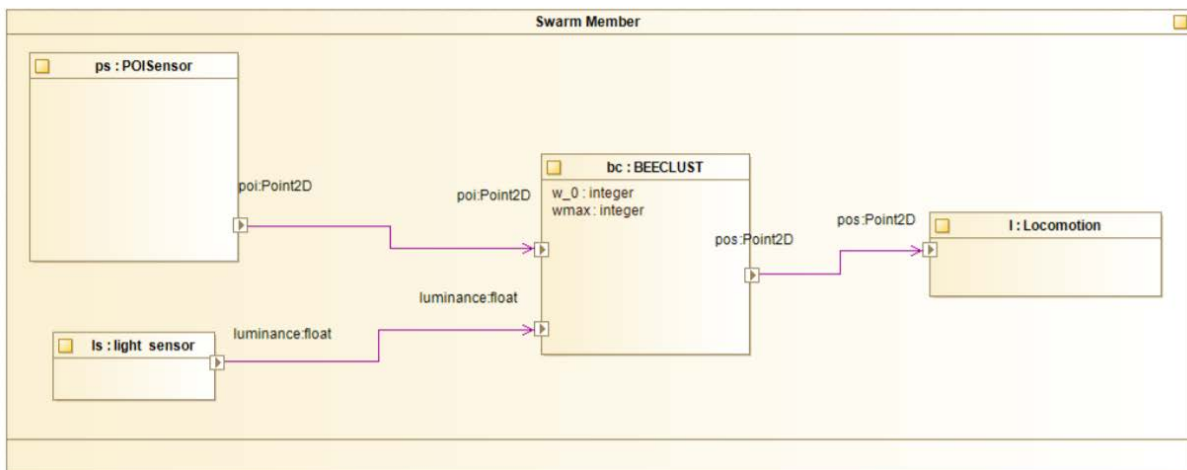


Figure 5 Model to represent a swarm member with BEECLUST

Typically, real-world applications come with needs that cannot be directly modelled with an existing nature inspired swarm intelligence algorithm. Therefore, it is useful to have a process that allows constructing customized swarm intelligence algorithms. This is enabled by a library that provides single behaviours extracted out of given swarm intelligence algorithms.

The Modelling Tool allows to construct a state machine with all the required behavioural elements. An example is visualized in Figure 11.

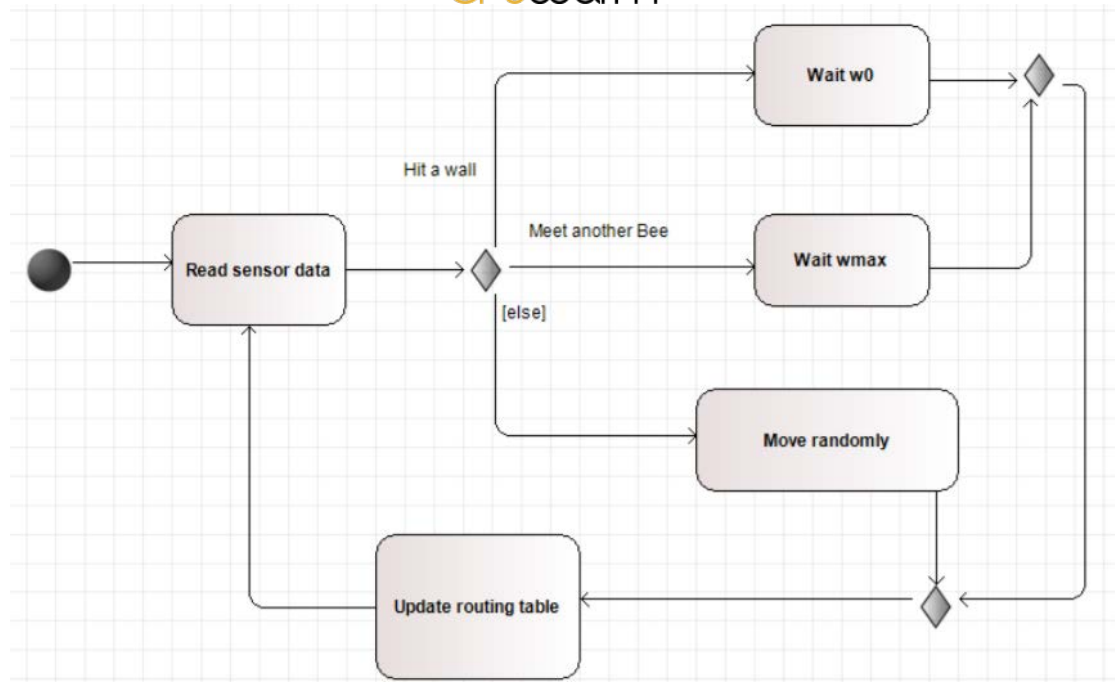


Figure 6 Caption missing

Currently the swarm modelling libraries are still in conception phase and are hence not yet fully implemented. New implementation details will be added in future deliverables.

2.4 Initial Simulation Environment

The initial Simulation Environment includes the optimization tool and the optimization simulator highlighted in Figure 2. These two components are implemented using different existing tools. FREVO⁵ is chosen to implement the optimization tools, while various simulators are chosen as candidates for the optimization simulator.

FREVO (Framework for evolutionary design) is a tool developed by LAKE and KLU for evolving and evaluating self-organizing systems using evolutionary algorithms [4]. Users can define the problem, the controller representation, the optimization method, and the ranking method through the FREVO GUI (see Figure 7). FREVO runs the evolutionary optimization which creates candidate solutions to the problem. The candidates are evaluated through simulations that implement the problem definition. The best performing candidates are evolved in the optimization process leading to an optimized solution after multiple iterations. This solution is then selected as the optimized algorithm to solve the problem.

⁵ <http://frevo.sourceforge.net/>

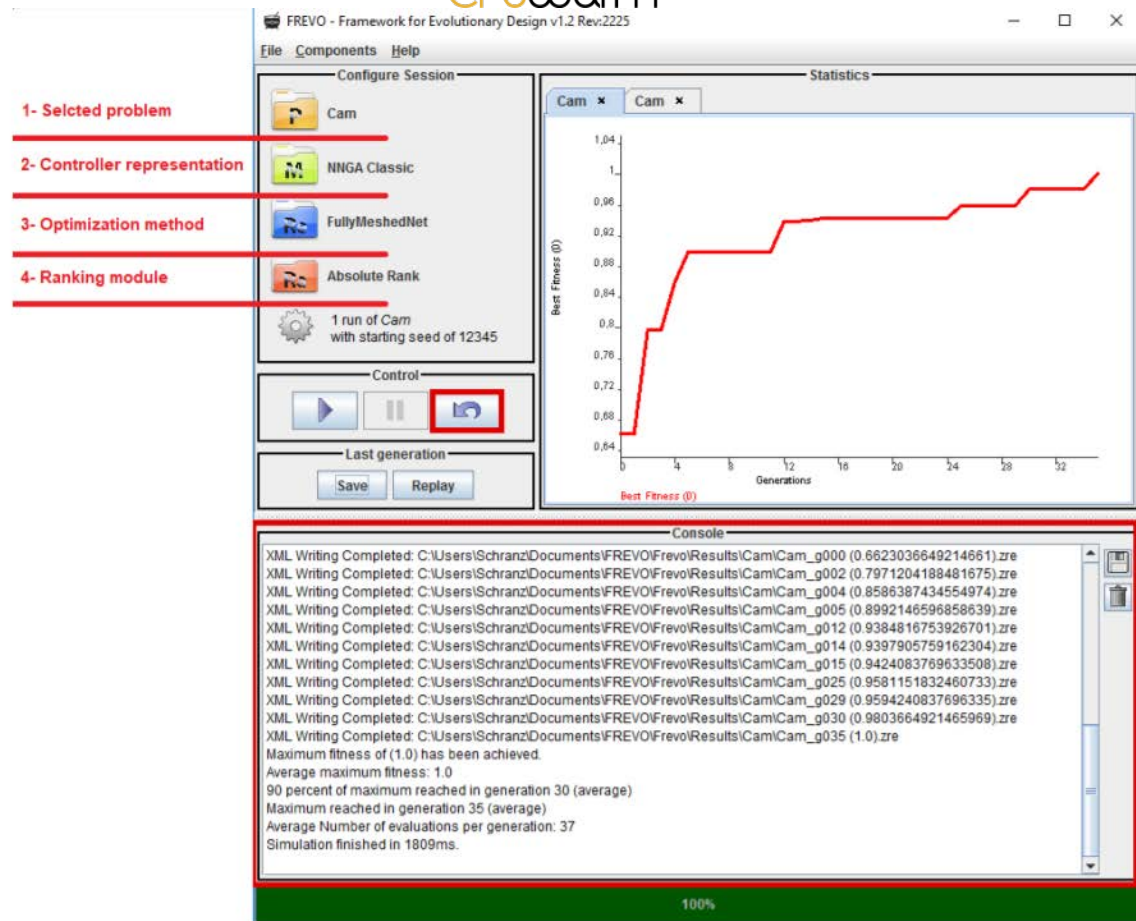


Figure 7 - FREVO graphical user interface

In the first integration phase, the ROS⁶-based Stage⁷ simulator and Gazebo⁸ simulator are chosen as the optimization simulator.

Stage is a low-fidelity two-dimensional robot simulator. It provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate. A screenshot of such a simulation can be seen in Figure 8.

⁶ <http://www.ros.org/>

⁷ <http://playerstage.sourceforge.net/doc/stage-svn/>

⁸ <http://gazebo.org/>

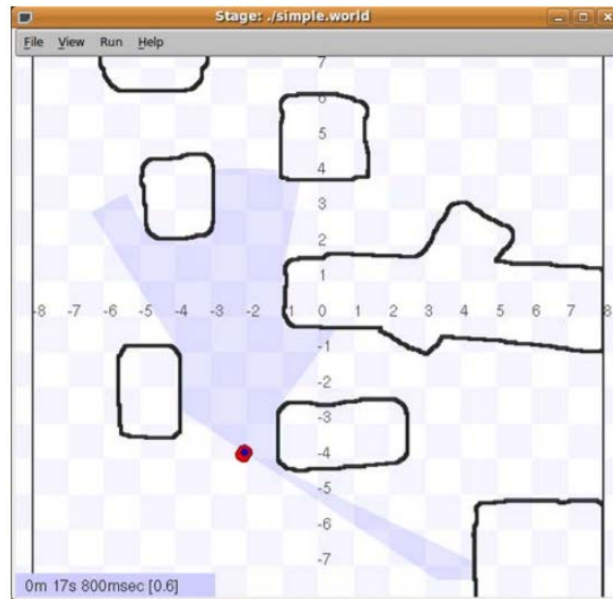


Figure 8 - Stage simulator

Gazebo is a 3D high-fidelity simulator for robotic applications. It provides many features such as dynamic robotic simulation, sensor simulation, etc. Figure 9 shows a screenshot of the Gazebo simulator.

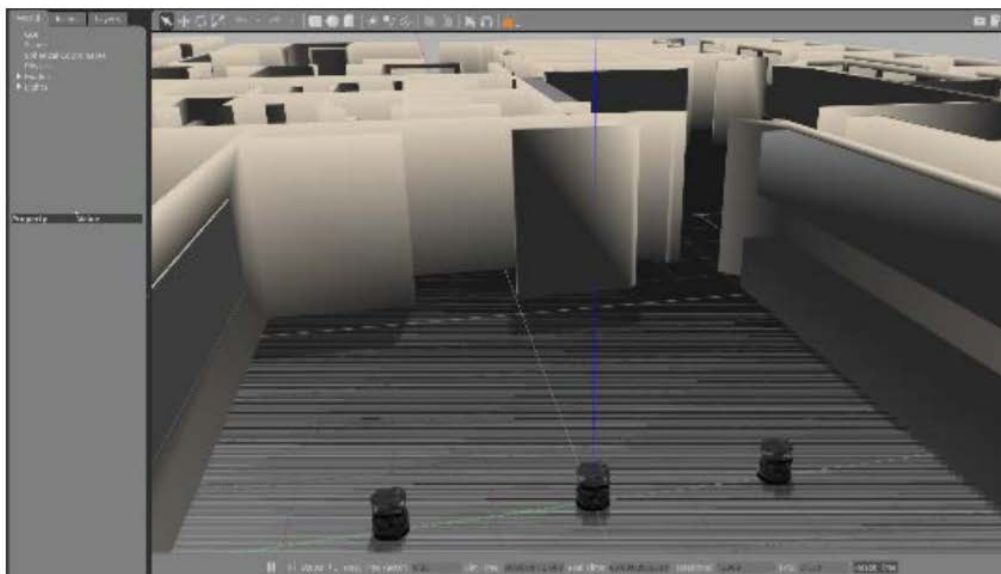


Figure 9 - Gazebo simulator

In order to have an easy and convenient evaluation of the communication between the Optimization Tool and the Optimization Simulator, the project team developed a simple, single-purpose simulator called Minisim. The Minisim simulator serves as an initial tool to evaluate possible communication options without requiring the developers to dive too deep into the implementation details of Stage or Gazebo.

3 CPSwarm Initial Components Integration

In the initial CPSwarm components integration, components described in Chapter 2 are interconnected as shown in Figure 10. The CI test takes detailed integration instructions or integration scripts, related dependencies, and test cases together with the software to be tested. Successful execution of the integration test produces the latest build of the system and a notification that the process is successfully completed. However, if the integration test execution is unsuccessful, an issue is initiated and a notification about the status of integration test is produced.

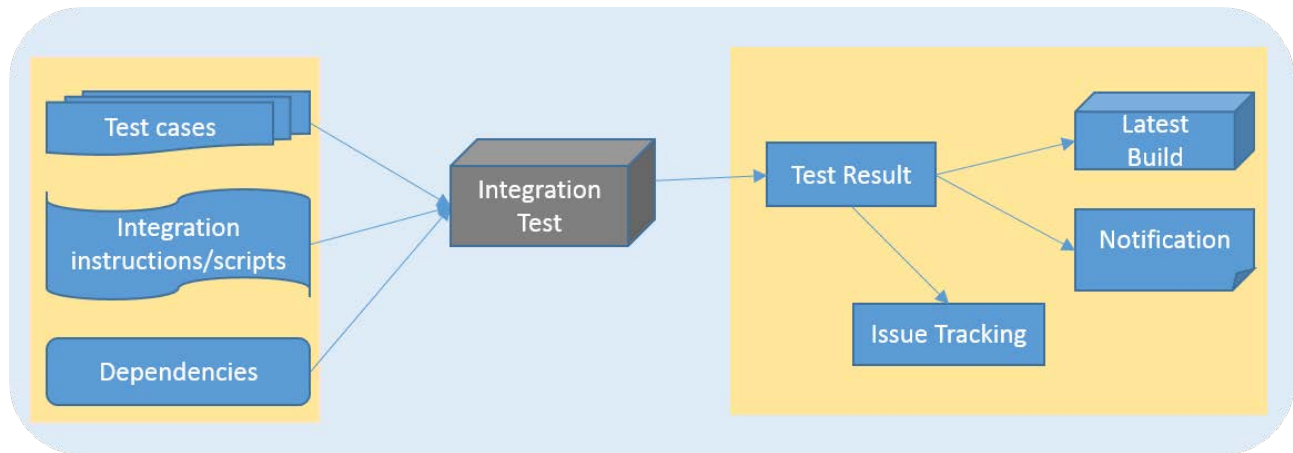


Figure 10 Integration Test Set Up

In this chapter, aspects regarding the integration of the aforementioned components are described. First, the interface of each component will be described in section 3.1. Then, the basic principles and tools used for CI will be introduced in section 3.2. Finally, the setup of CI tools will be presented in section 3.3.

3.1 Component Integration and Interfaces

The current interfaces between the initial components are depicted in Figure 11. It is worth mentioning that since the initial architecture was defined in D3.1, some limitations have been discovered in the evolvement of the development process. Therefore, modifications have been made to the initially defined architecture. Although not yet officially documented (since the updated architecture will be documented in the future deliverable D3.2 in M16), the changes have been applied to the definition of interfaces for the initial CPSwarm components as seen in Figure 11.

Comparing the Figure 11 with the relevant part in the initial architecture diagram (Figure 2), the major difference found is that there is no direct interface between Modelling Tool and Optimization Simulator. The reason for this change is the following: At the time of defining the initial architecture, the Modelling Tool was conceived as a central configuration user-interface for all the components in the CPSwarm system. Therefore, the configuration of the Optimization Simulator should have been done in the Modelling Tool and an interface between these two components was necessary for passing the configuration data. However, limitations of such approach have been discovered. The limitations include the following:

- **Complication by heterogeneous interfaces:** interfaces to communicate with different components could vary significantly, e.g. currently the Modelling Tool and the Optimization Tool use a file-based interface, while the Optimization Simulator uses an MQTT-based interface. This means that if the Modelling Tool is to interact with all other components, the aforementioned interface functionalities must be built in this single component, which makes this component over-complicated and hard to maintain.
- **Coupled inputs from multiple components:** in the initial architecture diagram, the Optimization Simulator receives coupled information from two interfaces: the Simulator Configuration API and the Simulator API. It has been proven that it is difficult to handle different inputs consistently. Instead, if the coupled inputs come from only one interface, it would be much cleaner and easier to handle.

Since then, the interface between the Modelling Tool and the Optimization Simulator has been removed.

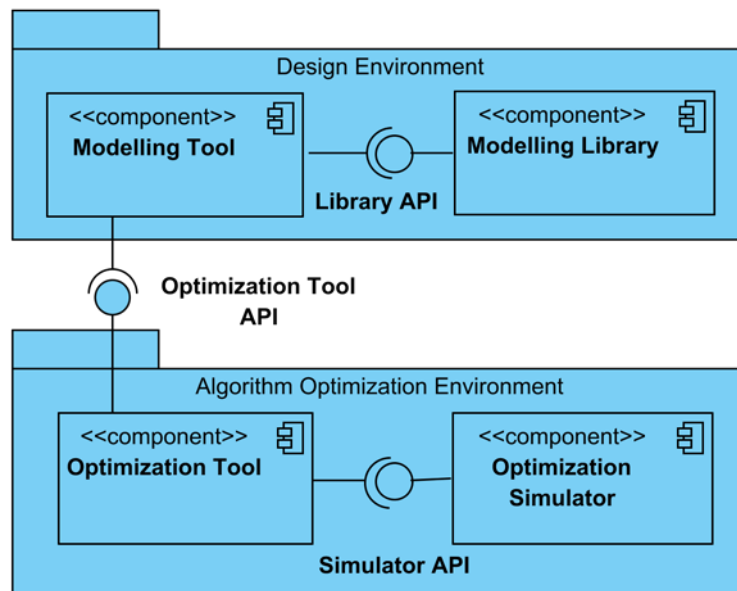


Figure 11 - Interfaces between the CPSwarm initial components

In the following sub-sections, the interfaces between these initial components, i.e., Library API, Optimization Tool API and Simulation API are described respectively.

3.1.1 Library API

The libraries related to CPSwarm are bundled as a single CPSwarm extension. This extension can be installed into a project of the Modelling Tool Modelio as a plugin during runtime. Once installed, the functionalities as well as the models in the libraries can be used within that project. Since Modelio is built upon a plugin framework, that plugin framework provides the mechanism for interaction between the Modelling Tool and the Modelling Library.

3.1.2 Optimization Tool API

Currently, the Modelling Tool Modelio and the Optimization Tool FREVO communicate via a file-based method. After the modelling is finished, files containing modelling and configuration data are generated by the Modelling Tool and these files are read in by the Optimization Tool.

In the current implementation, the Modelling Tool generates two files: one XML file and one Java source code file. The XML file contains modelling data necessary for the Optimization Tool and the Java source code file specify how the data should be read by the FREVO. These two files are then included in the FREVO's build path and FREVO needs to be recompiled to take the models into account.

It is worth mentioning that this approach serves merely as initial exploration and evaluation for the interface between the Modelling Tool and the Optimization Tool. More advanced methods for passing data from the Modelling Tool to the Optimization Tool will be explored in the future to enhance usability. This allows a more seamless integration of these two components, e.g., a recompilation of FREVO is not necessary to parse the data.

3.1.3 Simulator API

For scalability reason, MQTT has been chosen as the communication interface between the Optimization Tool and the Optimization Simulator. With this approach, the CPS controller is executed in the Optimization Tool. It acts as the brain for the simulated robots in the Optimization Simulator and receives simulated sensor signals from and sends action commands to the simulated robots via MQTT messages. For a more detailed description of the MQTT interface please refer to D6.1.

3.2 Continuous Integration (CI) Platform

As mentioned in D3.7, CPSwarm adopts the CI as the software integration strategy. In order to realize this approach, several off-the-shelf software solutions are used. Furthermore, state-of-the-art techniques are implemented in order to achieve robust software testing with minimum maintenance requirements and maximum fault tolerance.

3.2.1 CI Platform Deployment

The CPSwarm CI platform consists of several software components responsible for different aspects of the continuous integration process. All the utilized components are free to use for open source projects.

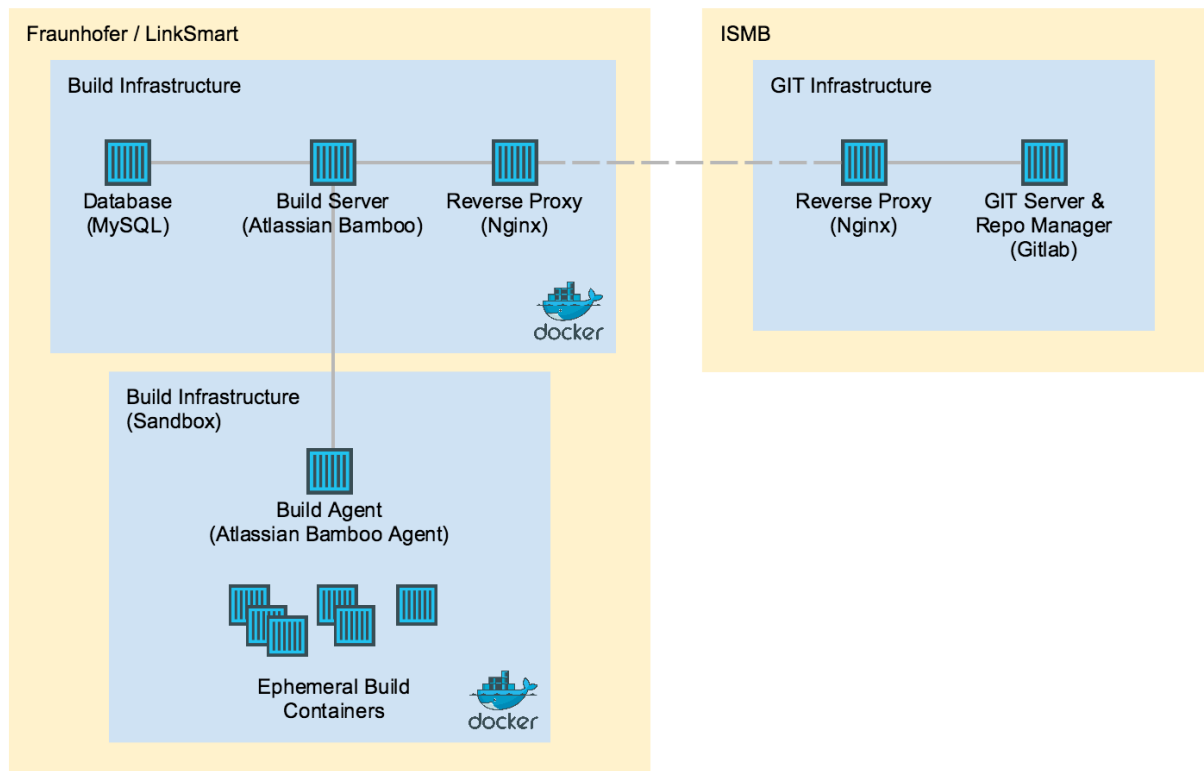


Figure 12. CPSwarm Initial Continuous Integration (CI) Platform. Two independent networks (yellow) consisting of virtual machines (blue), hosting containerized applications.

The components of the CPSwarm CI platform are administrated by the consortium members. In particular, ISMB provides the infrastructure for code management while FRAUNHOFER develops and maintains the build system. Figure 12 illustrates the deployment model of the platform including the main software components. These components are described below.

GIT Infrastructure

The GIT server and management User Interface (UI) is provided by an instance of GitLab⁹, a leading open-source software for code management and continuous integration. The instance is deployed and maintained by ISMB. Furthermore, an Nginx¹⁰ engine performs SSL termination and securely retrieves local resources on external requests. In the CPSwarm CI platform, we only utilize the code management features of GitLab and realize continuous integration using other tools. The GitLab instance is available to consortium members at ISMB PerT Area Git Repository¹¹. Members can perform push/pull git operation on projects created by them and those which they have given write permissions. Pull operations are allowed on all other projects related to CPSwarm. These operations can be performed over HTTPS as well as SSH. Additionally, members can use the management UI to create and modify projects, view code and branching history, and manage access rights.

Build Infrastructure

From a functional view, the build system is composed of a build server, at least one build agents, and any number of ephemeral build containers. These functional suites consist of one or more components that are

⁹ <https://gitlab.com>

¹⁰ <https://nginx.org>

¹¹ <https://git.repository-pert.ismb.it>

containerized with Docker¹². The containerization enables component isolation and portability. Each component is further explained below:

- **Build Server:** The build server is an instance of Atlassian Bamboo¹³, a professional tool for continuous integration, deployment, and delivery. Atlassian offers free licenses of Bamboo to projects that are open-source and public [5]. We currently utilize a Bamboo instance deployed as part of the LinkSmart®¹⁴ ecosystem. The instance is accessible at LinkSmart Pipelines¹⁵. Bamboo is connected to the GitLab server, listening to changes in the source codes. Currently we perform polling every five minutes. Depending on the configuration, the developers will be notified about the status of successful and/or failed builds by email.
- **Database:** The build server uses a MySQL¹⁶ database server to store build plans, logs, and other service information.
- **Reverse Proxy:** The build server runs behind an Nginx reverse proxy serving external requests. Requests are secured with SSL termination using an external component.
- **Build Agent:** A build agent is an Atlassian Bamboo Agent, responsible for performing builds and different kinds of tests. Each agent can perform one job at a time. Currently, the system has one agent in deployment but it can be easily replicated to allow parallelized builds and tests. The build agents subscribe to a broker exposed by the build server to be informed about build jobs. Once a job is published, agents start picking and executing tasks and publish the resulting logs and artefacts to the build server. The execution of tasks is done in ephemeral build containers. For better isolation, the build agent resides in another virtual machine (VM), which is separated from the VM other services run on. This way, even if errors happen on the build agent, which corrupt the VM, other build services would not be affected.
- **Ephemeral Build Containers:** The Docker containers are created for a specific job and removed upon job completion. A job may create more than one container in order to perform integration tests that require multiple running services. In any case, all containers related to a job are destroyed after job success or failure. The ephemeral container approach helps to isolate build tasks from each other. Furthermore, it ensures that tests are not influenced by each other or the hosting operating system of the build system. The ephemeral build containers reside in the same VM as the agent.
- **Issue Tracking:** One aspect of continuous integration is to link build results into actions and responsibilities. In the CPSwarm CI platform, we are looking into the possibility of integrating builds with an issue tracking system so that developers are informed and assigned to builds that are failed by their pushed codes or artefacts. The final realization of the issue tracking system will be reported in future deliverables.

3.2.2 CI Platform Guidelines

As mentioned in the previous section, Bamboo has been used as the continuous integration tool. Bamboo is a powerful and flexible system that allows different kinds of implementation based on different use cases. In order to properly utilize Bamboo for continuous integration, it is essential to get familiar with the Bamboo environment and terminology.

¹² <https://www.docker.com>

¹³ <https://www.atlassian.com/software/bamboo>

¹⁴ LinkSmart® is a trademark used by Fraunhofer for IoT software utilities

¹⁵ <https://pipelines.linksmart.eu>

¹⁶ <https://www.mysql.com>

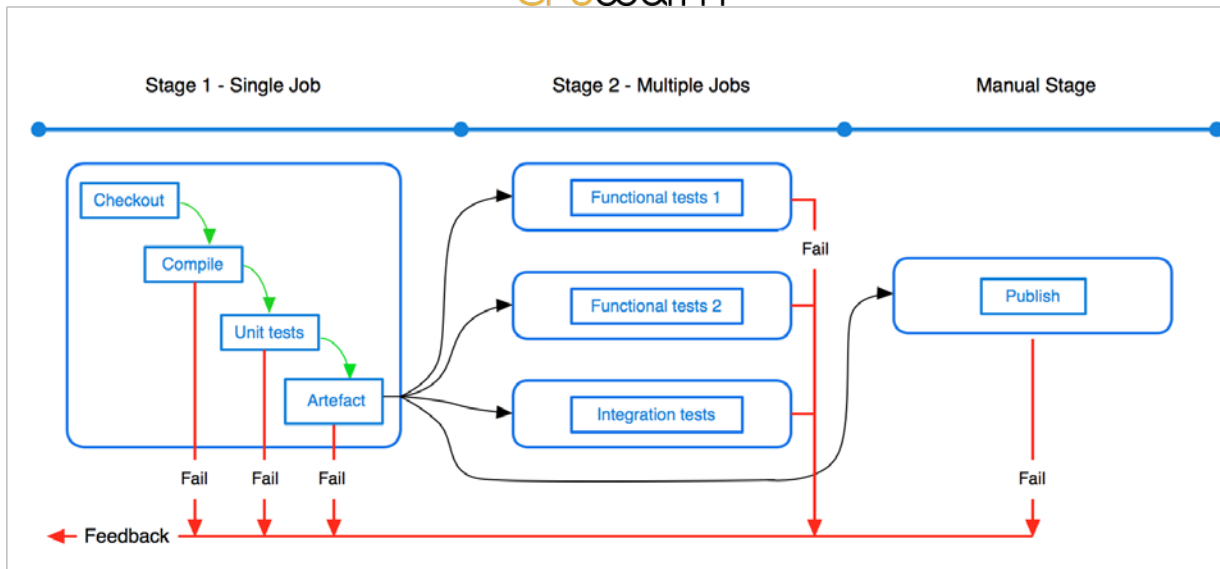


Figure 13. Multi-stage Bamboo plan [6]

At the highest level of abstraction, Bamboo divides the CI workflow into build and deployment projects. Build projects contain instructions for building, testing, and publishing software snapshots. Deployment projects include instructions for building and publishing releases as well deploying them on target systems.

A build project is a logical grouping for a set of build plans. We have created one project for CPSwarm. Build projects are structured as below:

- **Build plan:** Plans are independent instructions with separate triggering mechanisms. Each plan consists of one or more stages. Each implemented and testable CPSwarm component has at least one plan. Plans are triggered manually, after changes detected in the source code, or following a predefined schedule.
- **Stage:** Each stage within a plan represents a step within in the build process. A stage may contain one or more jobs which Bamboo can execute in parallel. For example, there can be a stage for compilation jobs, followed by one or more stages for various testing jobs, followed by a stage for deployment jobs.
- **Job:** A group of tasks with shared requirements resulting in one or more artefacts.
- **Task:** A piece of work that is executed as part of a job. Check out source code, the execution of a script, and a shell command are only few examples of tasks.

Figure 13 illustrates a build plan with three stages. Stage 1 consists of a single job with four tasks. Each task leads to the next and a failure at any tasks will break the job and send a feedback. Stage 2 contains three parallel jobs with single tasks all triggered after the successful completion of Stage 1. Failure of each job is reported. Stage 3 is a manual stage that can be triggered after the success of Stage 1. This stage has one job with a single task.

A deployment project allows defining tasks similar to build plans but designated for a specific target environment. In other words, deployment projects consist of one or more environments, each with a single job. The job includes a set of tasks in order to build and deploy the project to the target environment. Deployment projects can be triggered manually or automatically after a successful build plan.

In CPSwarm, different components are separated into different plans because they are developed by independent parties and managed in separate code repertories. Furthermore, during build-time tests, Docker containers are utilized to contain components for better dependency isolation. A Docker¹⁷ container is a

¹⁷ <https://www.docker.com/>

lightweight, stand-alone instance of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, and settings. Available for both Linux and Windows based apps, containerized software always runs the same, regardless of the environment. Containers isolate software from its surroundings, for example, differences in development and staging environments. They help to reduce conflicts between teams running different software on the same infrastructure.

3.3 Integration Test Setup

This section presents the setup for integration tests regarding the CPSwarm project. Since we leverage Docker for testing, the dockerization details are documented in 3.3.1. In 3.3.2, the component tests we perform are demonstrated. In 3.3.3, the setup of all the Bamboo plans and their relationships are explained in details. Finally, in 3.3.4, the results of integration tests are presented.

3.3.1 Dockerization of CPSwarm Components

Dockerization is the process of installing a software along with its dependencies into a Docker image. Such Docker images are typically built using Dockerfiles. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using the command `docker build` users can create an automated build that executes several command-line instructions in succession to build an image from a Dockerfile. In this section, the dockerization details of the three CPSwarm components are documented in each of the following sub-section, respectively.

3.3.1.1 Dockerization of Modelio and its CPSwarm Extension

The initial Modelling Tool, initial CPS Modelling Library, and initial Swarm Modelling Library are delivered as the CPSwarm extension of Modelio. In order to build a container in which this extension can be tested, Modelio must be installed inside the container. Currently, Modelio version 3.6.1 is compatible with the CPSwarm extension. As the kernel of the Docker container is Linux-based, the Linux version of Modelio must be used. Therefore, Modelio 3.6.1 for Ubuntu was chosen for installation in the Docker container.

Since Modelio is a GUI application which requires display support to run, a virtual display has to be created inside the Docker container so that Modelio can run normally. For this purpose, an application called Xvfb¹⁸ is used. Xvfb is an X server that can run on machines with no display hardware and no physical input devices. It emulates a dummy framebuffer using virtual memory. With Xvfb, a dummy display can be created so that Modelio can run inside a Docker container, even though there is no real physical display. For more details about the dockerization process, please see *Annex A: Dockerfile for Modelio*.

3.3.1.2 Dockerization of FREVO

FREVO is a Java application which requires JDK 8 and Apache Ant¹⁹ to be built and run. To build the Docker image, a JDK base image withant packages installed has been used. Furthermore, FREVO's source code as well as necessary test data are copied into the image. After that, FREVO is built and prepared using a set of commands. During the testing phase, a predefined script executes the test cases inside the Docker container. For more details about the dockerization process, please see *Annex B: Dockerfile for FREVO*.

3.3.1.3 Dockerization of Minisim

Minisim is a Java application developed with the help of Apache Maven²⁰ build management tool. Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information. The dockerization process of Minisim involves the following steps: a proper Maven Docker image is chosen as the base image. On top of that, Minisim's source code as well as necessary test data is copied into the image. After that, a Maven test command starts the build and test process. For more details about the dockerization process, please see *Annex C: Dockerfile for Minisim*.

¹⁸ <https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml>

¹⁹ <http://ant.apache.org/>

²⁰ <https://maven.apache.org/>

3.3.2 Component Testing

As stated in D3.7, the responsibility of formulating tests for each component belongs to the component developers. As a result, unit tests were provided for different components by their respective developers. These tests are integrated into the CI framework and run automatically to verify the correctness of each component. Passing the tests of all components is the pre-condition for a further integration test. The details of unit tests are documented in the following subsections:

3.3.2.1 CPSwarm Extension Component Tests

For the CPSwarm extension, component test cases written with the help of JUnit are provided as part of the source code. The unit tests are managed with Maven and executed automatically, when the source code is built by the CI tool.

3.3.2.2 FREVO Component Tests

For FREVO, LAKE has provided the test suites as well as test scripts to perform component tests. In the test suites, multiple optimization test cases are provided with pre-defined parameters for problem definition, algorithm representations, ranking algorithms, etc. By executing the test scripts, FREVO runs the optimization process against these test cases and generate hash files, which represent the outcome of these optimization processes. These hash files are compared to a set of pre-generated hash files. Because the test cases also specify the random seeds, the optimization results are deterministic despite the random evolution process involved. This means, if everything is implemented correctly, the newly generated hash files match the pre-generated ones. As a result, a test is passed only when the hash files generated by the tests match those pre-generated ones.

3.3.2.3 Minisim Component Tests

For Minisim, unit test cases written with the help of JUnit are provided as part of the source code. Similar to the CPSwarm extension, the Minisim unit tests are performed using Maven. The tests involve verifying the internal logic of the Minisim simulator as well as the API interface of the MQTT client wrapper. The component test is executed automatically, when the source code is built by the CI tool.

3.3.3 Integration Testing

The integration test cases describe how the test should be carried out in CPSwarm when components, dependencies, and test cases are integrated. The integration test cases focus on whether the flow of data and control from one component to the other takes place as intended. Hence, integration test cases demonstrate scenarios where one component is being called from another in order to test the overall application functionality to make sure the application works when the different components are brought together.

In order to perform integration tests, the components that are combined to form the CPSwarm workbench, namely Modelio, Frevo, Minisim, and associated scripts are integrated into a pipeline. The pipeline is a set of Bamboo CI plans which are executed by Bamboo whenever changes in code are detected. The test plans are described in the following section.

3.3.3.1 Test Plans Setup in Bamboo

This section describes the Bamboo CI plans that form the testing pipeline.

Modelling Tool - Modelio CPSwarm Extension Build and Test

The goal of this plan is to build the CPSwarm extension for Modelio from source. This plan consists of two stages and each stage consists of multiple tasks, which are executed in sequence. They are detailed below.

Stage 1: Build Modelio CPSwarm Extension from source (Figure 14)

The following tasks are executed within this stage:

1. The source code of CPSwarm extension is checked out from the Modelio Forge into the current stage.
2. A Docker container with Maven base is created. The CPSwarm extension is created and packaged in this container.

After the execution of the tasks, the packaged artefact is exported for later use by other plans.

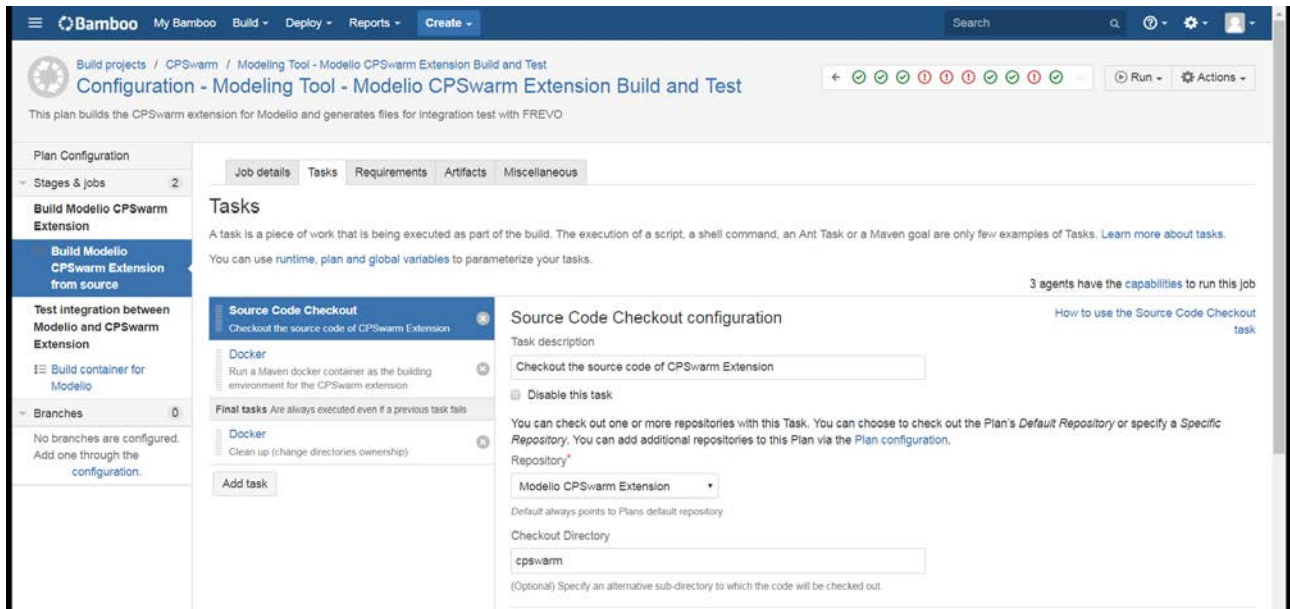


Figure 14 - CPSwarm Extension Build Plan (Stage 1)

Stage 2: Build Container for CPSwarm Extension testing within Modelio (Figure 15)

Before the execution of tasks, the aforementioned packaged artefact are imported into this stage. The following tasks are then executed:

1. Necessary files (a Modelio reference project, test scripts, etc.) for this stage is checked out from a GitLab repository into the current stage.
2. The imported CPSwarm extension artefact is copied to a proper place under the directory of the Modelio reference project.
3. A Docker image created as described in 3.3.1.2.
4. A Docker container is instantiated from the created image. In this container, test scripts are executed and files are generated for integration test with FREVO.

After the execution of all tasks, the generated files are exported for later use by other plans.

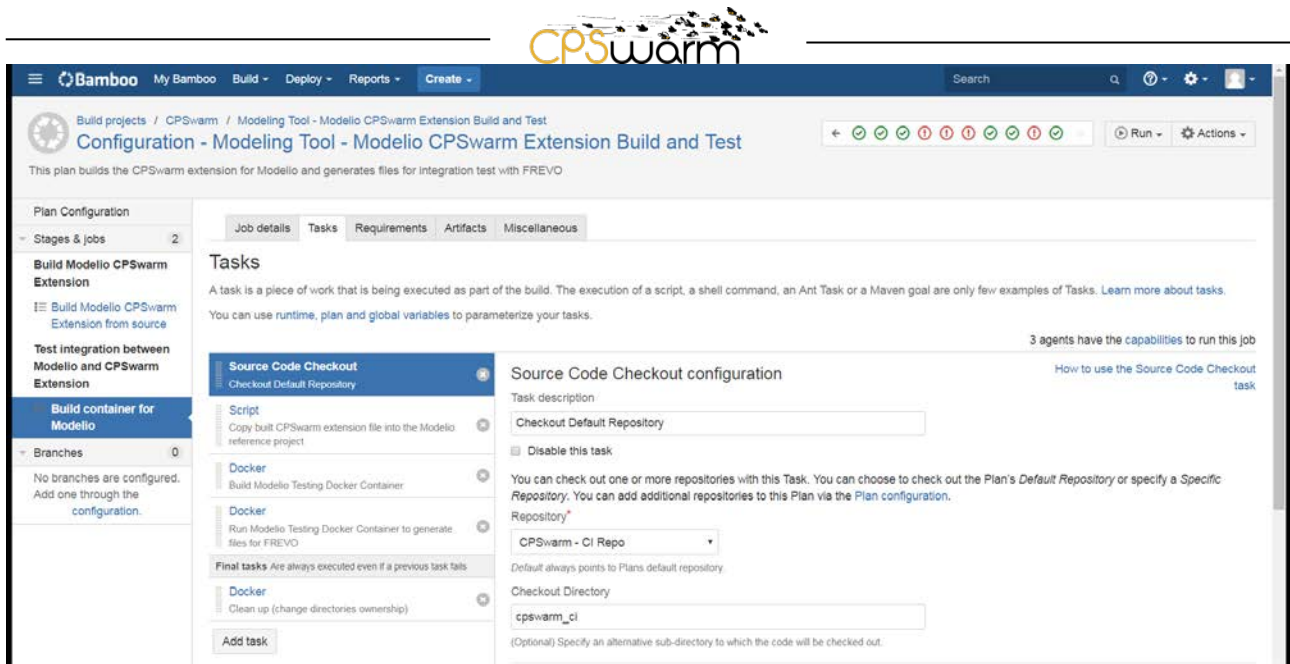


Figure 15 - CPSwarm Extension Build Plan (Stage 2)

Optimization Tool - FREVO Initialization

The goal of this plan is to generate hash files, which serve as reference for FREVO's component test later on. Due to the technical characteristic of this test, these files need to be generated on the same machine as the one on which the component test runs. As a result, this plan is configured to run only once in the CI setup phase, instead of running it every time new code is committed. This plan consists of one stage.

Stage 1: Initialization (Figure 16)

The following tasks are executed within this stage:

1. The reference source code of FREVO is checked out from its repository into this stage.
2. A Docker image is created with the reference FREVO source code included. During build time, the source code is compiled with Ant.
3. A Docker container is then instantiated from the newly created Docker image. Commands and scripts are executed to generate the hash files. Those hash files are the reference files for the subsequent FREVO component test.

After the execution of the tasks, the hash files are exported as artefacts for later use by other plans.

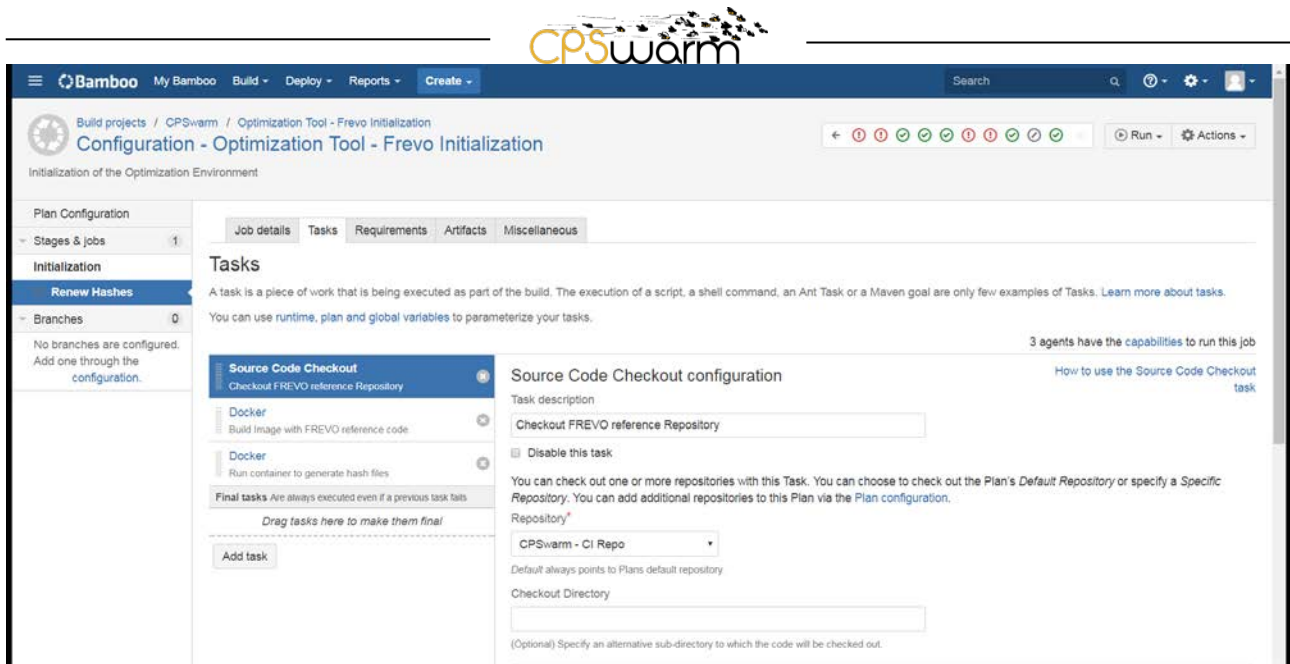


Figure 16 – FREVO Initialization Plan (Stage 1)

Simulation Environment - Minisim Build and Test

The goal of this plan is to build Minisim and run its component test. This plan consists of two stages. In each stage, multiple tasks are executed in sequence. They are explained below.

Stage 1: SimulationWrapper Test (Figure 17)

The following tasks are executed within this stage:

1. The source code of SimulationWrapper is first checked out from the GitLab repository into this stage.
2. After the checkout is completed, a Docker image is created as described in 3.3.1.3 with the downloaded source code included.
3. As the test involves interaction with the MQTT broker, a Docker container with Mosquitto MQTT broker installed is instantiated and run.
4. A Docker container is then instantiated from the newly created Docker image. Commands and scripts are executed within this container to build and run the component test of SimulationWrapper.

After executing the tasks, the built artefact from task 4 is exported for later use in the next stage.

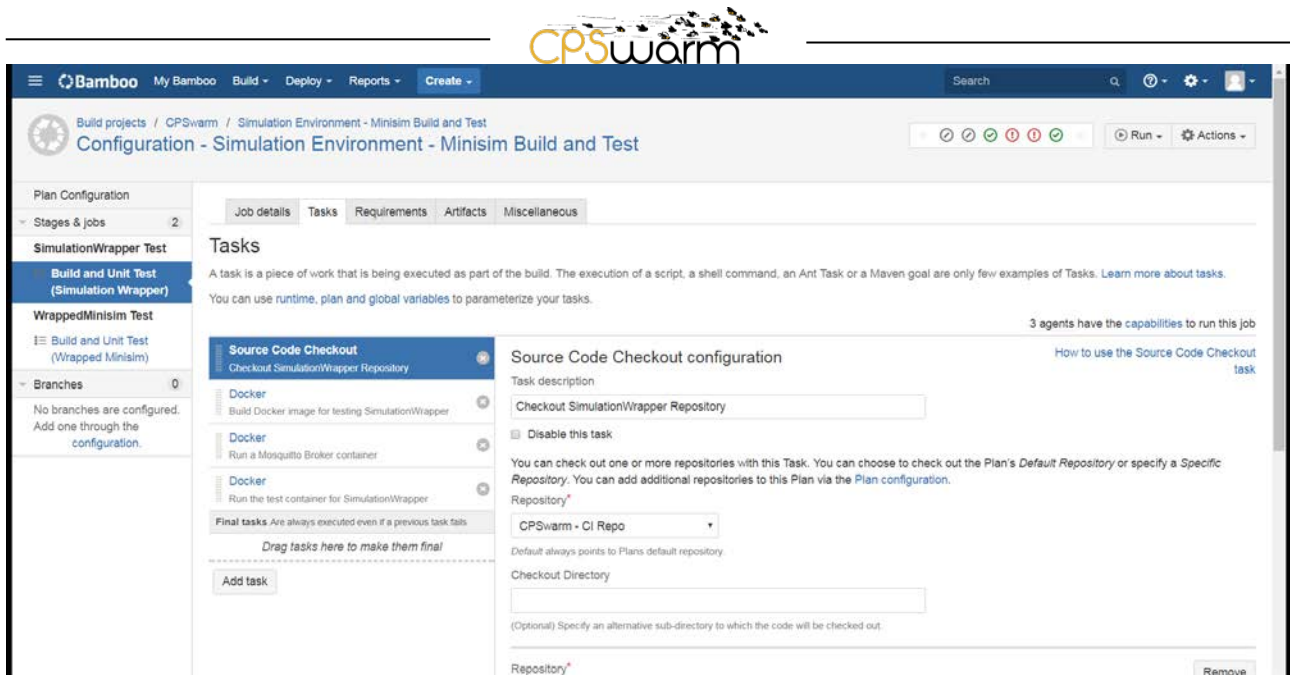


Figure 17 - Minisim Build and Test Plan (Stage 1)

Stage 2: WrappedMinisim Test (Figure 18)

Before executing the tasks, the artefact exported from the previous stage is imported into this stage. The following tasks are then executed:

1. The source code of WrappedMinisim is checked out from the GitLab repository into this stage.
2. After the checkout is completed, the imported artefact is put into the proper place under the source code directory. This artefact is needed for the build process.
3. A Docker image is created as described in 3.3.1.3 with the downloaded source code included.
4. As the test involves interaction with MQTT broker, a Docker container with Mosquitto MQTT broker²¹ is instantiated and run.
5. A Docker container is then instantiated from the newly created Docker image. Commands and scripts are executed within this container to build and run the component test of Minisim.

²¹ <https://mosquitto.org/>

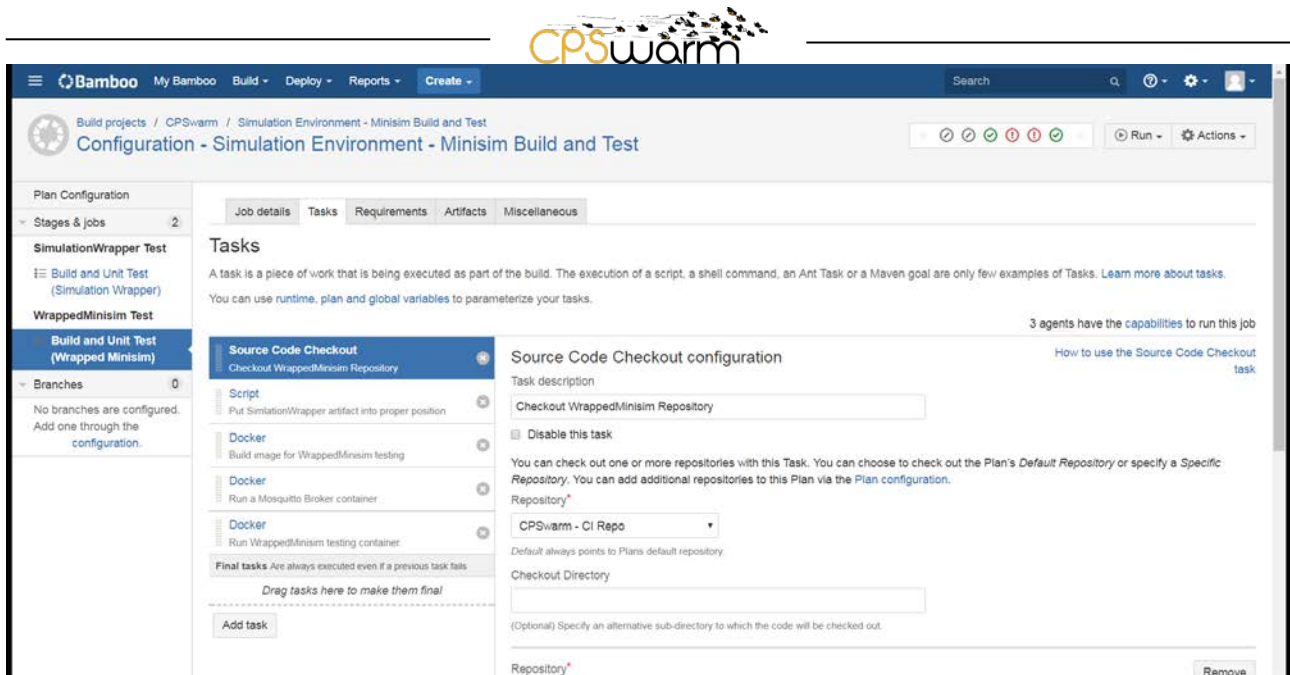


Figure 18 - Minisim Build and Test Plan (Stage 2)

Optimization Tool - FREVO Build and Test

The goal of this plan is to build FREVO and run its component test. However, since FREVO resides between Modelio and Minisim, this plan also serves as an integration test for all three components. In this plan, files generated by the "Modelling Tool - Modelio CPSwarm Extension Build and Test" are imported as part of FREVO's source code for compilation. This serves as an integration test between Modelio and FREVO. After successful compilation, a test process similar to the aforementioned "Optimization Tool - Frevo Initialization" plan is performed. However, instead of checking out the reference source code of FREVO, the newly committed FREVO source code for CPSwarm is used. This test process involves the participation of WrappedMinisim. As a result, it serves as the integration test between FREVO and Minisim. This plan consists of only one stage.

Stage 1: Build and Run Test Suites (Figure 19)

Within this plan, the following tasks are executed in sequence:

1. The source code of FREVO is first checked out from the GitLab repository into this plan.
2. The artefacts exported from plan "Optimization Tool - Frevo Initialization" are imported to the Bamboo agent.
3. The artefacts exported from plan "Modelling Tool - Modelio CPSwarm Extension Build and Test" are imported into this stage and put into the source code directory of Frevo.
4. After the download is completed, a Docker image is created as described in 3.3.1.2. During the image build process, FREVO is compiled into executable binaries.
5. A container with WrappedMinisim is run.
6. As the test involves interaction with the MQTT broker, a Docker container with Mosquitto MQTT broker is instantiated and run.
7. A Docker container is instantiated from the newly created Docker image. Commands and scripts are executed to run the test suite of FREVO, which test the integration between FREVO and Minisim.

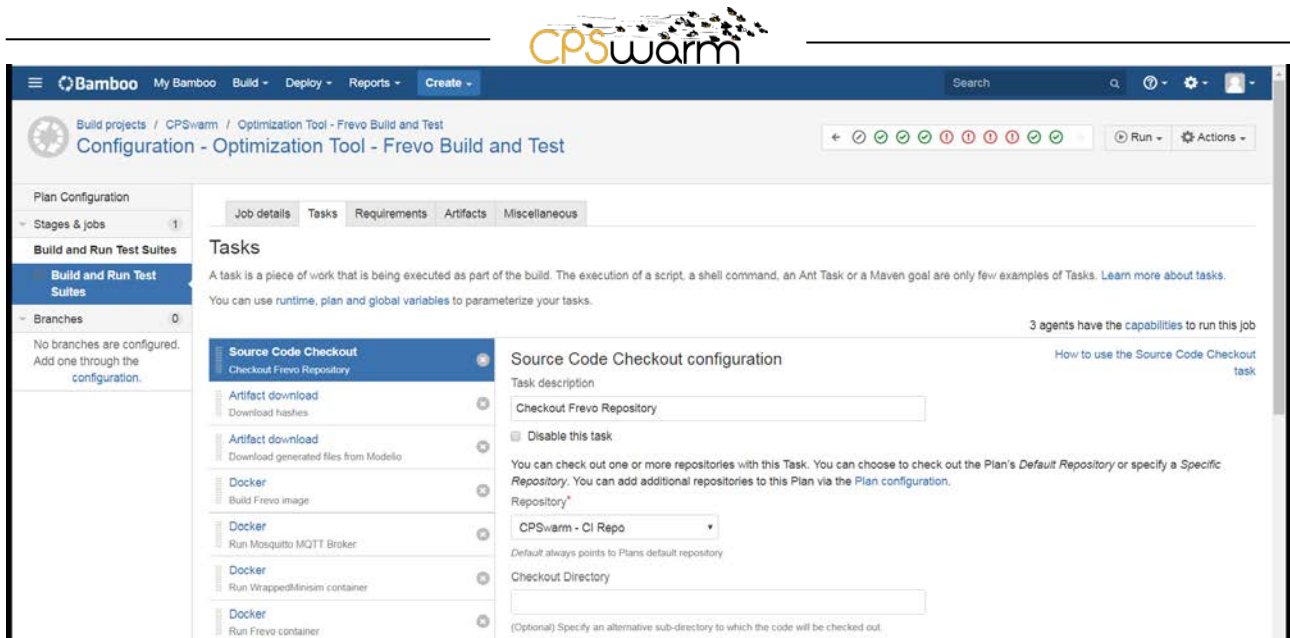


Figure 19 – FREVO build and test plan

3.3.3.2 Trigger Relationships between Test Plans

In Bamboo, the trigger condition for each plan can be configured separately. A plan can be triggered to run by detection of changes in a repository, or by the execution of another plan. The trigger relationship between the aforementioned plans is illustrated in Figure 20.

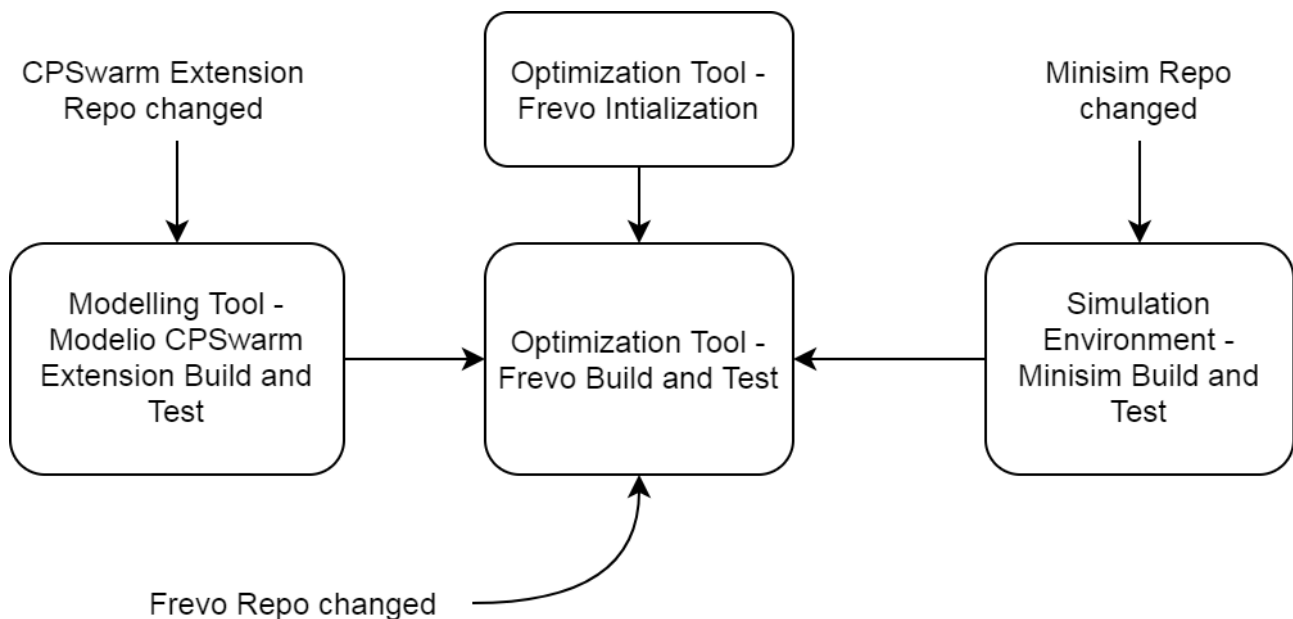


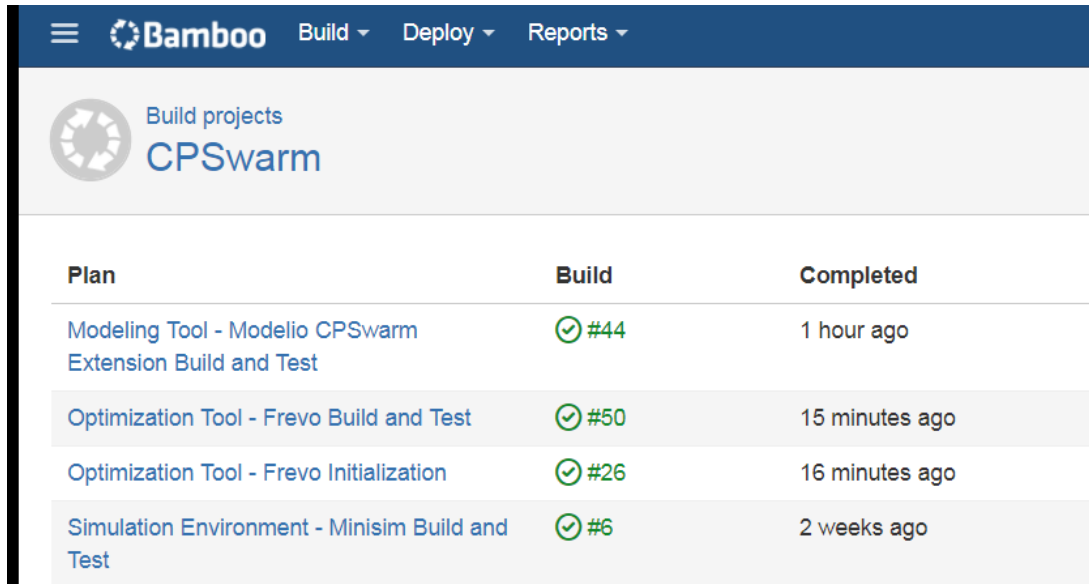
Figure 20 - Trigger relationship between plans

In this diagram, the arrow is pointing from a triggering element to the triggered element. It can be seen that the plan "Modelling Tool – Modelio CPSwarm Extension Build and Test" is triggered when it detects changes in the CPSwarm extension repository. Similarly, "Simulation Environment – Minisim Build and Test" and "Optimization Tool – Frevo Build and Test" plans are triggered by the changes in Minisim and Frevo repositories, respectively. This ensures that whenever changes happen in a repository, the correspondent element will be rebuilt and tested to verify correctness of that single component. The execution of plan "Modelling Tool – Modelio CPSwarm Extension Build and Test" and plan "Simulation Environment – Minisim

Build and Tests" both trigger the plan "Optimization Tool – Frevo Build and Test". This is to ensure that once one of these components is rebuilt, the integration test is run to verify the integration of all components.

3.3.4 Integration Test Results

As demonstrated in Figure 21, all integration tests have passed. As a result, the conclusion can be drawn that all CPSwarm components are integrated successfully for the first phase of development within the project.



Build projects CPSwarm		
Plan	Build	Completed
Modeling Tool - Modelio CPSwarm Extension Build and Test	✓ #44	1 hour ago
Optimization Tool - Frevo Build and Test	✓ #50	15 minutes ago
Optimization Tool - Frevo Initialization	✓ #26	16 minutes ago
Simulation Environment - Minisim Build and Test	✓ #6	2 weeks ago

Figure 21 - Integration Test Results

4 Conclusions

This deliverable is a report documenting the implementing of the Initial CPSwarm Workbench and associated tools as well as the integration result of the first development phase in project CPSwarm. In this document, the available initial components and their interfaces were reviewed. After that, technical details regarding the setup of integration tests were presented. In 3.2, the deployment details of the CI tool Atlassian Bamboo were introduced. In 3.3, the dockerization of components, the setup of integration test cases, and the integration results were demonstrated. At this point, the integration tests pass, which indicates that integration of the first phase components is successful.

In the upcoming months, development of other CPSwarm components will start and the initial components will also be revised and modified according to newly identified requirements. The next integration phase will end in M22 of CPSwarm project. By the end of M22, the deliverable 3.5, which is an updated version of this deliverable, will be submitted. D3.5 will further document more comprehensive integration results of the CPSwarm components.

Acronyms

Acronym	Explanation
CI	Continuous Integration
BPMN	Business Process Model and Notation
MARTE	Modeling and Analysis of Real-time and Embedded systems
FREVO	Framework for evolutionary design
API	Application Programming Interface
VM	Virtual Machine

List of figures

Figure 1 - Relationship of this document with other deliverables and Tasks	4
Figure 2 - CPSwarm system architecture extracted from D3.1 (Components to be integrated are highlighted in red rectangles. The annotations show the mapping between components in the diagram to the components defined in deliverables)	5
Figure 3 - Modelio graphical user interface	6
Figure 4 - Subcomponent diagram of the Modelling Tool.....	7
Figure 5 Model to represent a swarm member with BEECLUST	8
Figure 6 Caption missing.....	9
Figure 7 - FREVO graphical user interface	10
Figure 8 - Stage simulator.....	11
Figure 9 - Gazebo simulator.....	11
Figure 10 Integration Test Set Up.....	12
Figure 11 - Interfaces between the CPSwarm initial components.....	13
Figure 12. CPSwarm Initial Continuous Integration (CI) Platform. Two independent networks (yellow) consisting of virtual machines (blue), hosting containerized applications.	15
Figure 13. Multi-stage Bamboo plan [6]	17
Figure 14 - CPSwarm Extension Build Plan (Stage 1).....	20
Figure 15 - CPSwarm Extension Build Plan (Stage 2).....	21
Figure 16 - FREVO Initialization Plan (Stage 1)	22
Figure 17 - Minisim Build and Test Plan (Stage 1)	23
Figure 18 - Minisim Build and Test Plan (Stage 2)	24
Figure 19 - FREVO build and test plan.....	25
Figure 20 - Trigger relationship between plans.....	25
Figure 21 - Integration Test Results.....	26

References

- [1] X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms*, Luniver Press, 2008.
- [2] H. H. Thomas Schmickl, "Bio-inspired Computing and Networking," CRC Press, 2011, pp. 95-137.
- [3] X.-S. Yang, "Firefly Algorithms for Multimodal Optimization," *Stochastic Algorithms: Foundations and Applications*, pp. 169-178, 2009.
- [4] A. Sobe, I. Fehévari and W. Elmenreich, "FREVO: A tool for evolving and evaluating self-organizing systems," in *Proceedings of the 1st International Workshop on Evaluation for Self-Adaptive and Self-Organizing Systems*, Lyon, 2012.
- [5] Atlassian, "Atlassian Bamboo Open Source Project License Request," 11 2017. [Online]. Available: <https://www.atlassian.com/software/views/open-source-license-request>.
- [6] Atlassian, "Bamboo Best Practice - Using Stages," 11 2017. [Online]. Available: <https://confluence.atlassian.com/bamboo/bamboo-best-practice-using-stages-388401113.html>.

Annex A: Dockerfile for Modelio

```
FROM ubuntu:16.04
```

```
MAINTAINER Junhong LIANG <junhong.liang@fit.fraunhofer.de>
```

```
ARG MODELIO_PKG_NAME=modelio
```

```
## Install Modelio as Debian package
```

```
## Beside, xvfb is installed to create a virtual display for Modelio
```

```
RUN apt-get update && \
```

```
    apt-get install -y wget && \
```

```
    mkdir /modelio && \
```

```
    wget -O /modelio/${MODELIO_PKG_NAME}.deb https://www.modelio.org/download/send/24-modelio-361/91-modelio-361-debian-ubuntu-64-bit.html && \
```

```
    apt install -y /modelio/${MODELIO_PKG_NAME}.deb && \
```

```
    apt-get install -y xvfb
```

```
## To run test, you also need to mount the modelio project and the test scripts to the container.
```

```
## As example, let's assume we have a directory /Docker/modelio on host, which contains a project folder called EmergencyExitRos
```

```
## as well as a test script launchGeneration.py. In this case, you can run your container like this:
```

```
##
```

```
##     Docker run --name modelio -v /Docker/modelio:/modelio modelio_image
```

```
##
```

```
## then the following command would create a virtual display with Xvfb and then generate files according to the project setting
```

```
CMD Xvfb :1 -screen 0 1024x768x16 & DISPLAY=:1.0 modelio-open-source3.6 -consoleLog -workspace /modelio -project EmergencyExitRos -batch /modelio/launchGeneration.py
```

Annex B: Dockerfile for FREVO

```
FROM openjdk:8-jdk-slim

RUN apt update && \
    apt install -y ant

RUN addgroup --gid 1000 frevo && \
    adduser --uid 1000 --ingroup frevo --system frevo

COPY repo/frevo /home/frevo
RUN chown -R frevo:frevo /home/frevo
RUN rm /home/frevo/testsuite/hashes/*
WORKDIR /home/frevo

USER frevo
RUN ant build
RUN java -jar createscripts.jar

# mount to export hash files
VOLUME /home/frevo/testsuite/hashes

WORKDIR testsuite

# Remove piping of logs
RUN sed -i 's/>\dev\>null 2>&1//g' testsuite.sh
ENTRYPOINT ["/testsuite.sh"]

## Running locally:
# git clone https://git.repository-pert.ismb.it/ftavakolizadeh/ContinuousIntegration.git
# git clone https://git.repository-pert.ismb.it/mrappaport/cpswarm.git repo
# docker build -t frevo-test .
# docker run -v $(pwd)/hashes:/home/frevo/testsuite/hashes --rm frevo-test -renew
# docker run -v $(pwd)/hashes:/home/frevo/testsuite/hashes --rm frevo-test
```

Annex C: Dockerfile for Minisim

```
FROM maven:3-jdk-8-alpine
```

```
WORKDIR /home
```

```
VOLUME /home
```

```
ENTRYPOINT ["mvn", "test"]
```

```
## Running locally:
```

```
## SimulationWrapper
```

```
# git clone https://git.repository-pert.ismb.it/CPSwarm/SimulationWrapper.git
```

```
# docker build -t simulation-wrapper .
```

```
# docker run -v $(pwd)/SimulationWrapper:/home simulation-wrapper -
```

```
Dtest_broker=tcp://example.com:1883
```

```
## WrappedMinisim
```

```
# git clone https://git.repository-pert.ismb.it/CPSwarm/WrappedMinisim.git
```

```
# docker build -t wrapped-minisim .
```

```
# docker run -v $(pwd)/WrappedMinisim:/home wrapped-minisim -
```

```
Dtest_broker=tcp://example.com:1883
```