IMOLA software specifications and development process documentation

***I**ntegration of **M**aintenance **O**perations and **L**ogistics **A**ctivities*

Version	Date	Author	Comments
1.0	Augustus 29, 2018	Ben Vermeulen	Initial version with user requirements, scope, early version of architecture layer diagram
1.1	September 24, 2018	Ben Vermeulen	Revised architecture, extended description of use case flows, early version of technical challenges and data structure
1.2	October 9, 2018	Ben Vermeulen	Updates layer diagrams, extended class diagram based on tests with software implementation

Contents

1 Introduction ... 4
2 Software Development Plan ... 5
2.1 Scope ... 5
2.2 Process approach & deliverables ... 5
2.3 Development environment ... 6
2.4 Software Test Plan ... 6
3 Requirements specification ... 8
3.1 Software functionality ... 8
3.2 Usage/ usage context ... 8
3.2.1 "Insight in sensibility of tactical asset rescheduling" ... 8
3.2.2 "Optimization of tactical asset rescheduling" ... 8
3.2.3 "Determine usage pattern of (redundant) components" ... 9
3.2.4 When to use either one of the two functions of the scheduler? ... 9
3.2.5 Enhanced component degradation simulator ... 9
3.3 Interface specification ... 10
4 Design specification ... 11
4.1 Architecture ... 11
4.2 Deployment viewpoint/ user environment ... 13
4.3 Main use cases and flow-of-events ... 14
4.3.1 Maintenance of the asset and component repositories ... 14
4.3.2 Maintenance of the scheduling project ... 15
Creating, opening a project ... 15
Edit the asset pool of the project ... 15
4.3.3 Conduct tactical (redundant) component usage scheduling ... 16
4.3.4 Conduct tactical asset pool scheduling ... 16
4.3.5 Import external data ... 17
4.4 Design technical changes ... 17
4.4.1 Separation of concerns into two modules in the MaSeLMa API ... 17
4.4.2 Revise three-application solution for resource planning? ... 17
4.4.3 Operating mode issues: user-own operating mode labels, consider tasks, add flags ... 18
4.4.4 Integrate and enhance tactical planning ... 18
4.4.5 Obtaining external data ... 18
4.4.6 Extension & integration of ProSeLo API ... 19
4.5 Data structure design ... 19

4.5.1 Human actions for data in-/ output 19
4.5.2 Data lake 20
4.5.3 Scheduling project 20
4.5.4 Pass-through 20
4.6 User Interface Design 20
4.7 Class Diagrams 21
4.8 Use Case Realizations: concurrency 21

1 Introduction

This document lays down the software development process of the IMOLA maintenance and resource planning application building upon the MaSeLMa application and integrating the ProSeLo API. Where the previous MaSeLMa and ProSeLo schedulers offer decision support for the ***operational*** maintenance schedule only, the IMOLA application offers decision support at the ***tactical*** level by using previously written APIs for operational planning under the hood. The planning is *tactical* in the sense that the user specifies tactical goals on the maintenance schedule and the application now finds an operating schedule/ asset usage that minimizes the lower cost and/ or higher availability subject to reaching those goals. Typical examples of tactical planning goals are, e.g. throttling usage of a diesel engine to have its major overhaul maintenance coincide with a mandatory vessel inspection or assigning printing jobs to have preventive replacement of laser heads coincide with a readily planned visit of a maintenance engineer to the factory.

In particular, the integrated application is designed to focus on two different types of tactical planning: (1) asset pool scheduling and (2) component usage planning, all subject to given maintenance constraints. The next chapter will provide extensive details on what we mean with both terms.

From the outset, we seek to develop a planning application that can be used in different sectors, but above all the maritime sector and the mid- to high-tech sector. There are substantial commonalities from the perspective of tactical planning; in all of these sectors there are arrays/ pools of assets and tasks/ usage may be scheduled to the various assets so as to tune the moments of maintenance. That said, though, there are also major differences with regard to ownership of the (arrays of) capital goods and (legal) responsibility for maintenance of these goods. For instance, in several high-tech sectors, asset owners have only one machine such that ‘usage throttling’ is not an option (other than changing the order of particular jobs, which may not very often possible). It may also be that service contracts are signed and that planning of usage and maintenance are not attuned, or maintenance is subject to usage constraints (which is merely operational planning). As such, we design the application for usage, first and foremost, in the maritime sector, particularly because of the interest of several parties, but also because it is the more generic case (with more than 1 operating mode, spare part delivery limitations, variable maintenance costs, etc.). However, we also target providing tactical planning for immobile capital goods as well, albeit at lower priority.

With regard to integration of resource planning, we have to recognize that the IMOLA project is too short to be able to incorporate all relevant resources (e.g. maintenance staff, tools, spare parts). Moreover, the complexity of incorporating all these resources in scheduling is a formidable task requiring a thorough scientific study first. That said, the IMOLA development team seeks to further integrate the MaSeLMa demonstrator setup in which the MaSeLMA GUI used SPMS.

The IMOLA project has a mere 5 months running time and limited budget mostly spent on developing additional software functionality which further increases the level of technology and business readiness with implementation at several pilot companies in mind.

2 Software Development Plan

2.1 Scope

In the IMOLA project, an integrated maintenance & resource planning application is provided, notably providing *tactical* decision support. There are now two different types of tactical planning requested by the pilot companies:

1. Tactical asset pool scheduling (APS) for ***two (or more) assets in the asset pool***. One of the pilot companies wants to know whether and, if so, when and of which assets to swap the operating schedules to advance/ postpone a mandatory preventive maintenance to a particular interval. An alternative perspective on this is that a list of tasks is to be assigned to an array of assets, and the scheduler assigns operating tasks to assets to advance/ postpone maintenance of the assets.
2. Tactical component usage scheduling (CUS) for two (or more) ***(redundant) components of a single asset***. The scheduler 'throttles' usage of particular components during particular operating intervals. Two of the pilot companies want to do so to advance/ postpone a mandatory preventive maintenance to a particular interval.

The IMOLA application is to provide a tactical maintenance & resource planning that is –within its constraints- *operationally* both feasible and near-optimal. To this end, the operational scheduler is used for each considered & compared tactical scenario, such that the ***operational*** solutions are (near-)optimal; this prevents comparing a poor operational solution in one scenarios with a top operational solution in another. In addition to that, the operational scheduler may still use the Gordian's Spare Part Maintenance Studio (SPMS) to integrate spare part costs and constraints.

One major, possibly very valuable extension currently considered out of scope of IMOLA is that allowing component usage scheduling in lower iterations of the tactical asset pool scheduling would give much more freedom to the asset pool scheduler.

The existing MaSeLMa GUI is extended to provide the APS (asset pool scheduling) and CUS (component usage scheduling) functionality. However, in deliberation with a panel of representatives of the pilot companies, we may decide to alter the MaSeLMA GUI to tuck away rarely used rudimentary functionality (e.g. operational scheduler functionality). In addition to that, it may be decided to make the GUI more user friendly (e.g. dragging tasks on a timeline rather than meticulously defining intervals), and providing a simple "dashboard" with tactical implications for further managerial decision making. The "HOWs and WHATs" are to be decided during the project.

Please note that integrating the ProSeLo API as planning module requires several additional developments in the GUI (e.g. conversion of input and output, visualization of output, etc.), in the project structure of the MaSeLMa API (retaining initialization settings and solutions), and of course functionality in the ProSeLo API.

2.2 Process approach & deliverables

Given the close range to the user and the aim to develop a pilot deliverable / demonstrator, we follow an incremental, iterative development process akin to scrum. Hereby, there are not only frequent meetings with the development team but also frequent test & feedback sessions with the pilot customers to allow for the necessary scrum volatility.

It is proposed to do bi- or triweekly meetings with the three developers in which the past sprint is reviewed, new functionality tested (see the section on the Software Test Plan), documentation

consolidated, and new sprints (re)defined. For each new sprint, sufficient time is to be allocated to discuss program-technical challenges, design and functionality scoping issues, process obstacles, user requirements, and, ultimately, design & functionality decisions. For this, the project manager functions as scrum master.

The project deliverables are specified in Figure 1 (for a detailed description, see the Project Proposal document). Deliverable 3, 4, 5, 6, and 7 require software development. The Gantt chart also shows during which months these software development activities take place.

	Deliverable	2018-08	2018-09	2018-10	2018-11	2018-12	2019-01
1	Functional specifications						
2	Consortium building & knowledge dissemination						
3	Basic demonstrator						
4	Pilots						
5	Implement basic version at Loodswezen						
6	Advanced demonstrator						
7	Implement at Loodswezen						
8	Knowledge dissemination						
9	“Follow up”-plan						

Figure 1. Gantt chart of project activities

Several of the deliverables are discuss in this document, see the table below.

Deliverable	Note
Functional specifications	This document
Consortium building & knowledge dissemination	-
Basic demonstrator	Discussed in this document
Pilots at first line companies	Mentioned in this document
Implement basic version at Loodswezen	-
Advanced demonstrator	Discussed in this document
Implement at Loodswezen	-
Knowledge dissemination	-
“Follow up”-plan	

2.3 Development environment

Both Felipe and Bas develop stand-alone modules and the integration/ communication is to be done in the application. Given the different ‘modules’ in the architecture and the distributed responsibilities, each of the developers has its own ‘stack’. See the table below.

	Ben	Felipe	Bas
IDE	Visual Studio Express		Visual Studio Express
Language	C#	Python	C#
Source control	Yet to be decided. Given the size of the project we may do without		

2.4 Software Test Plan

The IMOLA team follows a quasi-scrum approach with the developers. Given that each developer has its own specialization and works on separate modules and given the absence of a Quality Assurance department, developers have to evaluate/ test functionality of other developers. During the bi- or triweekly developer meetings, it is proposed to take one or two hours to discuss past developments, reflect on API or GUI implementations, test functionality (against preliminary documentation), and discuss (re)designs. Clearly, the limited time thus available for peer testing, quality assurance should

also rely on unit test functionality. As such, it is proposed that we –in parallel- maintain a test application/ console which can be run unattended and logs the status of the various modules and functionality.

So, ultimately, part of the development process, each of the developers is asked to maintain a minimal technical documentation, a test environment for the API including a minimal test plan (what it should and what it should *not* do), and a section of the test plan specifically for the GUI (written together with the principal GUI developer) once the GUI exposes the functionality.

3 Requirements specification

3.1 Software functionality

Given that the MaSeLMa GUI, the APIs of the MaSeLMa and ProSeLo projects, as well as the MaSeLMaBridge Interface providing access to SPMS are built upon, the reader is referred to the extensive documentation for those projects. There is explicit demand for a graphical interface showing the degradation as a function of usage, the predicted failure/ JIT preventive maintenance moments as well as the scheduled maintenance moment timeline.

The starting point of the extensions of the software functionality is the set of use cases of the three pilot companies in the maritime sector. We seek to generalize the functionality to allow users in that particular sector but also in other sectors to also use the application. After discussing the tactical planning with the pilot companies, we discerned two cases:

1. Decide whether to swap two assets of location to postpone/ advance maintenance to a particular time interval of interest. In general terms: attune the tasks (where tasks are tied to a particular location) in operating schedules. As such, the scheduler has to be able to determine to which asset to assign particular tasks.
2. Attune when to switch on and off particular redundant components of one single asset to postpone/ advance maintenance to a particular time interval of interest. The scheduler has to be able to swap which components are on/ off. Clearly, some of the components in the pool of redundant components have to be 'on'.

In both cases, there is a cross-check for spare part availability, delivery expenses taken into account, and a potential list of spare part related orders is generated.

The Scope section (2.1) provides our generalization of the software functionality.

3.2 Usage/ usage context

After extensive talks with the pilot customers, we discern several requirements, each of which has quite a different usage and usage context.

3.2.1 "Insight in sensibility of tactical asset rescheduling"

For one of the pilot customers, the primary goal is to get ***insight*** as to whether particular tactical asset planning scenarios are sensible at all. If so, then that particular customer seeks to further optimize the planning. However, for this particular customer, that one particular decision is a relatively infrequent one to make, for instance when new assets are added to the pool or a new scheduling problem with an alternative set of assets is considered. For example, the customer decides to start planning a (new) set of (new) vessels in a (previously not considered) context.

3.2.2 "Optimization of tactical asset rescheduling"

The idea is that rescheduling the usage of assets does affect the degradation rates and hence when maintenance is required. As such, rescheduling allows the customer to tune the maintenance moment to, for instance, align with low cost intervals in time (e.g. a vessel has to be docked for mandatory survey; a machine has to undergo major overhaul) or ensure availability of one or more assets during particular intervals in time (e.g. a vessel is used for a mission; a machine is scheduled to perform a printing job with a tight deadline).

The customer wants to be able to specify such 'intervals' and then have the application determine whether any asset rescheduling tactic is available and, if so, provide the customer with a viable or even the optimal asset rescheduling plan.

The customer will run this analysis when (1) new condition information becomes available on any of the components of the assets considered for rescheduling and particularly when this changes the estimated degradation rate, (2) any of the low-cost or availability intervals changes, (3) the asset pool or assets considered changes.

3.2.3 "Determine usage pattern of (redundant) components"

For two of the pilot customers, the primary goal is to control the usage of (some of the) redundant components such as to tune when these components are to be maintained. There are several scenarios conceivable: maintenance on the set (or a subset) of redundant components has to be:

1. 'sufficiently far apart' (or rather 'sufficiently close together', which can be achieved by using some components considerably more than others at least during a particular interval or use them in equal fashion,
2. advanced/ postponed to fall within a designated (low-cost) interval or rather to fall outside of a designated 'availability-required' interval.

3.2.4 When to use either one of the two functions of the scheduler?

Apart from the reasons mentioned above (targeted interval changes, trend breaking condition data arrives, asset pool changes, new problem considered), circumstances may call for running the scheduler again, e.g. when there is an unplanned opportunity for maintenance, for instance because maintenance on another component not part of the project is required, or when there is an unplanned failure which may require maintenance (but perhaps postponing is viable?). Table 1 provides an overview of when the customer is to rerun the tactical scheduler.

Table 1. In which occasions is the customer required to rerun the tactical scheduler?

	Target interval for maintenance changes	Trend breaking condition data	Unplanned maintenance opportunity	Sudden failure which may require maintenance	Asset pool changes	New asset scheduling problem considered
"Insight in sensibility of tactical asset rescheduling"	X				X	X
"Optimization of tactical asset rescheduling"	X	X	X	X	X	?
"Determine usage pattern of (redundant) components"	X	X	X	X		

3.2.5 Enhanced component degradation simulator

In addition to scheduling functionality, the requirements of the customer require enhancement of the component degradation simulation. In the maritime sector applications, for instance, the engine is treated as a single component and different criteria are used to determine when to conduct

maintenance: for some engines this is the cumulative number of liters of fuel used, while for others it is the total running hours (regardless of power or RPM).

In addition to the dimension of the condition variable, the simulation requires additional factors such as location (as a multiplier for the number of running hours) and season (as a multiplier on the fuel consumption).

Additional exploratory discussions with the pilot customers are required to provide details on the simulator.

3.3 Interface specification

At the outset it has been decided by the development team + project manager as well as per request of the pilot companies that the existing MaSeLMa GUI would be revamped. There are relatively few concrete demands from the pilot customers with regard to extensions of/ changes to the GUI. That said, for the pilot customers, particularly the visualization of the degradation curves of components as well as the maintenance activities on a timeline are critical. During the first stage of the development process, the focus is on evaluation of/ insight in scenarios and whether tactical planning is sensible. In the second stage of the development process, it is requested to (i) implement actual condition observations and usage data and (ii) concretely implement tactical optimization.

As described in the next chapter, given that the scheduler needs to be able to handle a pool of assets (rather than just one as before), it is opted to go for a repository for assets and a repository of components (with observation dataset associated) for quick system building across assets as well as central collection of data for enhance degradation/ usage estimation.

4 Design specification

4.1 Architecture

At the highest level, the architecture of the IMOLA solution is straight forward, see Figure 2. At the core of the architecture are the tactical usage & deployment scheduler and the operational maintenance planning engines of two APIs. The MaSeLMa API determines maintenance schedules for usage regimes with multiple operating modes and their associated degradation distributions, maintenance cost and maintenance options (as e.g. is the case in the maritime sector). The more specific ProSeLo API determines maintenance schedules for usage regimes with a single operating mode and single usage rate and maintenance costs (as e.g. in manufacturing). At the technical level, both have different approaches, each with advantages. However, an abstraction layer is to uniform the interface access, which -at the operational level- basically requires also providing a conversion & communication wrapper around the ProSeLo API. The IMOLA application thus allows passing input as well as presenting & visualization of the ProSeLo output.

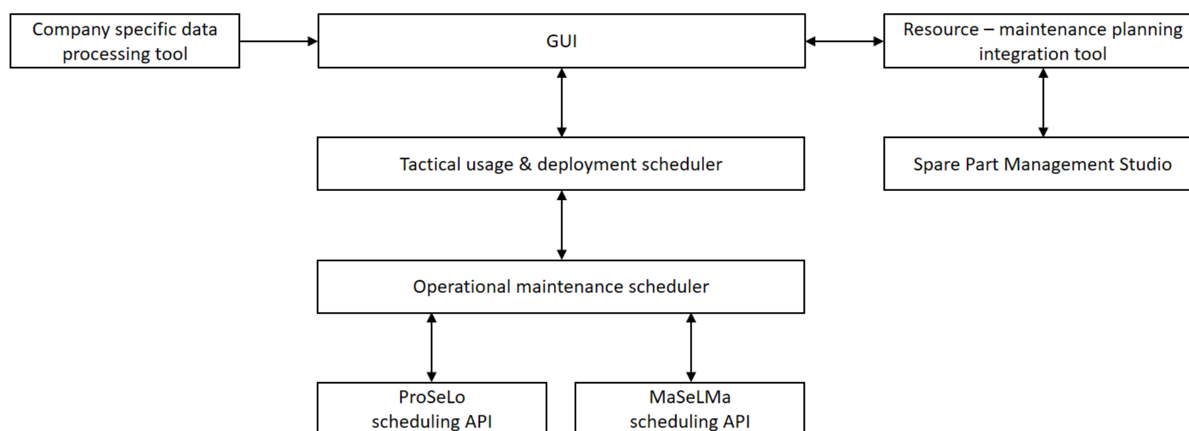


Figure 2. Overview of modules in the novel architecture, most importantly now also featuring the ProSeLo engine

Company-specific data is extracted/ processed externally, possibly with a separate processing tool or using company-specific ERP modules, and imported into the scheduling project in the GUI. Although technically feasible, there probably is too little time to integrate data extraction into the GUI such that the need of external tools is not required. Note that one of the pilot companies indicated that this would be highly preferable in the mid-long term so as not to mothball the application for being not user-friendly.

With regard to the spare part planning, we stick with the structure of the MaSeLMaBridge solution, which means that spare part scheduling is a multistep, iterative process of exporting a provisional maintenance schedule, importing that into & running the integration tool to determine spare part deliveries and associated cost vectors, exporting that, then importing into and running the maintenance scheduler again. This is rather tedious and particularly now that we are going to aggregate maintenance scheduling to an asset pool (e.g. fleet level), we should consider handling inventory optimization and automated ways to communicate with the spare part management studio in future updates.

At a lower level, the architecture becomes a bit more involved, particularly due to a (functional) separation of data (handling) and scheduling project (handling). This is a novel concept introduced in the tactical planner to ensure that the same types of components used in different assets can benefit

from data collected on any of them by re-estimating parameters. In addition to that, the MaSeLMa application required adding the cost specs, condition data, condition degradation specs, etc. to be added in each project (so every time one creates a new project). Now, this all stored in a database and can be added by point-and-click in any new project. Related to that, it may very well be that scheduling projects may contain different sets of assets, and simply adding them in a point-and-click (or drag-and-drop) style is much more user friendly. As such, we propose to programmatically separate scheduling functionality for a *single* project and data-handling functionality tuned to handle data *across multiple* projects. Consequently, as soon as new data is added, such as e.g. condition data, and new parameter values are determined, all scheduling projects using that data can be further enhanced. Given that the naming of and the specifications of the functionalities are the same as those in the MaSeLMa and MaSeLMaBridge projects, we simply provide a layer diagram of the functional components and modules in Figure 3 without extensive description.

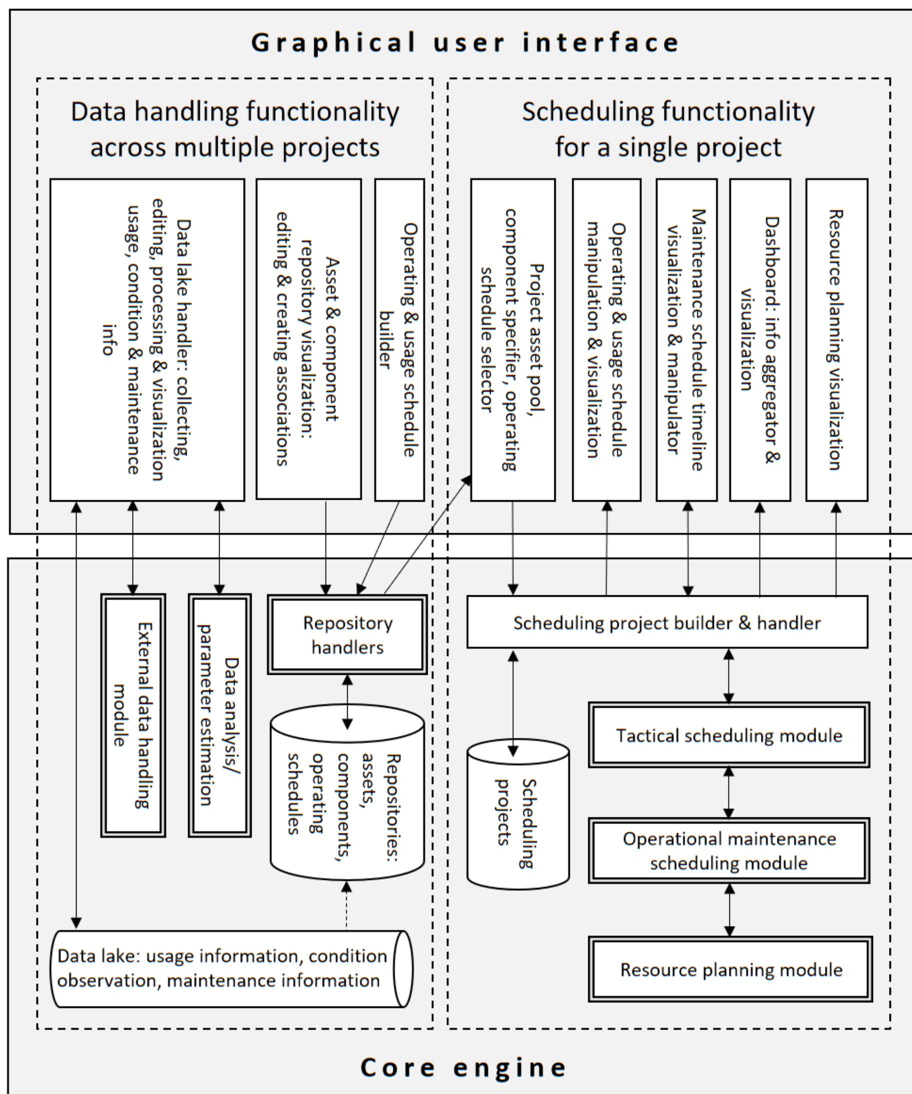


Figure 3. Layer diagram of top-level functionality and modules (and data sources) of the tactical scheduler application

Each of the more involved *modules* (i.e. the functionality boxes with a two-line compound type) requires further explanation. As the (ideas behind the) resource planning and data analysis/ parameter estimation functionality has not changed substantially since MaSeLMa(Bridge), we omit a visual representation and further description here.

The scheduling module is rather straight forward an input/output data conversion and asynchronicity handler to call the tactical planner functionalities for component usage or asset deployment scheduling, which, in turn, (iteratively) call the operational scheduler, see Figure 4. An alternative design would be to treat the penalty-based Genetic Programming scheduler as just an alternative scheduling method just as the ADP, L10, JIT and the many flavors of cost-based or rule-based rescheduling heuristics (such as implemented in older versions of the MaSeLMA application). The design would then become as what is contained in Figure 5.

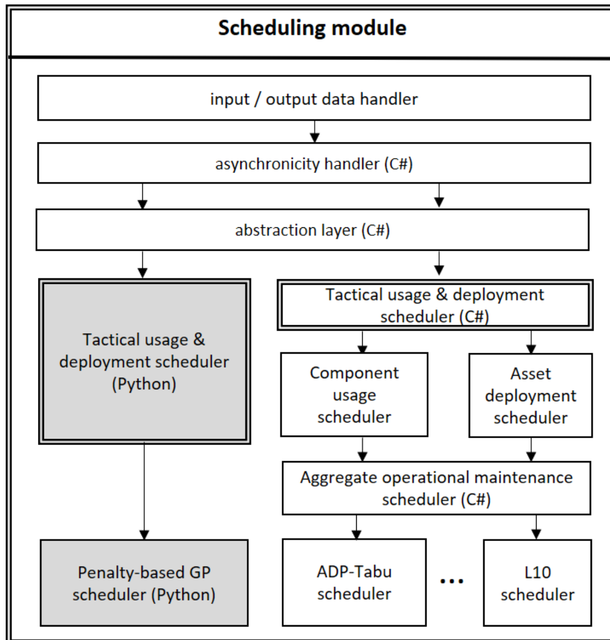


Figure 4. Layer diagram of the scheduling module including tactical usage & deployment and operational maintenance scheduling with tactical planning integrated into the ProSeLo API

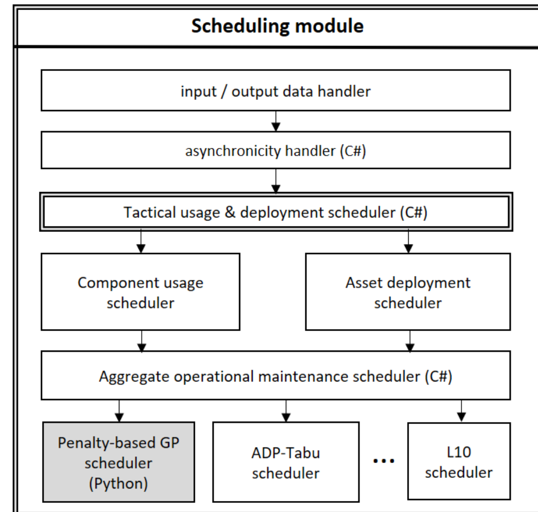


Figure 5. Alternative design for the scheduling module in which the aggregation across Python and C# functionality occurs at the operational rather than tactical level. In this case, the ProSeLo API would not integrate tactical planning.

At the moment of writing, there is an exploratory study of changes to the penalty-based scheduler for tactical planning. This study should help the development team + project manager to decide between the two designs.

4.2 Deployment viewpoint/ user environment

At this point in time, we only have a few pilot customers to consider, but the environment seems to be quite generic. One exception may be DMI/ RNLN which may prohibit access to data servers from our application.

In the simplest form, the user adds all input in the GUI manually and, depending on whether it is project specific, the data is stored in a scheduling project in the project directory specified by the user or the data is stored in a (local) data lake. More advanced would be to collect, import and convert the current data upon opening the scheduling application in an automatic and unattended way. At present this is considered out of scope of the IMOLA project. There may be unique workflows depending on the company specific ERP system and data management software packages. The startup to be established could/ should consider writing the data bus/ pump code part of the Software-as-Service business model. Also see the use case described below.

4.3 Main use cases and flow-of-events

We identified the following main use cases:

1. Maintenance of the asset and component ***repositories***
2. Maintenance of the scheduling ***project***; nota bene assets and components per asset included
3. Conduct tactical ***asset pool*** scheduling (APS)
4. Conduct tactical redundant ***component usage*** scheduling (CUS)
5. Import & process external data

4.3.1 Maintenance of the asset and component repositories

The main flow for this use case requires the application to be started, but does not need (but may have) a scheduling project open. The system provides the user the opportunity to edit either (1) repository of components or (2) the repository of assets, and edit the specifications of each of those components/ assets.

Editing the component repository

The application maintains a stand-alone repository of components which is not tied to a single asset or single project, notably because particular components are used in several assets. The condition, operating, and maintenance data is stored by the application in a (local) data lake and associated with the particular component ID. One subflow has the user to add, edit, and delete this component-level condition data, as well as registration of actual maintenance activities. Another subflow concerns estimation of degradation parameters from condition data.

The user can create or remove a component from the repository. Upon creating a component, the user is prompted to add specifications w.r.t. usage / degradation behavior (name of the variable, distribution type, initial parameters) and defining when (i.e. which level of the variable) maintenance is required. The user can manually specify degradation parameters, but also add condition and operating observations to estimate them automatically.

Components are simple classes with a degradation rate that depends on (i) operating mode, (ii) ‘season’, and (iii) ‘location’ (where both are categories/ labels). The user is able to set the base-rates for degradation for each of the operating modes as well as the ‘season’ and ‘location’ multipliers. However, the user can also import degradation condition data and an historical operating schedule (including season and location) and allow the estimation of the base-rate as well as the season and location multipliers. In addition to that, the user can specify whether to automatically include a fixed effect variable for each individual asset to which the component is added.

Editing the asset repository

The application maintains a repository of assets not tied to any project, but each asset is associated with an array of components from the component repository.

The user can create or remove assets from the repository. Upon creation of an asset, it is automatically added to the repository. The user has to specify an own ID/ name for reference in projects. The user may specify a “commission date” and a “mandatory survey regime”. Although this is rather specific for the maritime sector,

The user may associate (instances of the) components with the asset, which are stored as references rather than hard-copies to ensure using up-to-date data each time the asset is used.

4.3.2 Maintenance of the scheduling project

Main flow: this use case begins with the user opening/ creating a scheduling project. Unlike in the MaSeLMa project, creating a scheduling project requires adding assets only, no asset or component level information needs to be specified as this has already been done in creating the component and asset repositories.

In an open project, the application offers the following options to the user, each described in a subflow below:

1) creating, opening a project,
2) editing the tactical asset pool of the project,
3) editing the component portfolio of one of more assets,
4) manage component specifications (to be decided)
5) editing the operating schedule template

Creating, opening a project

Currently, the application allows only a single project to be open, such that this subflow starts with the application without any project open. Upon creation of a project by selecting a menu item or pressing the button on the bar, the user is prompted to provide a title. Editing the project itself is handled in other subflows.

Note: upon opening a previously created project, the user gets a warning if anything has changed in the asset or component data of the project since the last time it was open, e.g. observations or maintenance actions of any of the components in an asset have been edited, or the list of components associated with an asset has been edited. Note that such a change can be either in the repository editing subflow or in another project if we decide to provide functionality to edit those aspects of components.

Edit the asset pool of the project

The application shows a list of the assets contained in the scheduler's repository and offers the opportunity to add and remove assets (e.g. vessels, printers) from the pool (e.g. fleet, machine array). We may decide to add an option to allow the user to maintain the asset repository here.

Edit the component portfolio of an asset

Upon selecting an asset, the application enables functionality (enabling a button and menu option) for the user to include/ exclude components associated with the scheduling project. This simply is a checkbox for each of the components, but this will not effectively change the component portfolio of the asset. Note that at this level, the user is not allowed to add / remove components from the asset as this may break/ change other scheduling projects. Note that we may decide to allow it but then throw a warning of which projects will be affected (and also show a warning once such an affected project is opened).

Manage component specifications

It is yet to be decided whether we seek to allow managing component specifications also from a project or rather relegate that to the component repository handler. Moreover, if we allow the user to maintain the component specifications, we may also decide to add an option to allow the user to maintain the component repository from here.

Edit the operating schedule template

In each project, the user may specify an operating schedule template for each component. For the traditional notion of an operating schedule, please check MaSeLMa documentation. In contrast to

the MaSeLMa operating schedule concept, the user can now indicate whether actual usage of a component during an operating interval may be altered by the component usage scheduler (see below). If so, the base-level degradation rate is that of the operating mode, but whether or not the component is actually used is yet to be set by the usage scheduler. If the user specifies that the usage of the component is not allowed to be changed by the component usage scheduler, the component is by definition degrading at the base-level rate.

4.3.3 Conduct tactical (redundant) component usage scheduling

If a single asset is selected, the application enables the user to:

1) select a set of (redundant) components,
2) specify one or more 'target intervals' for each of the components during which maintenance is sought to occur, and
3) run the redundant component usage scheduler (CUS) to determine a pattern how to use each of the components.

See the use case diagram below. Design technically, it is to be noted that the design currently ensures that the user can only select components from one and the same, single asset. Moreover, technically, the user has to select one or more intervals for each component.

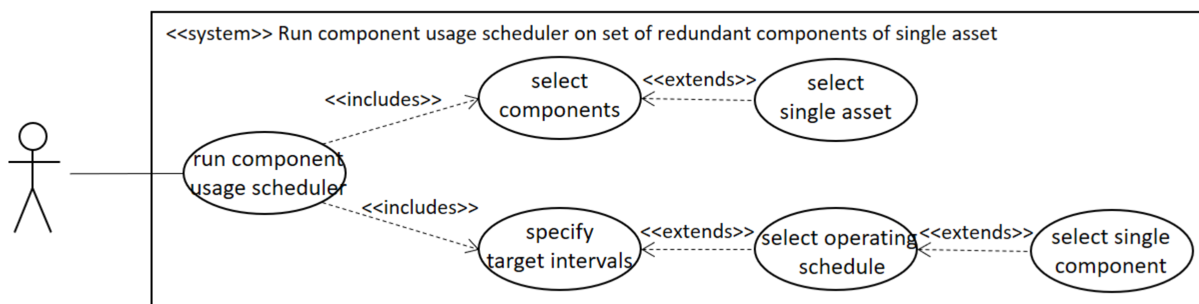


Figure 6. Use case diagram for component usage scheduling

If some of the components selected are indeed redundant, the user can specify how many of each set of redundant components are allowed to be turned off per period.

4.3.4 Conduct tactical asset pool scheduling

The asset pool scheduler (APS) advances/ postpones maintenance actions for target components to fall within particular designated intervals by manipulating asset-level properties (typically the operating schedule). The typical flow of events is that the user:

1) selects a set of assets from the asset list of the project,
2) selects one or more components from each asset,
3) selects designated intervals for each of the selected components during which maintenance is to occur[1],
4) executes the asset pool scheduler (APS) which will look for "swapping moments" at which assets will actually start to follow the operating schedule of the asset it swapped with.

[1] In the MaSeLMaBridge demonstrator, the intervals were derived from the asset commission date following a mandatory survey regime. We may decide that the designated maintenance intervals are again so restrictive rather than allowing picking intervals for each component. However, the user may find it confusing that targeted maintenance intervals are to be picked in case of the component usage scheduler but not in case of the asset pool scheduler.

See the use case diagram below.

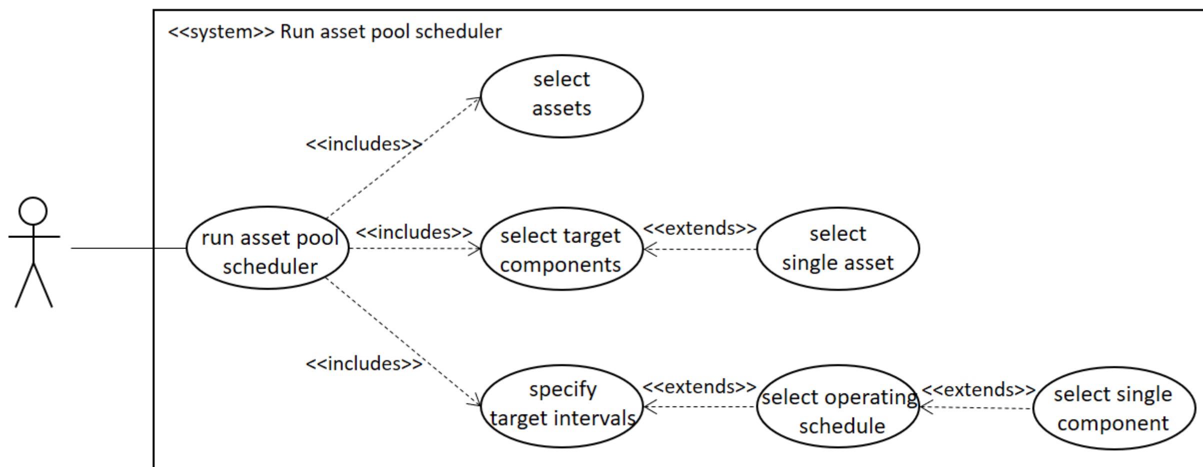


Figure 7. Use case diagram for asset pool scheduling

Somewhat problematic in the current design is that the user may select multiple or different types of components in the various selected asset. Moreover, it is ambiguous what the user wants with the target intervals for maintenance: simply that at least one but preferably more maintenance actions coincide with targeted intervals? Or that the first maintenance action coincides with a target interval?

4.3.5 Import external data

In the component repository handler (dialog), the user can import external data. As the functionality under the hood may well be highly specific, we need to discuss the viability of this solution. For the pilot companies, we can of course provide custom functionality. Arguably, this functionality is part of the Software as Service business model and we may design a module strategy to decouple custom code from the application (e.g. having a universal DLL interface while we build separate DLLs for each of the customers using specific compilation flags).

4.4 Design technical changes

4.4.1 Separation of concerns into two modules in the MaSeLMa API

In the current MaSeLMa API, all component specs, degradation distribution specifications, and condition data are part of a planning project. However, the requirement to handle planning across different assets ("fleet level") each with the same type of component as well as planning of usage for (redundant) components calls for maintaining a data lake outside of any particular project. As such, different projects simply have access to the same asset and component specifications. Program-technically, this also allows for the separation of concerns and two different main modules in the API (see Figure 3)

4.4.2 Revise three-application solution for resource planning?

Currently, the integration with SPMS through an Access interface is rather inconvenient and requires several manual actions of exporting and importing data using Open and Save File dialogs. This three-application solution for the resource planning (spare part availability check, cost determination, etc) is fairly tedious for customers. Although out of scope for now, if we find additional time or resources, we should reconsider this, particularly now that we are moving in the direction of an asset pool (so all assets for which inventory management is to be done are known) and a 'stand-alone module' for management of the data lake.

Inventory optimization typically is something the OEM does to serve the customer base. For asset owners also doing maintenance, our decision to offer 'capital asset *pool*' management (e.g. fleet), it may be convenient to handle inventory optimization also to the GUI. The OEM could/ should then abstract away from individual spare part events at the customer, but look at customer demand.

Moreover, we should see whether we can *automate* this for unattended processing by having the application access the data on a customers' server.

4.4.3 Operating mode issues: user-own operating mode labels, consider tasks, add flags

Now that the application is designed for use in other sectors, it is required that the user can set the labels for operating modes. In addition to that, a 'flag' is required for each of the intervals in the operating schedule specifying whether or not the usage scheduler is allowed to change the pattern of usage.

4.4.4 Integrate and enhance tactical planning

At present, both the 'swapping' scheduler as well as the 'usage pattern' planner run in a stand-alone console tool merely using the JIT maintenance scheduler and outputting a MaSeLMa project that can be opened in the MaSeLMa GUI. Moreover, the tactical planner simply uses a numerical procedure with bisection search to go over the asset swapping moment (basically changing the operating schedules each of the assets follow) as well as the usage patterns. Both are rather ad-hoc.

In the IMOLA project, the following changes are implemented:

1. The tactical scheduling routine is integrated into the MaSeLMA API and placed in a layer just above the operational scheduler (see Figure 3)
2. In terms of functionality the following adjustments are required:
    a. the asset pool scheduler should be generalized for more than 2 assets
    b. the component usage scheduler should be able to work with more than 2 components and should use only intervals that are flagged as 'adjustable'.

The general approach currently sought is a 'multilayered design' in which the tactical planner simply uses the operational planner to ask for maintenance moments. The tactical planner uses a heuristic to search over multiple operational schedules and picks the most suitable one.

Clearly, given the considerable and non-linearly increasing running time, this is suitable only in case of a low number of components and/ or low number of assets. The asset pool scheduler has theoretically $n! * t$ combinations (with n the number of assets involved in the swapping and t the number of periods). Similarly, if m out of n components are to be "on", the usage scheduling problem has $n! / (m! (n - m)!) * t$ combinations. In practice, the number of assets in the APS and number of redundant components in the CUS solvers are expected to be in the order of 2 to 5.

In addition to that, to speed up the search processes, we may seek to limit the degradation distributions allowed and particularly focus on JIT or L10 scheduling, thus basically ignoring clustering and opportunistic maintenance. Note that the MaSeLMa API does also simply allow evaluation of costs and availability for a given operational maintenance schedule, such that a design for the tactical scheduler that builds and evaluates operational maintenance schedules can be very fast (say, in a matter of a few second to several minutes).

4.4.5 Obtaining external data

Given the considerable number of tedious manual actions required to add operating schedule, usage data, etc., it is commendable to use the pilot to study implementing direct data pipes to server

sources/ ERP system for unattended processing. As said above, this is out of scope for IMOLA, but we should make notes on options when deliberating with customers so as to provide

4.4.6 Extension & integration of ProSeLo API

The ProSeLo API offers a fast heuristic for maintenance scheduling capable of handling considerable numbers of components but has basically 1 operation mode, deterministic degradation and no maintenance cost considerations. At present, we seek to integrate the ProSeLo API as separate module into the MaSeLMa API and offer access to functionality through and displaying output in the MaSeLMA GUI. Clearly, this requires conversion of input data for and access to ProSeLo API functionality, and conversion of output data for further visualization and storage in the scheduling project.

In terms of functionality, similar to the MaSeLMa API, the ProSeLo API requires redesign for a ‘multi’ asset setup as well as a way to “throttle” usage of assets. There is no redundancy of particular components in these assets yet (neither foreseen in high tech sector), so whether or not the usage pattern scheduler is to be implemented remains to be discussed. If we decide to provide usage scheduling also in the ProSeLo API, the daily usage should be changed from 1 [on] or 0 [off] to a value somewhere in [0,1].

In addition to this, also the ProSeLo API needs to be adjusted to have spare parts availability checks, notably by using the supply cost vector (with costs associated with emergency, regular delivery) into account in planning.

4.5 Data structure design

Here we discuss when the user is required to manually handle data input or output, where particular data is stored (and we thereby distinguish the repositories in the data lake from the data stored in scheduling projects), and the data pass-throughs that need to be made.

4.5.1 Human actions for data in-/ output

There are four phases in which human interaction is foreseen:

1. Initialization of the application / editing the repository
    a. Edit the asset pool
    b. Edit the component repository
    c. Editing the list of operating modes
2. Creating/ editing a project to reschedule assets or usage patterns
    a. Adding/ removing/ editing assets from a project
    b. Adding/ removing/ editing components to/ from assets
    c. Editing operating schedules for assets
    d. Specifying other degradation factors (season, locality)
3. Exporting/ importing/ manually adding or removing condition data, maintenance actions, etc. to/ from the data. Ideally, degradation curves are determined unattended and parameter values automatically updated.
4. Exporting the maintenance schedule/ dashboard summaries for further processing

Typically, either one or more of these human actions are required upon the occurrence of one or more of the events in Table 1.

4.5.2 Data lake

The following data has to be kept in a data lake (which is, for now, a local repository, particularly taking into account the strict security requirements of DMI/ RNLN):

1. Component specifications, notably degradation distribution, and including manually specified parameters, references to condition time series & maintenance activities per asset, estimated degradation parameters
2. Asset specifications, notably the list of components associated with the asset

4.5.3 Scheduling project

Each scheduling project contains:

1. References to assets, and, for each asset, a list of references to components involved in the project,
2. For each asset in a project ‘targeted maintenance intervals’ (e.g. a period of time in which mandatory inspection is required)
3. An operating schedule for each asset, whereby each operating schedule contains a flag specifying whether it is ‘adjustable’ by the usage pattern scheduler

The project could/ should allow both types of tactical schedulers, e.g. by selecting a (sub)set of assets and selecting/defining an interval for each asset and then pressing ‘asset pool scheduling’, and e.g. selecting a (sub)set of redundant components and selecting/ defining an interval for each component, and pressing ‘component usage scheduling’.

4.5.4 Pass-through

See the section on Design Technical Challenges. In sum:

1. Data conversion and pipes to/ from SPMS remain unchanged
2. A data interface has to be designed for ProSeLo API
3. A data collection tool may have to be designed to collect condition, usage data, etc. from company databases.

4.6 User Interface Design

Although the basic framework for the GUI and the underlying scheduling project structure is in place, the progressive aggregation from purely operational maintenance planning to tactical maintenance & asset management planning requires introducing several new elements across virtually all top-level libraries and certainly several considerable changes to the GUI.

First of all, the GUI is to offer a “repository dialog” in which the user can add/ remove assets to/ from the asset repository, add/ remove components to/ from the component repository, and allow adding/ removing associations of components from that repository to a selected asset.

Secondly, the GUI is to offer a list in that “repository dialog” or a stand-alone dialog in which condition observations of (together with operating information) and maintenance activities of components in particular assets are stored. This information is then to be used in the estimation of the degradation distribution parameters.

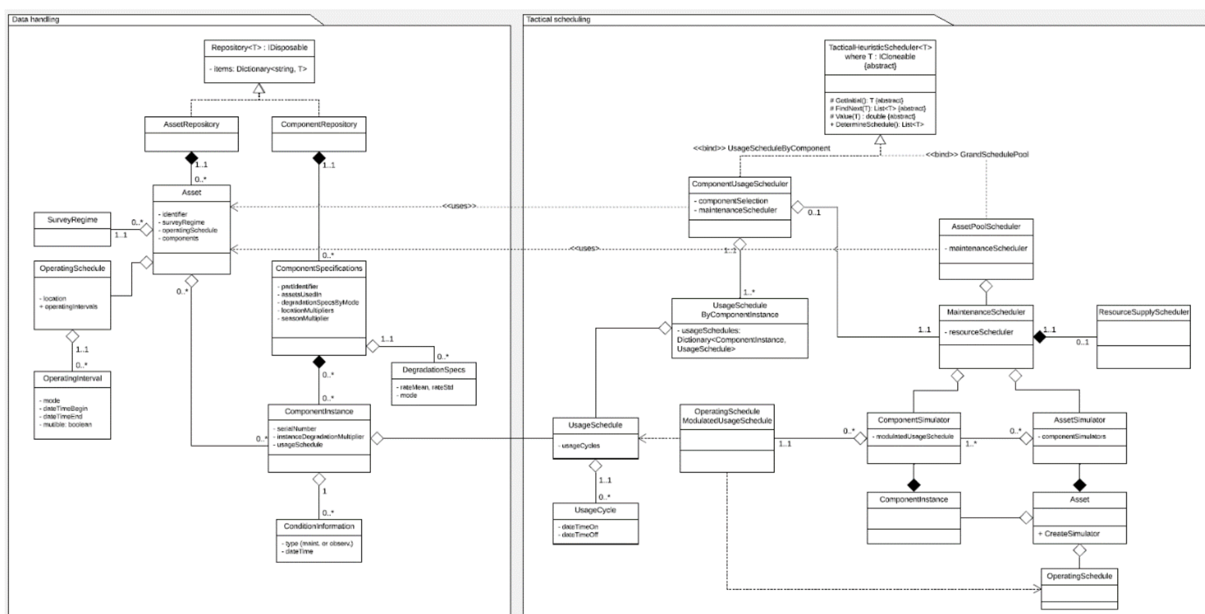
Thirdly, the current project view needs certain alterations. Notably, each project may now contain multiple assets and each asset contains several components. The subsystem part of the GUI may be used for assets. Part of the data handling for components needs to be moved.

Fourthly, the maintenance timeline view is to offer options to specify and select “target intervals” on the timeline of individual components for usage in the component usage scheduler (CUS). In collaboration with the pilot companies, we have to decide how the user can specify ‘target intervals’. A particularly interesting option is to simply add ‘blocks’ to the maintenance scheduling interval, comparable to the mandatory survey intervals in the MaSeLMABridge implementation, and allow the user to double click such a ‘block’ to in/exclude it.

Fifthly, it has been opted to provide a dashboard to represent scheduling information in a more human comprehensible way and to provide a quick summary of what is required of the customer in terms of decisions.

4.7 Class Diagrams

TO DO: SPECIFY WHY / WHICH PART OF ENGINE IS DESCRIBED HERE..



4.8 Use Case Realizations: concurrency

Looking at the use cases and picturing several functions to be called under the hood (e.g. for scheduling), there are several concurrency issues:

1. The Approximate Dynamic Programming MaSeLMa scheduler runs a pool of worker threads. Access to the MaSeLMa API is not yet disallowed during the ADP scheduling process. With customers going to use the application, we have to lock the API without stalling updating of the GUI.
2. Communication with the Python penalty scheduler is asynchronous. It may even be executed through system calls such that a polling thread is to be setup to monitor for output.
3. Communication with SPMS through the integration interface is arguably tedious for customers. As mentioned above, since the new GUI is going to host the full asset pool, further integration of inventory optimization is to be consider (but out of scope for the short IMOLA project).

4. Retrieving and processing external data, notably usage, maintenance and condition data. It is likely that an external tool is to be used to extract, convert and prepare the data for import. It is undesirable that data is extracted each and every run of the MaSeLMa GUI. Note that we can ask the customer to press an "update" button in the GUI, but the data will be added to the application's data lake gradually after which
    a. the components' degradation information repositories need updating
    b. the dashboard should give a warning of trend breaks or maintenance actions requiring rerunning either one of the tactical planners.

For (1), the worker thread pool is in place and sending back an event once done. This easily allows locking the API and/or blocking GUI access (e.g. through modal dialog). For (2), we need to design a technical solution on how to lock the API and/ or block GUI access. This can be an easy solution such as a monitor thread polling the existence of an output file produced by the Python API.

For (3) and (4), the concurrency issues are at stake in peripheral processes started by the user and in part outside of the GUI. With the current design there is no 'neat' solution just yet. However, the user is aware at all time of whether all required data has already been retrieved.