23/01288

FFI-RAPPORT

A NavLab 4 Configuration Framework (CF) to Automatically Verify and Inspect Configurations (INI-files)

Stein Kristiansen

A NavLab 4 Configuration Framework (CF) to Automatically Verify and Inspect Configurations (INI-files)

Stein Kristiansen

Norwegian Defence Research Establishment (FFI) 21 September.2023

Keywords
NavLab
Programvareutvikling
MATLAB
Navigasjon

FFI-report
23/01288

Project number
1684

Electronic ISBN
978-82-464-3492-6

Approvers
Martin Syre Wiig, *Research Manager*
Håkon Storli Andersen, *Director of Research*

The document is electronically approved and therefore has no handwritten signature.

Copyright
© Norwegian Defence Research Establishment (FFI). The publication may be freely cited where the source is acknowledged.

Summary

NavLab 4 is a highly configurable simulation and post-processing tool for navigation. While this configurability is a key strength, its complexity can make it difficult even for advanced users to create functional and complete configurations (INI files) that make NavLab behave as the user intends.

We present a Configuration Framework (CF) to simplify this task via

- an Application Programming Interface (API) and data structures that allow developers to specify the properties, requirements and parameters of the configurations affecting their code and to make this information available everywhere in NavLab via a central register

- a graphical user interface that presents this information to the user in an intuitive way

- mechanisms that use the above-mentioned information to automatically verify the completeness of the user-provided configurations

The verification of configurations that affect a piece of code is performed proactively (before the code is run) whenever the user executes the code. If any required configuration elements are missing, the CF can prevent the execution of the code to avoid potentially irrecoverable errors. In such cases, the CF informs the user of the missing elements via the above-mentioned graphical user interface so that the user can easily identify and add the missing configuration elements.

The task described in the first bullet point is referred to as *adding CF support* to configured code. In addition to developing the CF itself, we have provided CF support to the code that implements all core activities a user can perform in NavLab, including preprocessing, simulation, estimation, calibration of the Doppler Velocity Log (DVL) and summarizing and exporting analysis results. As a result, the facilities in the two last bullet points above are made available for nearly all existing NavLab functionality. The CF API and data structures are designed to be sufficiently flexible, generic, and intuitive to facilitate adding CF support to future extensions of NavLab 4.

Sammendrag

NavLab 4 er et svært konfigurerbart simulerings- og postprosesseringsverktøy for navigasjon. Det at verktøyet er så konfigurerbart, er en av dets styrker, men konfigureringen er også tilstrekkelig komplisert til at det kan være utfordrende selv for erfarne brukere å lage konfigurasjoner (INI-filer) som gjør at NavLab oppfører seg som ønsket og forventet.

Vi har utviklet et konfigurasjonsrammeverk (CF) som forenkler konfigurasjonen av NavLab via

- et programvaregrensesnitt (API) og datastrukturer som gjør utviklere i stand til å spesifisere egenskaper, krav og parametere til konfigurasjonene som påvirker koden deres, og til å gjøre denne informasjonen tilgjengelig overalt i NavLab via et sentralt register

- et grafisk brukergrensesnitt som presenterer denne informasjonen til brukeren på en intuitiv måte

- mekanismer som bruker den overnevnte informasjonen til automatisk å verifisere at konfigurasjonene brukeren har oppgitt, er fullstendige

Den automatiske verifikasjonen av konfigurasjoner som påvirker en gitt kodebit, gjøres proaktivt (før kodebiten kjøres) når brukeren ber om å kjøre denne koden. Hvis det mangler obligatoriske konfigurasjonselementer, kan rammeverket hindre kjøringen og presentere intuitiv informasjon som gjør at brukeren enkelt og raskt kan identifisere og legge til de manglende elementene. Dermed unngår brukeren feil under kjøringen som er forårsaket av manglende konfigurasjonselementer, og som kan kreve at NavLab må starte på nytt.

Aktiviteten beskrevet i første kulepunkt kalles *å gi CF-støtte* til konfigurert programvare. I tillegg til å utvikle selve CF-rammeverket har vi gitt CF-støtte til alle kjernefunksjonene i NavLab, inkludert preprosessering, simulering, estimering, kalibrering av Doppler Velocity Log (DVL) og oppsummering og eksportering av analyseresultater. Dermed er fasilitetene beskrevet i de to nederste kulepunktene gjort tilgjengelig for det meste man kan gjøre i NavLab. CF-API-et og datastrukturene er utformet for å være fleksible, generiske og intuitive nok til at CF-støtte enkelt kan gis til fremtidige utvidelser av NavLab 4.

Contents

Summary 3

Sammendrag 4

1 Introduction 9

2 Method 14

3 Configuration in NavLab 4 15

3.1 Configuration Activities 15

3.2 Configured Objects 17

3.3 Required INI files 19

3.4 Configuration Dependencies 22

4 Goals and Requirements 23

4.1 Goals 23

4.2 Requirements 24

4.2.1 Requirement 1: Proactive Verification 25

4.2.2 Requirement 2: Structured Representation of Configurations 26

4.2.3 Requirement 3: Support for Multiple Parameter Instances 26

4.2.4 Requirements 4: Support for many COs per INI file 26

4.2.5 Requirement 5: Support for many INI files per CO 26

4.2.6 Requirement 6: Separation of Concerns 26

5 The CF API: Central Concepts, Data Structure and Functions 27

5.1 Configurations and Configuration Structures 27

5.2 The Scope of a Configuration 27

5.2.1 Configuration Scope 28

5.2.2 Discussion of alternatives 30

5.2.3 Selected Alternative 31

5.3 Verification Levels 31

5.4 Configuration Versions 34

5.5 The CF Data Structure 35

5.5.1 cConfigurationRegistry 36

5.5.2 cConfiguration and cConfigurationParameter 37

6 **Applying the CF** 39

6.1 The Two Configuration Stages 39

6.2 Stage 1 40

6.2.1 Step 1.1: Create and register the configuration(s) in the global configuration registry 44

6.2.2 Step 1.2: Specify the name of the INI file 45

6.2.3 Step 1.3: Pass the configuration to the superclass 45

6.2.4 Step 1.4: Add configuration parameters relating to this class 45

6.2.5 Step 1.5: Call the *loadParameters* function of all ICOs 46

6.3 Stage 2 46

6.4 Front-End Integration 48

6.4.1 summaryGUI.mlapp 49

6.4.2 Invoking Proactive Verification upon CA Execution 51

6.4.3 Inspection of Configurations upon User Request 52

7 **Summary and Possible Future Work** 54

7.1 Possible Future Work 54

Appendix 56

A **Overview of Configured Code Supported by the CF** 56

References 59

Preface

I would like to thank Kenneth Gade and Einar Berglund for providing valuable feedback and suggestions for this work during several meetings, and for reading through and commenting on this report and the corresponding source code. I would also like to thank Kjetil Ånonsen and Ove Kent Hagen for their valuable feedback and suggestions during these meetings.

Oslo, 31.05.2023
Stein Kristiansen

1 Introduction

NavLab 4 is a generic simulation and post-processing tool for navigation, written in MATLAB (Gade, 2004) (Gade, 2003). It is a highly versatile tool that can be used for a wide variety of navigation tasks, and is based on a solid theoretical foundation to ensure statistical optimality throughout the entire system. NavLab provides facilities for preprocessing of real measurements, simulation of sensors and vehicle movements, calculation of statistically optimal real-time estimates of navigation quantities, improving these estimates via smoothing, and graphically visualizing and exporting data and results. NavLab can be used both with real-world and simulated data, or with a combination of the two. A wide range of sensors for aided inertial navigation systems (AINS) are supported, both for simulation and estimation, including accelerometers, gyroscopes and sensors that measure position, velocity and orientation.

In this document, the term *activity* refers to the execution of a discrete software module in NavLab. This includes the execution of one of the central processing modules, called *Preproc*, *Simulator* and *Estimator*. The term *run* refers to the sequence of activities executed to process a given real-world data set (called *raw data*), and prospectively to augment this data set via simulation. Which activities are available for any given run, and the behavior of each activity, is determined by the run configuration. This configuration exists in the form of user-provided INI files. All activities in NavLab 4 are highly configurable via a large number of both required and optional configuration parameters. Consequently, a large number of INI files are required for any given run, including multiple INI files per simulation, preprocessing or estimation sensor and one or more per activity. Some INI files include a relatively large number of parameters.

Currently, there exists no facility in NavLab to provide the user with a structured, complete and intuitive overview of the large configuration space. As a result, even advanced NavLab users have experienced difficulty remembering which parameters are available, how these affect their run, which are mandatory and which are optional, whether they have default values and if so what these values are. This has led to users unintentionally leaving out required configuration parameters, often resulting in difficult-to-understand or irrecoverable errors. While such scenarios may help the user to identify the lack of mandatory parameters "the hard way", missing parameters with default values may go unnoticed. While default values are useful to reduce the time needed to provide functioning configurations, the fact that they allow a run to proceed without errors may leave the user unaware of their effect on the run. Had the user known about these effects, (s)he may have decided to use different parameter values. Both problems above are exacerbated by the lack of a general system to detect incomplete configurations and inform the user about the missing configuration elements before an activity is executed.

To address these issues, this work provides an extension of NavLab 4 called *the Configuration Framework (CF)*. The overall goal of the CF is to make it easier and less time consuming to create NavLab configurations by increasing the awareness of which and how configuration parameters affect NavLab runs, and to prevent the occurrences of potentially time consuming

runtime errors caused by incomplete configurations. To achieve this, the CF provides the following two facilities:

1. A graphical user interface (GUI) that allows the user to inspect the different run configurations. This GUI shows a structured overview of all configuration parameters in each INI file, including their properties, requirements, descriptions and dependencies, and whether they are currently provided by the user.

2. Mechanisms that automatically and proactively verifies the INI files for completeness before executing the NavLab activity that is affected by them. This prevents the execution of activities upon incomplete configurations, and thereby avoids many runtime errors. Upon failure of verification, the above-mentioned GUI is shown with an intuitive overview of the missing configuration elements.

Currently, NavLab does not include data structures to describe configuration parameters and their properties and dependencies. Both facilities above rely on such a structured representation, and the data structure must be populated by the developers of configurable NavLab code. NavLab has an object oriented implementation, and only a subset of all objects used during an activity is configured with INI files. These objects are called *Configured Objects (CO)*. The CF provides the NavLab developer with

1. generic data structures to store the important aspects of configurations,

2. an Application Programming Interface (API) to store information about configurations in this data structure, and to support automatic verification of configurations, and

3. step-by-step instructions on how to apply the CF to any given CO.

The developer-provided information includes which INI files and parameters are available, which parameters are mandatory, what their default values are (if any), and which dependencies exist between them.

As part of this work, the CF has been applied to all COs that are currently maintained by The Norwegian Defence Research Establishment (FFI). Thus, in this version of NavLab the benefits of the CF are already available for all NavLab activities. In addition, we provide minimal, executable template classes illustrating how a developer can apply the CF in future extensions of NavLab. The CF, its application to COs, and the template classes are available in an extension of the NavLab 4 version described in (Kristiansen, 2022).

This document is structured as follows. Section 2 shortly summarizes the methods applied in this work, which includes code analysis, requirements analysis, design, implementation and application of the CF. Sections 3, 4, 5 and 6 present the results from each of these steps. Section 3 gives an overview of the configured NavLab code, including its overall structure and how its code is configured. This forms the foundation for the goals and requirements of the CF, which

are described in Section 4. Section 5 presents the design of the CF API, beginning with key concepts and discussion of design alternatives, and then the functions and data structures comprising the CF API. Section 6 explains how to use the API to add CF support to COs, to enable user inspection and proactive verification of configurations. CF support is already added to all COs maintained by FFI, and a complete list of these are found in Table A.1 in the Appendix. Finally, Section 7 summarizes the work and discusses possible future work.

A series of concepts and terms are introduced in this document. Each of these are explained the first time they are mentioned in the text. The central concepts and terms are summarized in Table 1.1, and the constituents of, and inter-relationships between the central concepts are illustrated in Figure 1.1.

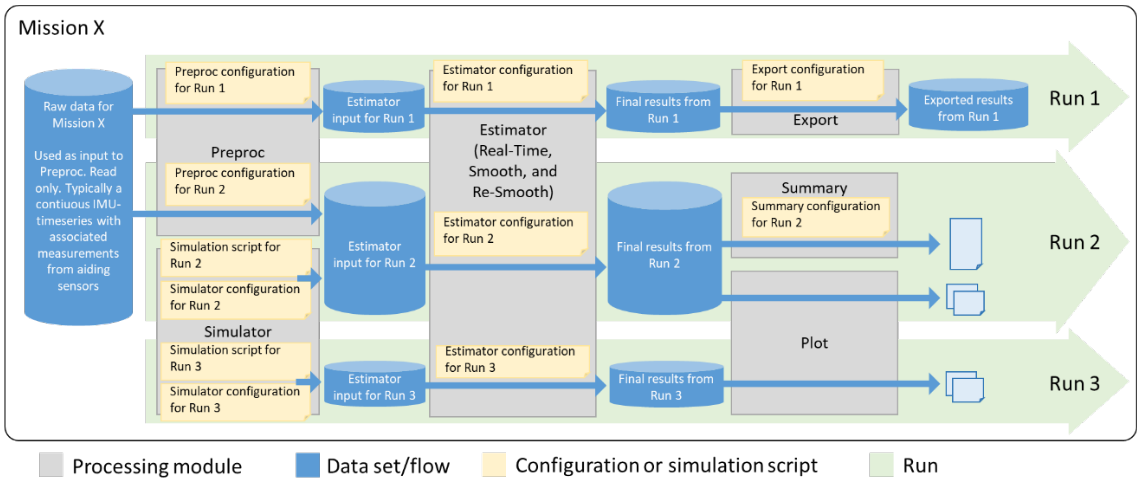


Figure 1.1 Central NavLab concepts and their relationships.

Term / Concept	**Definition and description**
Configuration In Figure 1.1:	The set of parameters in an INI file that controls the behavior of a piece of executed NavLab code. When preceded by the word “Run” or “Activity”, it refers to the configuration of an activity or a run as a whole. When not preceded by any of these words, or by the word “CO”, it refers to the configuration of a single CO. “Activity”, “Run” and “CO” are defined below.
Configuration Framework (CF)	The extension to NavLab 4 described in this document providing support for user inspection and automatic, proactive verification of configurations.

Configuration structure	The data structure to represent a configuration in the CF. This includes information about which INI files it requires, the names and properties of its configuration parameters, and its dependencies on other configuration parameters.
Processing module In Figure 1.1:	A discrete piece of the NavLab software that either produces simulated data or processes simulated or real-world data. The three central data processing modules are *Preproc*, *Simulator*, and *Estimator*. Depending on the context, a processing module may unambiguously be referred to as a *module*.
Activity	The execution of any code implementing a discrete functionality in NavLab. The all code expected to execute to completion as the result of a discrete user request via the GUI is regarded as belonging to the same activity. This includes (but is not restricted to) the execution of processing modules.
Raw data In Figure 1.1:	Data set consisting of measurements from the real world that is used as input to the *Preproc* module. This data set typically consist of a continuous time series of IMU measurements with corresponding measurements from aiding sensors.
Run In Figure 1.1:	A sequence of NavLab activities to process a raw data set in NavLab and/or produce simulated data. Different runs can be used to process/produce the same data set in different ways according to their configuration. The configuration of a run is stored in the form of INI files in a folder called a *run folder*. The current run folder can be selected in the NavLab GUI.
Mission	A collection of run folders, the corresponding raw data and any other files used by multiple runs, including configuration files such as navlab.ini. The folder in which this is collected is called a *Mission folder*.
Estimator input In Figure 1.1:	Data produced by the software modules *Preproc* or *Simulator* that is used as input to the *Estimator* module.
Results	Data produced by the *Estimator* module that can be save to file either via the menu items "Save session" and "Save session as …", or by exporting the data using the *Export* module.

In Figure 1.1:	
Configuration Activity (CA)	A NavLab activity that performs configuration of COs (defined below) during the initial stages of its execution. This includes loading configurations from INI files into memory, verifying these and applying it to COs.
Configured object (CO)	An object used as part of a CA that is configured according to the contents of an INI file (i.e., according to *a configuration*). Such objects contain three types of code: (1) code that loads and verifies its configuration, (2) code that applies the loaded and verified configuration to the CO, and (3) the code that executes part of an activity. CF support can be provided to a CO by implementing the static *loadParameters* function in its class, making its configuration user-inspectable and automatically verifiable.
Indirectly configured object (ICO)	CO that are configured indirectly during the configuration of another CO. An ICO is also a CO, i.e., the term *CO* refers to COs and ICOs collectively. The statement "the CO a of ICO b" and "the ICO b of CO a" is used to mean that ICO *b* was configured indirectly during the configuration of CO *a*.
CO and ICO classes (abbreviated *classname* (I)CO)	The classes from which COs and ICOs are instantiated, respectively.
Template class	Minimal, executable template classes illustrating how a developer can apply the CF to COs. Their code constitutes a starting point for the *loadParameters* function that can be copied and pasted into COs.

Figure 1.2 Central terms and concepts used throughout this document.

2 Method

As with most software solutions, the CF is designed according to a set of requirements. To understand which requirements need to be satisfied, it is first necessary to understand how configuration is currently performed in NavLab, which in turn requires an understanding of the overall structure of NavLab. The work towards the completion of the design, implementation and application of the CF was carried out through the following four steps:

1. **Code analysis:** The NavLab code was first carefully investigated with two goals in mind: (1) understanding the overall structure of NavLab and identifying the separate activities making up the processing pipeline, and (2) identifying patterns in the configuration code that are common throughout NavLab, as well as deviations from these patterns that need to be accounted for by the CF. The investigation was performed by studying the code, executing and stepping through the code with a debugger, and multiple discussions with the NavLab users and developers at the FFI. The output of this step is mainly documented in Section 3 and Figure 3.1, is used as the input to Step 2 below.

2. **Requirements analysis:** Based on the findings in Step 1, a set of requirements were developed. Fundamentally, the CF had to be able to represent all existing configuration parameters in NavLab while at the same time addressing central shortcomings of the way NavLab was configured. The requirements were developed during presentations and subsequent discussions with the key NavLab developers and users at FFI. The results from this step is mainly documented in Section 4, and is used as input to Step 3 below.

3. **Design and implementation:** After the initial set of requirements was formulated, an initial design was proposed, presented and discussed to/with fellow NavLab users and developers. The outcome of the discussions was used to refine both the requirements and the design, and Steps 2 and 3 was repeated iteratively. To unveil issues with the design, parts of it was implemented, and applied to COs, between the iterations. The process was iterated until the design and implementation converged into a satisfying solution. This step is documented in Sections 5 and 6. The output of this step was used as input to Step 4 below, and is reflected in the extension of the NavLab code.

4. **Application of the CF:** The CF was applied to all COs during and after the CF design and implementation. Remaining issues with the CF during was identified and corrected during this period. The output of Step 4 is mainly documented in Section 6 and in Table A.1, and reflected in the extension of the NavLab code.

3 Configuration in NavLab 4

Before we can develop the goals, requirements and design of the CF, it is first necessary to understand how configuration is performed in NavLab 4. This is achieved by understanding which activities can be performed in NavLab (explained in Section 3.1), the common approaches and mechanisms utilized to configure objects based on INI files (explained in Section 3.2), the INI files required for each activity (explained in Section 3.3), and which (types of) dependencies exist between the configurations (explained in Section 3.4). In these explanations, the term *configuration* refers to parameters in an INI file used to configure a given piece of NavLab 4 code.

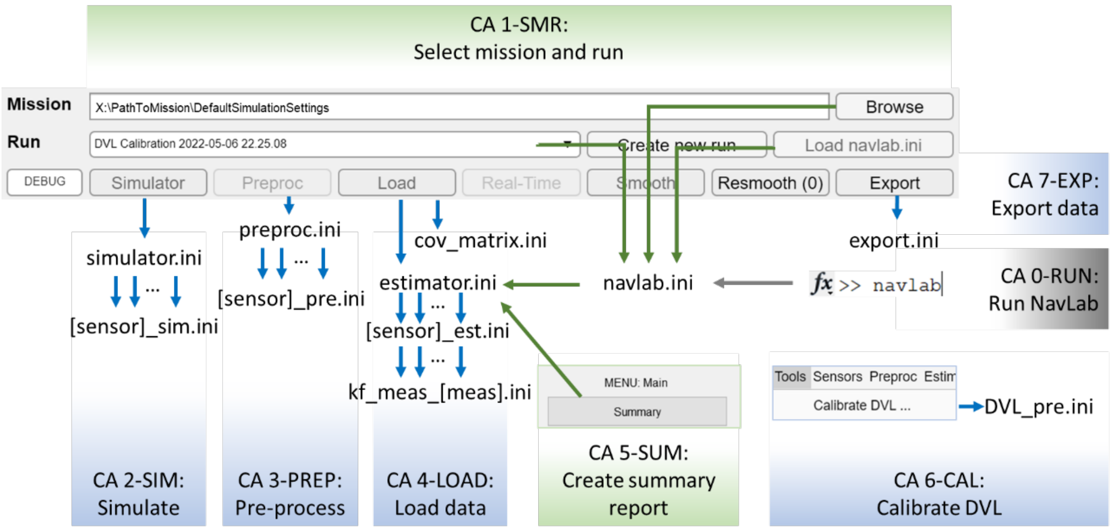


Figure 3.1 Structure of NavLab 4. The GUI shown is the top part of the main GUI window in NavLab 4. The blue and green boxes and arrows represent separate CAs and the INI files they use for configuration, respectively.

3.1 Configuration Activities

Figure 3.1 gives an overview of how NavLab 4 is structured. The figure refers to the concepts *mission*, *run* and *Configuration Activity (CA)*. To understand the NavLab structure, we must first clearly define these concepts and the ones they depend on. See also Table 1.1 for a summary of the definition of all concepts central to this document.

A *run* is a sequence of NavLab activities to process a raw data set in NavLab and/or to produce related simulated data. The configuration of a run is stored in INI files in a folder called a *run folder* which is selected in the NavLab GUI. A *mission* is a collection of run folders, the corresponding raw data and any other files used by multiple runs, including configuration files

such as navlab.ini. The folder in which this is collected is called a *Mission folder*. A *NavLab Activity* (or simply *Activity*) is the execution of any code implementing a discrete functionality in NavLab. An activity encompasses all code that is expected to execute to completion as the result of a discrete user request via the GUI. Such code can include, but is not restricted to, processing modules such as the simulator, preprocessor, estimator, etc. The subset of NavLab activity that reads, verifies and applies configurations as the initial part of their execution are called *Configuration Activities (CAs).* Under this definition, estimation, smoothing, re-smoothing and plotting are activities, but *not* CAs. The activity to loads the data used by these activities (called 4-LOAD) *is* a CA.

The arrows in Figure 3.1 show which INI files are read by which CA, and the order in which they are read. All INI files accessed by a CA are pointed at by arrows with the same color as the CA. For instance, CA 1-SMR only uses the two INI files navlab.ini and estimator.ini while CA 0-RUN only accesses navlab.ini.

Notice from Figure 3.1 that a subset of the INI files read by one CA may also be read other CAs. For instance, although estimator.ini is primarily used by CA 4-LOAD (*Load data*) to configure the estimator and smoother, it is required already during CA 1-SMR (*select mission and run*), and CA 5-SUM (*create summary report*)[1]. We also see that several activities access navlab.ini: CA 0-RUN (*run NavLab*)[2] and CA 1-SMR[3].

The different CAs in the figure are the following:

CA 0-RUN: Run NavLab. This CA is executed by issuing the command “navlab” in MATLAB. It creates of the main GUI window and initializes central NavLab data structures, including a globally accessible cNavLab object and CF registry (described in Section 5.5.1).

CA 1-RMS: Select a mission and run. This activity is executed either when selecting a new mission, upon which the mission’s default run will automatically be selected, or when selecting a run within the same mission.

CA 2-SIM: Performing a simulation. This is executed by pressing the button labelled “Simulator”.

CA 3-PREP: Preprocessing data. This is executed by pressing the button labelled “Preproc”.

[1] To determine for which bias values to generate warnings and errors in the summary report.
[2] To configure the cNavlab object created during the startup code. NavLab operation requires the existence of a cNavLab object. During the startup code, such an object is created, then its function *lookForNavLabIni* is invoked. This function configures the newly created cNavLab object using any navlab.ini it can find by first searching the folder containing the executed navlab.m/exe and then the folders in the MATLAB path.
[3] Whenever the cNavLab object is re-configured according to the navlab.ini in a newly selected mission or run.

CA 4-LOAD: Loading data. Data for the estimator is either created via simulation and/or as a result of preprocessing real data. This data is loaded into memory by pressing the button labelled "Load".

CA 5-SUM: Create summary report. This CA is executed by pressing the button labelled "Summary" in the plot menu, which is shown when pressing the button "Plot General" in the main window.

CA 6-CAL: DVL calibration. This CA is executed via the GUI shown with the menu item "DVL Calibration …" under the "Tools" menu.

CA 7-EXP: Exporting data. This CA is executed by pressing the button labelled "Export".

Only a subset of the CAs are typically executed for any given run, and some CAs require that others have been previously executed. For instance, CA 4-LOAD (*load data*) is normally executed after CA 2-SIM (*simulate*) and/or CA 3-PREP (*preprocess*). It is not the concern of the CF to detect and prevent errors caused by an incorrect order of execution, as this is already prevented by GUI mechanisms that (de)activate the appropriate GUI components.

Typically, the CAs are executed in the order listed above. The exceptions are CA 5-SUM and 6-CAL, which may be executed in arbitrary order. Furthermore, CA 5-SUM relies on the prior execution of the estimation activity (which is *not* a CA, as explained above). Whether CA 2-SIM and/or CA 3-PREP are executed depends on whether the run involves simulations and/or preprocessing of real-world data.

3.2 Configured Objects

Since the implementation of NavLab 4 is object oriented, the different CAs listed above involve the execution of code in many different objects. Only subset of these, called *Configured Objects (COs)*, are configured according to parameters in INI files. Typically, COs are configured immediately after instantiation using one or more of its member functions. These functions generally perform two overall tasks:

Task 1: Load and verify the contents of the INI file used for configuration. The contents of INI files is read using the function *getAllParam*. Searching for the use of this function helped understand where and how INI files are used in NavLab. Figure 3.1 was partly the result of this process. Exactly how configurations are verified varies among the COs, but typically include conditional statements that identify missing parameters. Upon the lack of mandatory parameters, error messages may be displayed to the user, or exceptions may be thrown and handled by a function further up in the call stack. This exception handling does not guarantee graceful error recovery, especially if the CO was configured at a late stage in its CA. Optional missing parameters may be given a default value or may simply not be used.

Task 2: Apply the verified configuration to a CO. In general, arbitrary actions may be taken based on the parameter values, but often the values are simply stored in object properties and accessed by subsequently executed code that is affected by their values.

If the verification in Task 1 is passed, the remainder of the CA is executed after both Task 1 and 2 completes, which may include the configuration of other COs.

The INI files loaded in Task 1 contain syntactically correct MATLAB assignment statements that specify which values are given to which parameters. *getAllParam* function obtains all lines with such statements and passes these to the MATLAB *eval* function. The resulting values are stored in a MATLAB struct with field names equaling the parameters' names. The values can be numerical or logical, strings, function handles or cell arrays.

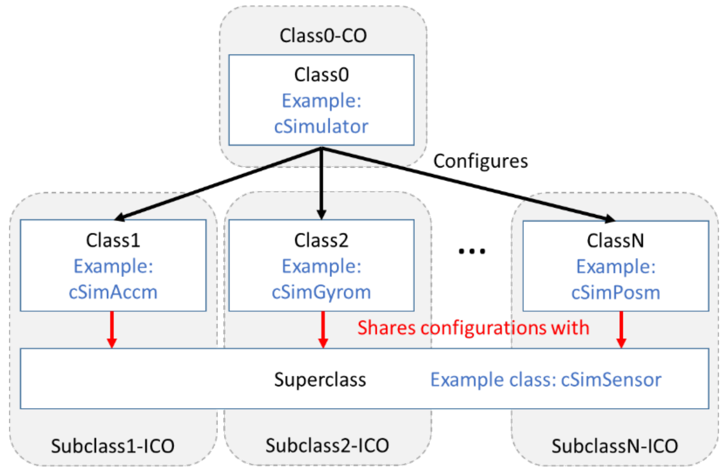


Figure 3.2 The relationship between COs, ICOs, and their classes.

NavLab makes extensive use of class inheritance. Thus, the functions that configure a single CO is commonly distributed across several (super/sub)classes. Figure 3.2 illustrates the relationship between classes (in blue) and their COs, using as an example the CA 2-SIM (*simulate*). The classes include the simulator class *cSimulator*, and those modelling the simulated sensors (inheriting from *cSimSensor*).

Once a configuration parameter controls the code in a given class, it naturally becomes the responsibility of that class to handle that parameter via Task 1 and 2 above (called *separation of concerns*). However, a CO not only contains code and variables defined in the class it was instantiated from, but also from all classes further up in the class hierarchy. For instance, a CO instantiated from the *cSimAccm* class (called *cSimAccm CO* for brevity) contains code and variables from both the *cSimAccm* and *cSimSensor* classes. For this reason, different COs sharing the same superclass each obtain their own, separate copy of the configuration parameters in said superclass (e.g., *cSimSensor*) and the value may differ among these copies. An example is the *h* parameter of *cSimSensor* that determines the length of the simulated duration between subsequent sensor measurements: all simulated sensors obtain a copy of this

parameter, typically with differing values. In summary: sensors such as *cSimAccm* CO are not only affected by the configuration parameters in their own class (*cSimAccm*), but also by those in their superclass (*cSimSensor*).

It is commonly the case that the configuration of one CO is invoked indirectly via the configuration of another CO. COs of the first kind are called Indirectly Configured COs (ICOs). Typically the list of ICOs to configure is provided as a configuration parameter for another CO. For instance, cSimulator CO invokes the configuration of all *cSimSensor* ICOs listed in its configuration parameter *esSensorName.* This is shown in Figure 3.2. As another example: the *cKalmanFilter* CO invokes the configuration of all *cSensor* and *cKfMeas* ICOs listed in its *esSensorName* and *esKfMeasParamFile* parameters. A similar situation occurs in CA 3-PREP (*preprocess*). These situations are seen in Figure 3.1 as a set of parallel arrows pointing to each ICO's INI file.

Note that an ICO is also a CO. Therefore, the term *CO* can be used to refer to COs and ICOs collectively. From here on, we use the statements "the CO a of ICO b" and "the ICO b of CO a" to mean that ICO *b* was configured indirectly as part of the configuration of CO *a*.

Table A.1 lists all COs and ICOs used in the CAs in NavLab.

3.3 Required INI files

The configuration of the COs in any given CA is based on a series of INI files. There are several pre-defined types of INI files, each of which can contain a specific set of configuration parameters. Some types of INI files have pre-defined names, while others are user-defined but follow certain naming schemes. The different types of INI files that exist in NavLab, their names or naming scheme, their contents, and their usages are as follows:

> **navlab.ini:** If a file with this name exists in the same folder as the NavLab executable (i.e., navlab.m or navlab.exe), or in the MATLAB path, it is read upon starting NavLab (CA 0-RUN) to configure the initial cNavLab object. This configuration is however updated upon the selection of a mission or run is selected (via CA 1-SMR) with its own navlab.ini. Its parameters specify the locations of the data and INI files used in the selected mission or run. It may also specify functions to execute upon central runtime events, i.e., when starting NavLab, right after selecting the mission or run, and before estimation starts and after it completes. It differs from the other INI files in that the keywords "<RUN>" and "<MISSION>" can be used to refer to path of the currently selected mission and run.

> **simulator.ini:** Used in CA 2-SIM (*simulate*), and its location is given by the parameter *sSimulatorIniLocation* in navlab.ini. It specifies the values of parameters that determine the behavior of simulation, such as the start and stop times, the time step duration, and the random number generator to use. It also contains the list of INI files of the sensors to

simulate[4] (described in the item immediately below), each of which are modelled by ICO of cSimulator. Note that the description of the simulated trajectory is not specified in simulator.ini, but instead in the MATLAB file *trajrate.m*. Since this is not an INI file, but is instead executed alongside the remaining NavLab code, it is not and need not be handled by the CF.

[simulated sensor name]_sim.ini: Used in CA 2-SIM (*simulate*), and are located in the same folder as simulator.ini. Each file specifies the parameter values for the simulation of one of the sensors listed in simulator.ini. These include parameters of the random processes used to model errors in the sensor measurements. Depending on the sensor, these errors have up to three contributions: (1) a white measurement noise, (2) a bias error, and (3) a scale factor error. The INI files contain values for the standard deviation (SD) of these three types of errors. The two latter types are typically modelled using gauss-markov processes, and therefore in addition have parameters specifying their time constant. Additional parameters include those for a sensor's sampling frequency, the name of the class simulating it, the dimensionality and names of its measurements, the format of the file storing its measurements, and any other parameters specific to the particular sensor type.

preproc.ini: Used during CA 3-PREP (*preprocess*). Its location is given by the parameter *sPreprocIniLocation* in navlab.ini. Its parameters describe overall aspects of how the preprocessing of sensor data should occur, including how to determine the start and stop times of preprocessing, the lever arm reference point, and the list of sensors from which there is data to preprocess. The latter list contains the names of the sensors' INI files (described in the subsequent list item).

[preprocessed sensor name]_pre.ini: Used during CA 3-PREP (*preprocess*)[5]. Their locations are given by the parameter *sSensorPreIniLocation* in navlab.ini. In addition to specifying the name of a sensor's preprocessing class, the parameters in this file determine how its data should be preprocessed. These include, among others, parameters related to outlier detection, sensor misalignment, constant errors, lever arms, how to read sensor data from binary files, formatting during plotting, sensor quality and functions used to transform measurement time stamps.

estimator.ini: Used during CA 4-LOAD (*load data*), CA 0-RUN (*start NavLab*) and CA 5-SUM (*create summary report*), and its location is specified by the parameter *sEstimatorIniLocation* in navlab.ini. Its parameters describe how estimation, i.e., Kalman filtering, smoothing and re-smoothing, should be performed. This includes parameters specifying the initial values for the covariance matrix, state vector, estimated position, orientation and velocity. Other parameters determine whether a spherical or elliptic earth model should be used, whether or not exact (via the matrix exponential) or

[4] Certain sensors can also be specified via logical values, i.e., a sensor of this type is simulated if *[sensor]_available* is set to *1*, where *[sensor]* is the sensor type.
[5] DVL_pre.ini is also used in CA 6-CAL (DVL calibration).

approximate discretization of the system model should be used in the prediction step, whether or not the real (typically simulated) trajectory should be used for linearization, the desired start and stop times for estimation and the number of Kalman filter time steps between each re-set of the navigation equations. Two central parameters are those that specify the list of sensors used for estimation and the list of measurements used in the Kalman filter. The sensors and measurements are specified in terms of strings used to determine the name of their INI files (described in the next list item).

[estimated sensor name]_est.ini: Used during CA 4-LOAD (*load data*), and their locations are given by the parameter *sSensorEstIniLocation* in navlab.ini. Each file specifies the parameter values used by the Kalman filter to model errors in the measurements of a given sensor. Since both the simulator and the estimator model these errors as gauss-markov processes, many of the parameters in [simulator sensor name]_sim.ini are also found in the corresponding [estimated sensor name]_est.ini. This includes parameters that characterize up to three contributions of the overall measurements error, i.e., (1) a white measurement noise, (2) a bias error, and (3) a scale factor error. The INI files contain values for the standard deviation (SD) of these three types of errors. The two latter types of errors are typically modelled using gauss-markov processes, and therefore have an additional parameter quantifying their time constant. Additional parameters specify the name of the sensor class used for estimation, which other sensors the sensor depends on, the mathematical names of its measurements, how the measurements should look in plots, and any other parameters specific to the particular sensor type.

cov_matrix.ini: Used during CA 4-LOAD. Its parameters specify the initial covariance of the errors in position, velocity and orientation that is estimated by the kalman filter. The quantities are calculated using the navigation equations, and are therefore only used in the class *cNavEq*. Each error is specified as values in the x , y and z directions, and all nine parameters are mandatory.

kf_meas_[measurement name].ini: This file is used during CA 4-LOAD (*load data*), and their locations are given by the parameter *sSensorEstIniLocation* in navlab.ini. Each run must have one such INI file for each sensor that produces measurements for the measurement update stage in the Kalman filter. Their parameters specify the name of the class representing the measurement, which sensors are used to produce the measurement, and a function used for in-filter outlier removal (called a *gain function*).

export.ini: This file is used during CA 7-EXP (*export data*), and its location is specified by the parameter *sRunFolder* in navlab.ini. It contains parameters describing how data should be exported when this is requested. This includes parameters to determine the destination folder and format of the exported data and whether or not results from real-time estimation and smoothing should be exported,

Table A.1 in the appendix provides an overview of all INI files required in each activity, and the COs and ICOs that use them.

3.4 Configuration Dependencies

In NavLab 4, the location of INI files can only be determined via the parameter values in different INI files; Section 3.3 lists several examples of INI files whose locations specified in *navlab.ini*. Dependencies can also exist between two individual configuration parameters, e.g., the default value of one parameter may be determined by the value of a parameter in a different configuration. For instance, the default value of the time step length of a simulated sensor (*h* in *[simulated sensor name]_sim.ini*) is set to the time step length of the simulator itself (*h* in *simulator.ini*). Dependencies between (parameters in) different configurations are called *inter-configuration dependencies*. All such dependencies are listed in Table A.1.

Dependencies may also exist between parameters in the same configuration (*intra-configuration dependencies*). Examples of this are found in the configurations for sensor classes inheriting from *cSimSensor*. Here, the names of some parameters can only be determined using the value of other parameters in the same INI file. For instance, the sub string *Df* in the parameter name *init_Df_acc_bias_x* in *IMU_sim.ini* is given by the parameter *sBiasMathName* in that same file.

Configurations and configuration parameters that depend on other configurations and/or configuration parameters are referred to as *dependent configurations* and *dependent configuration parameters*, respectively.

4 Goals and Requirements

Section 3 unveils three main shortcomings in the way NavLab is configured today. First, there exists no available API and data structures to allow a developer to represent configurations in a common structure in terms of their parameters, parameter properties and dependencies (called a *configuration structure*). Such a configuration structure would specify, among other things, which parameters are available, which are mandatory, and which (if any) default values they have. Second, there is no intuitive way for a user to inspect a configuration structure and the current values of its parameters from user-provided INI files. This would make it possible to easily obtain an overview of available and mandatory parameters for any given CA and the default values that affect their run. Third, there exists no generic mechanism that can leverage information in configuration structures to automatically verify the completeness of user-provided configurations, i.e., whether they satisfy the minimal requirements for error-free CA execution.

Prior to the CF, custom verification code had to be implemented by the CO developer, and was carried out in different ways in different CO classes. This was typically done using conditional statements on parameter values after being loaded into memory using the *getAllParam* function. The exact manner in which this was done, and the degree to which the user was informed about missing configuration elements, varied among the CO classes. Furthermore, graceful recovery of errors caused by missing configuration elements was impossible if previously executed CA code affected the program state irreversibly.

4.1 Goals

The above shortcomings in NavLab 4 can be addressed by: a framework with data structures that accurately represent the structure of NavLab configurations, a GUI that allows the user to inspect existing configurations, and code to proactively verify the completeness of a user-provided configuration to avoid errors during the execution of CAs. The work documented here contributes with a configuration framework (the CF) aiming to address the above shortcomings. It is designed to meet the following three goals:

Goal 1: Provide a framework that can be used for all COs in NavLab with data structures that accurately capture the structure of configurations, i.e., in which developers can properly store and describe their configurations in a uniform way.

Goal 2: Provide a front-end that presents the configuration structure from Goal 1 in an intuitive way to the NavLab 4 user, including a list of required and available configuration parameters and their properties, making it easy fix or improve configurations.

Goal 3: Provide mechanisms that leverages information in the configuration structures from Goal 1 to automatically identify incomplete configurations that may result in

irrecoverable malfunctioning of a CA. If the configuration is incomplete, the CF should prevent the user from executing said activity and instead present intuitive information about the actions needed to fix the issue. The front-end from Goal 2 can be re-used for this purpose.

Note that that the CF targets configurations specified in INI files. It is not, for instance, designed to prevent errors due to missing data or the incorrect order of CA execution. Neither does it currently provide means by which to facilitate the creation of INI files, although the front-end in Goal 2 could easily be extended for this purpose. The latter is described as part of future work in Section 7.

4.2 Requirements

This section explains the requirements the CF must satisfy to meet the three goals above. We first list the requirements, then detail each requirement in separate subsections. In addition to the requirements below, the design should strive to make the CF sufficiently structured to facilitate efficiency, and sufficiently flexible to account for exceptional, yet important use-cases.

The requirements the CF should meet are the following:

Requirement 1: Enable proactive configuration verification and inspection, i.e., before executing the CA using the configuration. Proactive verification first requires NavLab to be divided into a set of discrete CAs, each of which may or may not be executed in any given run, and each of which as a whole are subjected to proactive verification before execution. At this point, i.e., before the CA itself is executed, the COs used during the CA are yet to be created. Thus, proactive inspection and verification of configurations must be designed to work in the absence of the COs using these configurations.

Requirement 2: Structured representation of configurations, to facilitate automatic configuration verification and intuitive presentation to the user. This includes the ability to represent configuration dependencies and important properties of configuration parameters.

Requirement 3: Adherence to object orientation. As described in 3.2, one consequence of object orientation is that multiple COs may have their own copied o the same configuration parameter, potentially with differing values. This occurs both due to inheritance and because several COs may be instantiated from the same CO class. The CF must therefore allow different COs to have different values of the same configuration parameter.

Requirement 4: Support for many COs per INI file. Several configurations may use data from the same INI file, and each configuration might need only a subset of all parameters in that INI file.

Requirement 5: Support for many INI files per CO. A CO may need parameters from more than one INI file.

Requirement 6: Separation of concerns. Configuration parameters controlling the code in a given CO should be handled (loaded, verified and applied) by functions in the class of said CO. The CF should also avoid dependencies between COs, e.g., in the form of replicated code.

4.2.1 Requirement 1: Proactive Verification

Upon the presence of an incomplete configuration, part of the NavLab activity using that configuration may execute successfully for some time before the missing configuration causes an error. In the absence of the CF, such an error may result in an informative message to the user, or manifest as a less user-friendly programming exception, depending on the particular error handling code in the affected CO. And whether or not NavLab can gracefully recover from the error depends not only on the error handling code in the affected CO, but also on the effects of executing all code prior to the that of the CO where the error occurred. The reason for the latter is that the NavLab state may have been modified irreversibly by this prior code under the expectation that the activity would run to completion.

Modifying all NavLab code to account for the possibility that some late code may fail is obviously not feasible. To this problem, the CF instead needs to identify any missing parts of a configuration that cause errors *proactively, i.e., before the activity is executed* (called *proactive verification*). Such verification is invoked immediately after the user requests the execution of a CA, and proceeds to the actual execution of said CA only after the configurations of all COs used by CA are verified. By identifying missing configurations before the activity is executed, the user can be prevented from executing an activity that would otherwise result in an irrecoverable error.

As mentioned above, proactive verification first requires the division of NavLab execution into the activities that are subjected to proactive verification, called configured activities (CAs). We must therefore first determine the rules by which to divide code execution into separate activities. Based on our previous discussions we define all code execution resulting from a single, discrete user request via the GUI as belonging to the same activity[6]. Only a subset of all possible activities may be included in any given NavLab run, and only those configurations utilized by these activities should be required by the used and verified.

The COs used during an activity are typically created at an early stage of its execution, and are configured shortly after. Since verification now occurs before the activity is executed, these COs are normally not available at the time of verification. Thus, inspection and verification must be performed without access to CO instances.

[6] Furthermore, code-modules that implement clearly separated functionalities can be regarded as belonging to separate activities.

4.2.2 Requirement 2: Structured Representation of Configurations

Automatic configuration verification involves investigating whether a user-provided INI file meets all the requirements defined in the configuration structure. For this task to be tractable, it is necessary to represent the specification in a well-structured form, i.e., with a data structure that specifies the parameters making up a configuration, their names and properties and the dependencies between parameters and configurations. It should be specified whether a parameter is mandatory or optional, and whether or not a parameter has a default value, and if so, what the default value is. In general, it is necessary to design a data structure that properly captures all aspects that are common among NavLab configurations, and with sufficient flexibility to allow for important exceptions.

4.2.3 Requirement 3: Support for Multiple Parameter Instances

Since the design and implementation of NavLab 4 is object oriented, nearly all code is implemented in classes organized in a class hierarchy. If several COs are instantiated from the same class, each CO may need its own copy of the configuration parameters for the class. If different COs are instances of classes with a common set of super- and subclasses, they each need one copy of the subset of the configuration parameters belonging to those super- and subclasses. The data model used in the CF must therefore allow different COs to have different values for the same parameters.

4.2.4 Requirements 4: Support for many COs per INI file

As seen in Figure 3.1, several CAs may need one and the same INI file. This should therefore be supported by the CF. Only a subset of the parameters may be required by a given CA, depending on which COs are used in that CA.

4.2.5 Requirement 5: Support for many INI files per CO

A CO may be configured using several INI files. This is currently the case with *cNavEq* COs, and should therefore be supported by the CF.

4.2.6 Requirement 6: Separation of Concerns

The CF should not introduce dependencies between m files and classes, e.g., in the form of replicated code. Upon changes in the code, such dependencies may necessitate making the same modification in several places, which implies unnecessary work, reduces code readability, and increases the risk of errors caused by inconsistent code.

5 The CF API: Central Concepts, Data Structure and Functions

This section defines concepts that are central to the CF, and needed to design its data structures and functions. We first determine what we mean by a configuration and configuration structure (Section 5.1). Then, we discuss what should be included in the scope of a configuration (Section 5.2). Section 5.3 then explains a central concept in the CF, namely the *verification level*. Another CF concept, the *configuration version*, is described in Section 5.4. Finally, in Section 5.5, we present the CF data structures designed to represent and verify configurations.

5.1 Configurations and Configuration Structures

NavLab must be extended to allow a developer to specify the data structure of configurations, called a *configuration structure,* in such a way that the requirements in Section 4.2 are satisfied. It is important to understand the distinction between a configuration and a configuration structure. A configuration structure is a data structure that represents the build-up of a configuration, i.e., how its constituents are structured and inter-related. This includes configuration parameters and their properties and requirements (default values, whether they are mandatory, etc.), the name and location of an INI file, and any dependencies the configuration has on other configurations. Note that a configuration structure does *not* itself include the values of its parameters, it only describes the structure of the configurations that do contain these values. The values themselves are stored thus stored in a *configuration* and are obtained from the INI file specified in their configuration structure. A configuration structure is provided by the NavLab developer, a configuration is provided by the NavLab user. As a result, different NavLab runs often use the same configuration structures, but different configurations.

5.2 The Scope of a Configuration

Before we can represent configurations in the CF, and verify these, it is necessary to clearly determine the scope of a configuration. This requires establishing the criteria by which to determine what should and should not be included as part of a configuration and the code affected by it.

During the CF design, the scope of a configuration was eventually determined to be: all parameter values controlling the code in *parts of, or all of, one single CO*. A configuration thus encompasses a subset of the parameters given in an INI file, and can never affect more than one CO. Note that one INI file may nevertheless contain parameter values belonging to several configurations, in which case the INI files contain parameters values affecting several COs. Furthermore, a CO may use several configurations, and thus several INI files.

The reasoning behind the above choice of scope is not obvious. This section therefore describes comparisons of a series of possible configuration scopes, the degree to which they satisfy the requirements identified in Section 4.2, and why we chose the alternative we did.

During the presentation of the alternatives below, the loadParameters function is mentioned repeatedly. This function is explained in detail in Section 6. For now, it suffices to know that **the code in *loadParameters* it executed before each CA to (1) create configuration structures denoting the requirements of the configurations used in the CA, and (2) proactively verifying the user-provided configurations (INI files) according to these structures.** Verified configurations are stored in a global registry accessible from anywhere in the NavLab code.

5.2.1 Configuration Scope

	Requirement					
	1	2	3	4	5	6
Alternative A	✓	✓	✓	✓	✓	
Alternative B	✓	✓	✓		✓	
Alternative C	✓	✓		✓	✓	✓
Alternative D	✓	✓	✓	✓		✓
Alternative E	✓	✓	✓	✓	✓	✓

Figure 5.1 Overview of which requirements (columns) are satisfied by which configuration scope.

We only want proactive verification of those configurations that are actually used during the NavLab activities performed by the user. Before we can create data structures to describe such configurations, we must determine what a configuration regards, i.e., its scope. **The scope of a configuration must be determined in order to determine which parameters belong to a configuration and which code is affected by a given configuration.** We identify six possible scopes, and discuss the degree to which each one meets the requirements in Section 4. This is summarized in Table 5.1. The scope we finally chose for the CF, Alternative E, is based on these discussions. Before we present the discussions, we describe each alternative as follows:

Alternative A: One configuration regards a complete CA. In this case, one *loadParameters* function is created per CA to load and verify the configuration of the entire CA. With this scope, a configuration contains configuration parameters from multiple INI files that affect all COs in the CA. For instance, the configuration for CA

2-SIM would contain all parameters affecting *cSimulator*, *cTrajSim* and all classes inheriting from *cSimSensor*.

Alternative B: One configuration regards a complete INI file. In this case, one *loadParamaters* function is created per INI file to load and verify one configuration with all parameters that can be stored in the file. For instance, since estimation gyroscopes and accelerometers use the same INI file, a single configuration would contain parameters affecting both sensors. Furthermore, one configuration would include all parameters belonging to estimator.ini which is used by several CAs, including CA 0-RUN (*start NavLab*), CA 1-SMR (*selecting a run or mission*), CA 4-LOAD (*load sensor data*) and CA 5-SUM (*create summary report*).

Alternative C: One configuration regards a single class. In this case, one *loadParameters* function is created per class used in a CO, and this function loads and verifies the configuration affecting all code in that class. The *loadParameters* function is thus not concerned with parameters its sub- and superclasses. For instance, one configuration structure would be created for the *cSimSensor* class and a separate one for the *cSimPosm* class, even if a *cSimPosm COs* inherit code from the superclass *cSimSensor*. The same would be true for many other NavLab classes, such as those inheriting from *cKfMeas* and *cSensor*.

Alternative D: One configuration regards a complete CO, and contains (a subset of) the parameters in one INI file that affect the entire CO. Note that a CO may contain code and variables from many different CO classes. As opposed to in Alternative C, only a single configuration is loaded and verified per CO, even if it has code from several super/subclasses. In addition, different COs may have variables and code from the same class, either because they are separate instances from the same class or because they share a superclass. In this case, one configuration is loaded per CO even if they (partly) contain parameters the same class but with potentially differing values.

To adhere to the principle of *separation-of-concerns*, one *loadParameters* function is still created per class, but each one loads and verifies only that part of the configuration affecting the code in its own class. As such, the *loadParameters* functions in all super/subclasses work together to construct the same configuration structure, which eventually describes all parameters affecting the code in one complete CO. For instance, only one configuration would describe all code used to model a sensor. Even if a *cPosm* CO has code and variables from several different classes, including *cPosm* and *cSensor*, only one configuration structure is constructed with all parameters affecting its behavior.

Alternative E: One configuration regards only parts of a CO, and contains (a subset of) the parameters in one INI file that affect only those parts of the CO. This approach resembles Alternative D, but enables a CO to use parameters from multiple INI files. As in Alternative D, the *loadParameters* functions in all super/subclasses contribute to the same configuration structure for a given CO. However, in Alternative

E, a CO may be associated with multiple configuration structures. For instance, one configuration structure would include only those parameters in estimator.ini affecting the *cSummarySensor CO* (configured in CA 5-SUM), while a separate one includes those affecting the *cKalmanFilter* CO (configured in CA 4-LOAD).

5.2.2 Discussion of alternatives

Alternative A does not meet Requirement 6 (separation of concerns). Since a configuration affect code in COs, their developer should maintain their configuration structured. In Alternative A, the *loadParameters* function introduces dependencies between a CA and the COs it uses. In case any of the classes of the COs are modified such that its configuration structure changes, Alternative A requires the modifications to be performed in a completely different location than in the modified CO classes themselves. This can especially be problematic if different developers maintain CO classes for the same CA, as they must then modify the same *loadParameters* function. Furthermore, it introduces redundancy (and thus dependencies) among different *loadParameters* functions: if a CO class used in multiple CAs is modified, all the corresponding *loadParameters* functions must also be updated. As a result, Alternative A introduces an increased risk of code inconsistency, and difficulty determining which configuration structure describes which CO classes.

Alternative B does no satisfy Requirement 4 (support for several COs per INI file). Several COs may use the same INI file and not all COs may need all parameters in an INI file. And since only a subset of COs are used in any given CA, not all parameters in an INI file may be needed in any given run. The *loadParameters* function in Alternative B would not be able to determine which parameters are required for a given run or CA, since it does not have any information about which COs are used. As a result, it would need to include all parameters in the INI file in proactive verification. It is however not reasonable to require the user to provide all parameters in an INI file if the run/CA only needs a subset of those parameters. For instance, CA 5-SUM (create summary report) only requires some of the parameters in estimator.ini, and CA 6-CAL (calibrate DVL) only requires some of the parameters in DVL_pre.ini. Furthermore, Alternative B does not satisfy Requirement 6 for the same reasons as Alternative A.

Alternative C does not satisfy Requirement 3 (support for multiple parameter instances). A CO depends on configuration parameters affecting the code in all of its super/subclasses. If two differently configured COs are instantiated from the same class, they need their own, separate copies of the same configuration parameters, which is not possible if only one configuration is created per class. Similarly, if two CO classes share the same superclass, two COs instantiated from this same CO class may need different values for the parameters they share via the common superclass, which is not possible in Alternative C. For instance, in NavLab the classes modelling sensors for simulation and estimation inherit parameters from a common superclasses for which they may have differing values.

Alternative D does not satisfy Requirement 5 (support for several INI files per CO). This is currently required by cNavEq COs and may be required by COs of other classes in the future.

In Alternative E, all requirements are satisfied by the fact that configurations are represented as separate objects, one of more of which are created and linked with a CO via its *loadParameters* functions. Importantly, it supports the case where CO classes share a common superclass, which is often the case in NavLab 4 (e.g., with sensor classes). Furthermore, it provisions for the possibility of having several configurations per CO.

5.2.3 Selected Alternative

Since **Alternative E** is the only one satisfying all requirements, we define the scope of a configuration according to that in Alternative E, and design our CF data structure accordingly. This approach requires configurations to be modelled as separate entities, such that the *loadParameters* functions for one CO can handle several configurations, each of which creates a configuration containing a subset of the parameters in an INI file. Configurations are represented as objects of the class *cConfiguration* (described below).

During execution of a CA, at least one *cConfiguration* object is created per CO. This is done via the *loadParameters* function of the corresponding CO class, which also verifies and registers the *cConfiguration* objects under unique names in the central configuration registry. For certain classes, only one CO is instantiated at any point in time. In these cases, the name of the CO class can be used as the unique name for the configuration. Currently, this is the case for *cNavLab*, *cPreproc*, *cSimulator*, *cKalmanFilter*, *cSummarySensor, cDvlCalibration, cExportParam and configureExport_app*. For sensor classes, several COs may be instantiated per class. For instance, in runs involving several positioning sensors, several *cPosm* COs are used at the same time. In these cases, a unique name for each sensor's configuration is formed by concatenating its class name with the unique sensor name given in preproc.ini, simulator.ini or estimator.ini.

A single *loadParameters* function is implemented per CO class, but only the one used to instantiate a CO (typically a leaf class) needs to be invoked. The invoked *loadParameters* function creates a *cConfiguration* object that is subsequently passed to all *loadParameters* functions above it in the class hierarchy (explained in detail in Section 6). The *loadParameters* function in each of these classes use the CF API to add to the *cConfiguration* object their own set of parameters, which simultaneously loads and verifies the corresponding parameter values in the user-provided INI file(s).

5.3 Verification Levels

To meet Goal 2 and 3 in Section 4.1 we must be able to verify configurations and present intuitive information to the user in case a configuration fails verification. This section explains what we mean by a configuration failing a verification.

The verification of configurations in the CF is mainly concerned with determining how complete user configurations are. It does this by identifying which elements of a configuration is provided and which are missing, considering how severe the consequences of missing

elements are, and determining whether the configurations satisfy the minimal requirements to execute a CA without errors.

There are many types of issues that may cause the incomplete configurations, some of which have more serious consequences than others. In the CF, the severity of a configuration issue is measured in terms of the verification level at which the configuration fails – the lower the verification level, the more severe the issue. The CF ranks these issues according to the following seven verification levels (Verification Level 1 regards exceptional errors, and is therefore explained below the list):

Verification Level 2: **Unmet configuration dependencies.** A configuration fails at Verification Level 2 if it depends on other configurations or configuration parameters that are not provided by the user. This is commonly the case when the user omitted the parameter specifying location or name of INI file required by another configuration. In this case, there is no way of determining the location of the INI file containing the configuration, and therefore no way to verify or even load of the affected configuration.

Verification Level 3: **Missing INI file.** If the configuration passes Verification Level 2, it is typically possible to find the location and name of the INI file containing the configuration. It may nevertheless be the case that the INI file does not exist, in which case the configuration fails at Verification Level 3.

Verification Level 4: **Missing required configuration parameter.** If the configuration passes Verification Level 3, the user has provided the required INI file. If the user omits a *required* configuration parameter from this configuration, the configuration and the missing configuration parameters fail at Verification Level 4. By default, a configuration that passes Verification Level 4 is considered as having passed proactive verification (this is however configurable via the *preventTaskUpon* parameter, as explained below).

Verification Level 5: **Missing optional configuration parameter.** If all configurations used in a CA passes Verification Level 4, the user should be able to run the CA without errors (given, of course, that the configuration structure properly specifies all required parameters). The user may still have omitted optional parameters with default values. This may have been done intentionally or unintentionally, and a user normally wants to know about the latter. Configurations that lack user-provided values for optional parameters with default values are considered to fail at Verification Level 5, as are the omitted configuration parameters. The user can obtain this information at any time by using the CF to inspect all configurations in a CA. This provides the user with intuitive information about the verification level at which all configurations and parameters used in the CA fails, allowing the user to quickly locate any missing configuration parameters.

Verification Level 6: **Missing optional configuration parameter without a default value.** This means the same as Verification Level 5, but for parameters without a default value. This is arguably a less serious issue than failing at Verification Level 5, since the absence of a default value indicates that the parameter may not be used if not given a value by the user.

Verification Level 7: **No issues.** If a configuration passes Verification Level 6, the CF has not identified any issues with the configuration. Internally in the CF implementation, this is indicated by a configuration "failing" at Verification Level 7, even if the configuration passed every stage of the verification process. Verification Level 7 is included mainly to reduce code complexity.

It is generally the responsibility of the CF to verify the completeness of configurations based solely on its configuration structure. Therefore, the developer normally do not need to manually implement any custom verification code, but instead only needs to specify the configuration structure in the *loadParamters* function. However, certain omissions of required configuration elements may be too difficult to detect and/or too uncommon to justify allocating specific verification code in the CF to detect them. Such omissions may nevertheless yield errors with sufficiently severe consequences to justify the prevention of CA execution upon their detection. For instance, when the INI files for sensors do not state the sensor's class name, their *loadParameters* function cannot even be executed, which prevents the identification of any configuration issues. This is because the *loadParameters* function must be static and as such can only be invoked using the class name. In the CF, such errors are called *exceptional errors*, and although they cannot be detected by the generic verification code in the CF, the developer may wish to implement custom error handling code in the *loadParameters* function that can. Upon an exceptional error, such code can register in the CF data structure that the configuration failed at Verification Level 1, along with information about the error and how to resolve it. This results in the prevention of CA execution and the presentation of the provide information about the error to the user via the CF GUI.

The parameter *preventTaskUpon* in navlab.ini specifies the verification level a configuration needs to pass for it to be considered as verified. If any configuration used in a CA fails at a verification level lower than or equal to this value, the CA is prevented from execution to avoid execution errors. By default, this value is set to 4, i.e., a CA is prevented from execution if any required configuration parameters are missing. To enforce stricter rules on the user-provided configurations, the value of *preventTaskUpon* can be increased to, e.g., force the user to provide values for all optional parameters.

Whenever a CA is prevented from executing, a GUI is presented to the user with intuitive information about the verification level at which the configurations in the CA fails, with additional information about how to fix the issue. The same GUI can be opened upon request at any time via the user inspection facility of the CF (explained in detail in Section 6.4), providing the user with information about all configurations used in a CA.

5.4 Configuration Versions

Up to four different versions of a given configuration can exist at the same time, i.e., those with the states *unverified*, *failed*, *verified,* and *applied*. This section explains the meaning and importance of these states.

An unverified version of a configuration is one that exists only as configuration parameters in an INI file. That is, they have not yet been loaded or subjected to verification by a *loadParameters* function. A configuration that is instantiated as a *cConfiguration* object in a *loadParameters* function, and subsequently subjected to verification by the CF, either obtains the status as a *failed* or *verified* depending on the outcome of the verification. All such configurations are stored in the *configurations* list in the configuration registry, and the lowest verification level at which they fail is stored in the configuration's *failLevel* property. As explained in Section 5.3, all configurations that fail at a level higher than that stored in the *preventTaskUpon parameter* are considered as having passed verification, and as such obtain the corresponding status. Conversely, those that fail at a level below or equal to that indicated in *preventTaskUpon* obtains the state *failed*. An *applied* configuration is one that is not only verified by the CF, but that has also been successfully used to configure its CO. Thus, the applied configuration is one that is actually in use at a given point in time (explained in detail in Section 6.3).

It is important to distinguish between the different versions of the configurations when handling the dependencies between them. In most situations, it is most appropriate for a dependent configuration to obtain the required parameters from an *applied* configuration, simply because this is the one currently in effect. However, in certain situations, one only has access to the verified (thus, not yet applied) version of the required configuration. This is the case when the configuration of an ICO depends on parameters in the configuration of its CO. In this case, the *loadParameters* function of the ICO was invoked as part of the execution of the *loadParameters* function of its CO. This scenario is perhaps most easily understood via an example: The default value of the *h* parameter for simulated sensors is set to the *h* parameter of the simulator. Therefore, the former *h* parameter depends on the latter one. Since sensor classes (which inherit from *cSimSensor*) are ICO classes, their *loadParameters* functions are invoked by the *loadParameters* function of their CO class, which is *cSimulator*. Thus, the verification of the sensors' configuration occurs immediately after the verification of the simulator's configuration, and therefore invariably occurs before the simulator's configuration is applied. Importantly, if the user omits or changes the value of the *h* parameter in simulator.ini, he/she would expect this to be reflected in the default value of the sensors' *h* parameter at the time its *loadParameters* function is called. For this to be the case, the sensors' configurations need to use (depend on) the *verified* version of the simulators configuration, and *not the applied* one.

5.5 The CF Data Structure

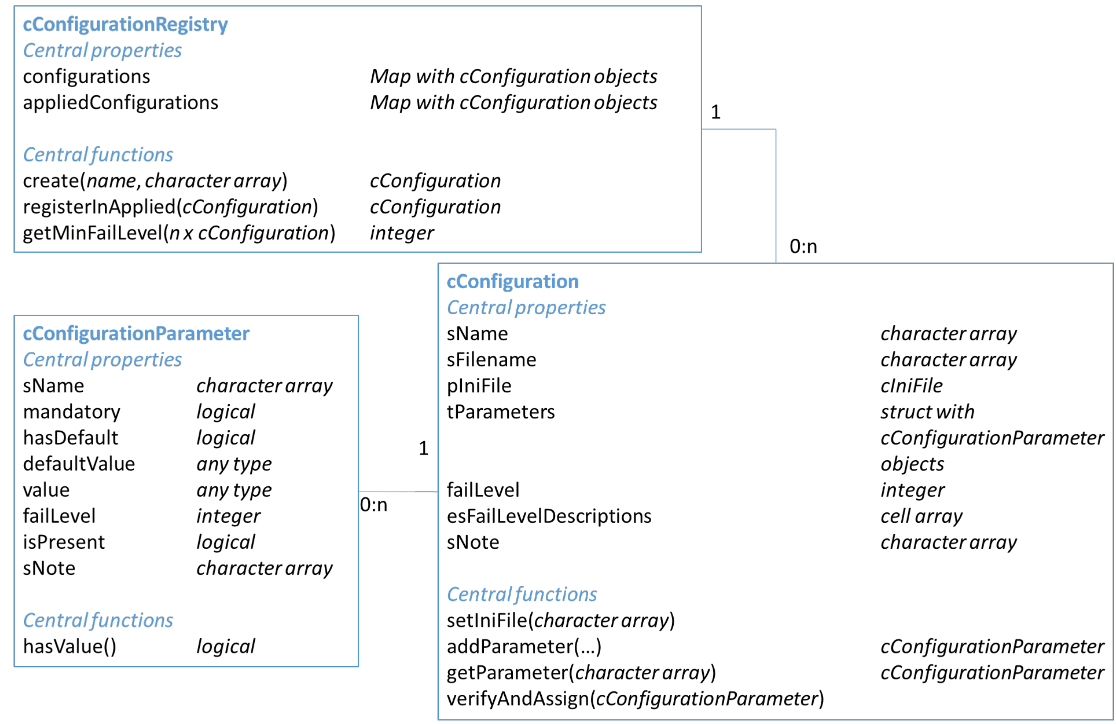


Figure 5.2 Class diagram for the data structure used to represent configurations in the CF.

To facilitate intuitive user inspection and automatic proactive verification, a structured representation of the configuration parameters in NavLab is required. Such a data structure must be sufficiently generic and flexible to meet all the requirements of both current and future NavLab configurations, while at the same time be sufficiently rigid to facilitate uniform, functional and recognizable representations of configurations throughout NavLab. The latter is important to enable automatic verification of the completeness of the configurations, for the developer to quickly understand which information to store about their configurations, and for the user to easily interpret the information when inspecting a configuration.

This section describes the data structure used by the CF to store information about configurations in NavLab 4. Since the NavLab 4 implementation is object oriented, we represent each entity as a MATLAB class. The design of the classes included in the data structure is based on the analysis of the goals, requirements and the chosen definition of a configuration scope.

We have decided to include three classes in this data structure. To adhere to the chosen definition of a configuration scope (Alternative E), a separate class is needed to represent configuration structures, called *cConfiguration*[7]. Configuration parameters are represented as objects of the class *cConfigurationParameter*, and stored in a struct in the *cConfiguration* object to which they belong. Finally, the configurations must be accessible by all code in NavLab 4,

[7] In the naming convention in NavLab 4 class names begin with the character *c*.

including the COs that depend on them. We therefore store handles to all created *cConfiguration* objects in a globally accessible *cConfigurationRegistry* object.

All three classes inherit from the MATLAB *handle* type. Object of these classes are accessible via their *handles*, which function in a similar way as pointers in other object-oriented programming languages. These are described as follows: "*Handle classes define objects that reference the object. Copying an object creates another reference to the same object.*" (The MathWorks, Inc, 1994-2022). By inheriting the handle class, we enable direct manipulation of the configuration registry, configurations and configuration parameters via their handles. This way, modifications can be made to the same object from anywhere in the NavLab code as long as handles to this same object is used.

Figure 5.1 presents an overview of the three classes in the CF data structure, and their most important properties and functions. The names of the properties and functions are shown to the left in each class, while the types of the properties and that of values returned by functions are given in italic font to the right. The arguments to the functions are also given in terms of their types in italic font. Only the central properties and functions are shown to avoid clutter. Most helper functions and properties used only internally by a class are also excluded from the figure to avoid clutter. The details of the CF implementation is obtainable by inspection of the code.

The functions presented in Figure 5.1 constitute the most important part of the CF API used by the NavLab developer. These are used to implement the *loadParameters* function as described in Section 6.2 which makes possible the inspection and proactive verification of configurations. The most central of these are the *create* function in *cConfigurationRegistry* and the *setIniFile* and *addParameter* functions in *cConfiguration*, which work together to (1) specify the configuration structure, (2) load the configuration into memory, and (3) verify the configuration.

5.5.1 cConfigurationRegistry

Figure 5.1 shows two class relations and their cardinalities (number of object instances allowed on each side of the relation). The first relation shows that an object of type *cConfigurationRegistry* can be related to zero or more objects of type *cConfiguration*. This captures the fact that any number of configurations may be stored in the registry, and zero exist before any is added. This relation is realized by storing handles to the *cConfiguration* objects in a MATLAB data structure called *configurations* of type *Map*. Variables of the *Map* type maintain a map between pairs of objects of any type. In this case, it maps the name of a configuration, given as a character array, to a handle to its *cConfiguration* object. By making this map directly readable by other objects (public), and the *cConfigurationRegistry* object globally accessible, all configurations are made available anywhere in NavLab 4 via their unique name in the map.

New, empty configurations are created and added to the *configurations* map using the *create* function in *cConfigurationRegistry*, providing as arguments the name of the configuration and a character array specifying its dependencies. A configuration may depend on either applied or

verified versions of one or more other configurations and/or on specific configuration parameters. Said character array contains one token per dependency formatted as follows: "*configuration name[.parameter][:applied or verified]*". The first part is required. It specifies the name of the configuration the new configuration depends on, or the configuration containing parameters that it depends on. The two subsequent parts in square brackets are optional. If the dependency regards a configuration parameter, the name of the parameter is given after a period sign. Otherwise, the dependency is assumed to regard the configuration as a whole. The final part may specify whether the configuration depends on the applied or verified version of the configuration (or parameter). If an INI file is modified and verified (e.g., via configuration inspection, as explained in Section 6.4.3), but the corresponding configuration is not yet applied, both verified and applied versions of the same configuration exists at the same time. In this scenario, the applied version holds the parameter values currently in use by its CO, while the verified one holds the modified values stored in its INI file. In case the specification of configuration version is omitted from the dependency list, the applied version is assumed.

Unless all dependencies in the dependency list is satisfied, the *create* function sets the *failLevel* property of the newly created configuration to 2, indicating that the configuration fails at Verification Level 2 (unmet dependencies). To avoid this, all configurations or configuration parameters in the dependency marked as verified must in fact exist as verified, and that all those listed as applied must exist as applied. A handle to the newly created *cConfiguration* object is finally returned from the *create* function. Its *setIniFile* and *addParameter* functions are thereafter used to specify the name of its INI file and to add its parameters.

Configurations are placed in the *configuration* list in cConfigurationRegistry upon being subjected to verification, regardless of whether they pass verification or not. Configurations that have not yet been subjected to verification are therefore recognized as such by their absence from this list. Whether a configuration have passed verification is determined by examining the value of its *failLevel* property – if this exceeds the value of the *preventTaskUpon* parameter in navlab.ini, the configuration have passed verification.

It is the responsibility of the developer to store a configuration in the *appliedConfigurations* map after the configuration is in fact applied. This is achieved by the developer by calling the function *registerInApplied* (further explained in Section 6.3). Whether or not a configuration is applied can thereby be determined by whether or not it is stored in the *appliedConfigurations* map.

Further explanation of the use of the *create* and *registerInApplied* functions is provided in Sections 6.2.1 and 6.3, respectively.

5.5.2 cConfiguration and cConfigurationParameter

The second relation in Figure 5.1 shows that *cConfiguration* object can be related to zero or more *cConfigurationParameter* objects. This captures the fact that a configuration can consist of an arbitrary number of configuration parameters, and it has zero parameters before any is added. This is realized via a MATLAB struct called *tParameters* where the value and name of

each field is a handle to a *cConfigurationParameter* object and the name of the parameter, respectively.

A *cConfigurationParameter* object has three additional properties: s*Name* (a character array), s*FileName* (a character array), and *failLevel* (an integer). These hold the name of the configuration, the name of the INI file with its parameter values, and the lowest verification level at which the configuration parameter fails, respectively.

The functions *setIniFile* sets the name of the INI file used by the configuration. If the file exists, it reads contents of this file into a *cIniFile* object and stores this in the *pIniFile* property. If it does not exist, the *failLevel* property is set to 3 to indicate that the configuration fails at Verification Level 3 (missing INI file).

The values for the verification levels are stored in named constants[8] in the *cConfiguration* class where the names indicate the meaning of the verification level. These constants are excluded from Figure 5.1 to avoid clutter. This class also has a corresponding cell array with descriptions of each verification level, called es*FailLevelDescriptions*.

The function *addParameter* is used to add a configuration parameter to the configuration, and simultaneously performs configuration verification. The "…" in Figure 5.1 abbreviates arguments for the parameter's name (type: character array), whether it is mandatory (type: logical), whether it has a default value (type: logical), what this default value is (any type), and a description of the parameter (type: character array). The function creates a *cConfigurationParameter* object with the corresponding property values, and passes a handle to it to the function *verifyAndAssign*.

In *verifyAndAssign*, the *value* property of the newly created *cConfigurationParameter* object is assigned the value of its configuration parameter, provided that the INI file exists and was loaded in the *setIniFile* function, and that the value was provided in the INI file. The *isPresent* property is set to *true* or *false* according to whether or not it was present in the INI file. If not, and the parameter was marked as mandatory, the *failLevel* of the *cConfigurationParameter* object and its *cConfiguration* object is set to 4 to indicate a missing mandatory parameter. If the parameter was not provided, and was optional, its value is set to the provided default value if any, and the *failLevel* is set to 5 or 6 depending on whether or not a default value exists. The *hasValue* function in *cConfigurationParameter* returns *true* if it's parameter's value is provided by the user (i.e., *isPresent* is *true*) or it has a default value (i.e., *hasDefault* is *true*). This function is used, e.g., to determine if dependencies on the parameter are met. If a configuration depends on a parameter, the *hasValue* function of the parameter determines if the dependency is met. Finally, the *getParameter* function in *cConfiguration* takes the name of a configuration parameter as a function argument, and returns a handle to the corresponding *cConfigurationParameter* object (or an empty array if it does not exist).

[8] Named constants are explained in (The MathWorks, Inc, 1994-2022).

6 Applying the CF

This section explains how to add CF support to a CO, i.e., how to make the configurations used by a CO user-inspectable and amenable to proactive verification.

First, Section 6.1 explains how the process of configuring a CO is divided into two stages. Each stage is explained in detail in the two subsequent subsections. Section 6.2 explains Stage 1, which is implemented by the *loadParameters* function, and Section 6.3 explains Stage 2 where the configuration that was loaded and verified in Stage 1 is applied to configure a CO. Section 6.4 explains how to add support to the front-end code. This allows configurations to be inspected by the user via the CF GUI, and is required to invoke proactive verification upon the execution of its CA.

The steps explained in this section are applied to all NavLab code maintained by FFI. This comprises all COs the eight CAs shown in Figure 3.1. In addition, the CF has been applied to four template classes provided for demonstration purposes. These template classes are intended to help developers add CF support in extensions of NavLab.

6.1 The Two Configuration Stages

```
classdef COClass
          # Class properties
          methods (Static = true)
                    function createdConfigurations = loadParameters(configuration)
Section 6.2                   pRegistry = setGetApplInstance().pConfigurationRegistry;
                              # Stage 1, Steps 1.1 to 1.5
                    end
          end
          methods
                    function configureCO()
                              pRegistry = setGetApplInstance().pConfigurationRegistry;
Section 6.3                   # Stage 2, Steps 2.1 to 2.4
                              registry.registerInApplied(sConfigurationName); # Step 2.5
                    end
                    # Other methods, prospectively affected by the configuration
          end
end
```

Figure 6.1 The skeleton of a CO class showing the functions needed to support it by the CF.

Figure 6.1 illustrates the skeleton of a CO class in terms of the functions required to add CF support to the CO. The overall configuration process is divided into two stages, mirroring the two tasks performed during configuration prior to the introduction of the CF (described in Section 3.2). In Stage 1, a configuration is loaded into memory and subjected to verification.

Simultaneously, the corresponding configuration structure is specified. In Stage 2, a verified configuration is used to configure its CO, and Stage 2 is executed only if proactive verification in Stage 1 is passed.

Stage 1 is entirely handled by the *loadParameters* function, the implementation of which is structured according to the five steps described in Section 6.2. By following these five steps, the developer makes sure the *loadParameters* function does what is required to specify the configuration structure and load and verify the user-provided configuration. This function is invoked by the front-end code (described in Section 6.4) whenever the user wishes to inspect the configurations and to perform proactive verification upon CA execution.

The primary focus in this document is on Stage 1, since this is the most important to meet the goals in Section 4.1. The CF is nevertheless also useful in Stage 2 (configuring a CO), and its final step (calling the *registerInApplied* function) is required to inform the CF that the configuration is in fact successfully applied. This enables to distinguish between verified and applied versions of a configuration (explained in Section 5.4). The details of how Stage 2 is implemented are determined by how the CO is configured, which may vary greatly among different classes. Prior to the CF, Stage 2 was also sometimes distributed across multiple functions. Its overall implementation nevertheless typically mirrors that in Stage 1, with the obvious distinction that the purpose in Stage 2 is to apply a configuration, while in Stage 1 the purpose was to load and verify it.

6.2 Stage 1

Stage 1 is implemented by implementing the *loadParameters* function. This function defines one or more configuration structures, and loads and verifies the corresponding configurations.

The *loadParameters* function must be implemented as a static function in order to support proactive verification. This is because the function needs to be executable before any COs are instantiated. The *loadParameters* function normally takes only one function argument named *configuration*. Additional arguments should be avoided whenever possible, as additional arguments may make it impossible to call the function from anywhere in the NavLab code. The reason is that such additional arguments may require information that is available only at certain locations in the NavLab code. In some cases, additional parameters are nevertheless required, examples of which will be given in this section.

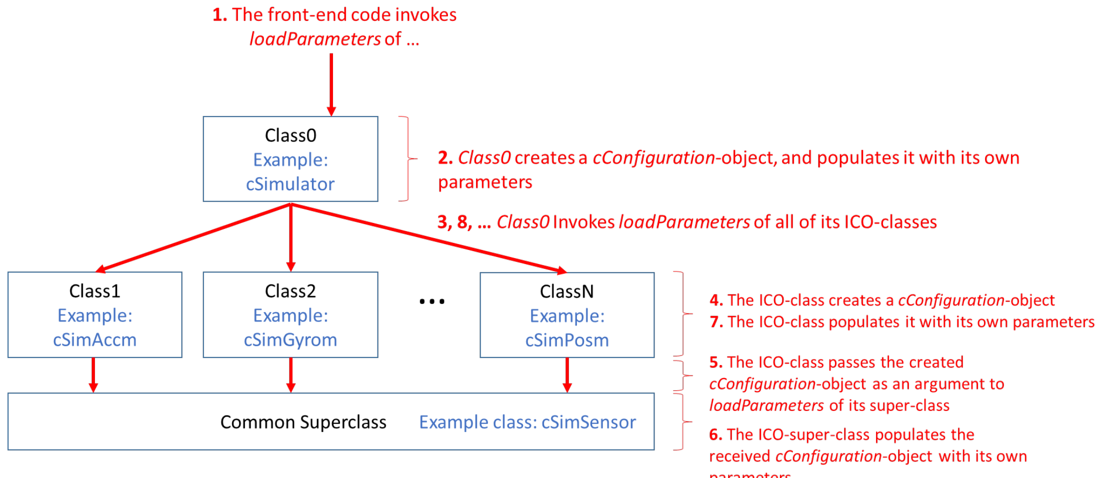


Figure 6.2 How loadParameters is invoked between COs and ICOs, and between sub and superclasses.

As discussed in Sections 3.2 and 5, a CO's configuration holds parameters from the class used to instantiate it in addition to those from all classes above it in the class hierarchy. To adhere to the principle of separation-of-concerns, any of these classes should only handle the configuration parameters that affect the code in that same class. That is, each one implements its own *loadParameters* function. To verify the complete configuration of a CO, it is therefore necessary that all *loadParameters* functions in all super/subclasses contributing to the overall configuration structure to be executed. A given configuration nevertheless regards only a single CO (according to the chosen alternative E described in Section 5.2), which means that all these executions should "collaborate" to create a single *cConfiguration* object.

The process to load and verify the configuration of a CO is illustrated in Figure 6.2. For instance, to the left of the figure we see that *Class1* is a subclass of class *Superclass*. Both of these are used to create an instance of *Class1* CO. We also see that *Class*1 is an ICO class, since its *loadParameters* function is invoked by another CO class, namely *Class0*.

The process to load and verify the configuration of a CO is initiated by executing the *loadParameters* function in the class used to instantiate a CO, which is typically a leaf class. In the example in Figure 6.2, the *loadParameters* function in *Class1* is invoked first from the *loadParameters* function in *Class0*. *Class1* creates a *cConfiguration* object using *cConfigurationRegistry/create*[9], which also verifies whether the configuration passes Verification Levels 2 and 3 and registers it in the configuration registry. Directly afterwards, it passes this to the *loadParameters* function of its superclass, here called *Superclass*. *Superclass* then adds its own configuration parameters to the received *cConfiguration* object. Each parameter is added using *cConfiguration/addParameter*, which *both* adds *and verifies* the parameter, i.e., determines whether the parameter and its configuration passes Verification

[9] We use this MATLAB syntax to refer to member functions in classes, i.e., the term *A/b* refers to the member function *b* in class *A*.

Levels 4, 5 and 6. If *Superclass* had its own superclass, it would pass a handle to the same *cConfiguration* object upwards in the class hierarchy before adding and verifying any of its own configuration parameters. In this way, the newly created *cConfiguration* object initially propagates upwards in the class hierarchy until reaching its root before any configuration parameters are added and verified.

Afterwards, the *cConfiguration* object propagates back down in the class hierarchy via return statements in the *loadParameters* functions, during which the *loadParameters* function of each class adds and verifies the configuration parameters affecting the code in that same class. For instance, after populating the object with its own parameters, *Superclass* returns the object to *Class1,* which in turn populates it with its own configuration parameters. Therefore, at the end of the journey through the class hierarchy, the complete configuration for the CO is both loaded and verified, and exists in the form of a *cConfiguration* object in the global configuration registry. The final step of the *loadParameters* function in *Class1* is to invoke the *loadParameters* functions of all its ICO classes (if any; in this example only Class 0 has ICO classes).

The above implementation can be divided into five discrete steps, each of which are described in the subsections below. The descriptions are accompanied with the example code in figure Figure 6.3, taken from the template class *cSubClass* provided with the CF. To properly demonstrate how everything above is done in practice, the template classes include a subclass (named *cSubClass*), its superclass (*cSuperClass*), and two ICO classes (*cIndirect1* and *cIndirect2*) the configuration of which is invoked from *cSubClass/loadParameters*. According to the explanation above, configuration verification begins by executing the *loadParameters* function in *cSubClass*. Since the implementations of the *loadParameters* functions in the other classes follow the same five-step structure, we only list the code in *cSubClass* for brevity. The functions used in the code snippets and the list below are explained in Section 5.5.

```
1 methods (Static = true)
2
3 function esCreatedConfigurations = loadParameters(pConfiguration)
4
5 % STEP 1.1:
6 if(nargin < 1)
7      pRegistry = setgetAppInstance().pConfigurationRegistry;
8      pConfiguration = pRegistry.create(mfilename('class'), {});
9      esCreatedConfigurations = {mfilename('class')};
10
11     % STEP 1.2:
12     pIniFile = [fileparts(mfilename('fullpath')) '\' mfilename '.ini'];
13     pConfiguration.setIniFile(pIniFile);
14 else
15     esCreatedConfigurations = {};
16 end
17
18 % STEP 1.3:
19 loadParameters@cSuperclass(pConfiguration);
20
21 % STEP 1.4:
22 pConfiguration.addParameter('optionalWithDefault', false, true, 42, ...
23     'Optional parameter with default value 42.');
24 pConfiguration.addParameter('optionalWithoutDefault', false, false, 0, ...
25     'Optional parameter without a default value.');
26 pParam3DefaultValue = pConfiguration.getParameter('superclass_param1');
27
28 if(pParam3DefaultValue.hasValue())
29     pConfiguration.addParameter('dependent', ...
30             false, true, param3DefaultValue.value, ...
31             'Optional, parameter with default value from superclass.');
32 else
33     pConfiguration.addParameter('dependent', true, false, 0, ...
34             ['NOTE: mandatory only because missing default value ' ...
35             'obtained from superclass parameter "required"']);
36 end
37
38 pConfiguration.addParameter('ICOClasses', true, false, 0, ...
39     'List of classes of indirectly configured objects (ICOs).');
40 pConfiguration.addParameter('relativeICOIniFileLocation', true, false, 0, ...
41     ['The folder of the INI files used by the ICOs, relative ' ...
42     'to the folder of the INI file for cSubclass.']);
43
44 % STEP 1.5:
45 pICOParam = pConfiguration.getParameter('ICOClasses');
46 if(isa(pICOParam, 'cConfigurationParameter') && ...
47     pICOParam.hasValue())
48     ICOArray = pICOParam.value;
49     for i=1:numel(ICOArray)
50             ICOClass = ICOArray{i};
51             loadParametersFunction = ...
52                     str2func(strcat(ICOClass, '.loadParameters'));
53             esNewConfig = loadParametersFunction();
54
55             esCreatedConfigurations = {esCreatedConfigurations{:} esNewConfig{:}};
56     end
57 end
58 end % loadParameters
59 end % methods (Static = true)
```

Figure 6.3 Template codfor the loadParameters function in cSubClass.0

The five steps needed to implement a *loadParameters* function are explained in the following subsection. The line numbers mentioned are those in Figure 6.3, which shows the *loadParameters* function of the template class *cSubClass*.

6.2.1 Step 1.1: Create and register the configuration(s) in the global configuration registry

Step 1.1 is primarily achieved via the function *cConfigurationRegistry/create* (Line 8) which means that the configuration registry must first be obtained from the app object (line 7). An explanation of the app object is found in (Kristiansen, 2022). Since *cSubclass* COs only depend on a single INI file, only one configuration is needed. Therefore, the name of the configuration (the first argument of *create*) is set to the name of the class as obtainable via the *mclass* function. As explained in Section 5, this ensures that the configuration is given a unique name whenever only one object is ever instantiated from the class (which is the case for many of the classes in NavLab).

The dependencies of the configuration must be specified as the second argument to the *create* function. As revealed in Line 8, *cSubclass* COs does not have any dependencies. Configurations for the template classes *cIndirect1* and *cIndirect2* do however have such a dependency, i.e., on the parameter *relativeICOIniFileVerified* in *cSubclass*. This is reflected by the arguments used to create their configurations. Below is the corresponding statement in *cIndirect1/loadParameters*:

```
pConfiguration = registry.create(mfilename('class'), ...
      {'cSubclass:verified', ...
      'cSubclass.relativeICOIniFileLocation:verified'});
```

cIndirect1 and *cIndirect2* are ICO classes, and their *loadParameters* functions are invoked from their CO class *cSubclass*. The second function argument shows that the ICO configurations require that the *cSubclass* configuration has been loaded and verified. Note that since the *loadParameters* functions of these ICOs are executed from *cSubclass/loadParamters*, which is executed proactively before any *cSubclass* COs are instantiated, the dependent configurations could not yet have been applied. Hence, the keyword "verified" is used in the dependency list above. As discussed in Section 5.4, the same situation occurs in CAs 2-SIM, 3-PREP and 4-LOAD.

Since the *loadParameters* function of a CO class invokes the *loadParameters* function of all its ICO classes, several configurations may be created as the result of a single execution of a *loadParameters* function. *loadParameters* is normally invoked by front-end code, which requires that a list of all such created configurations are returned from *loadParameters*. The reason is that information about these configurations may need to be displayed to the user. This is either because the user explicitly requested to inspect the configurations or because proactive verification failed (both of which could be why *loadParameters* was called in the first place), and the GUI that presents such information needs to know which configurations to present

information about. Therefore, the name of the configuration created in Step 1.1 must be added to the return value es*CreatedConfigurations*, as shown in Line 9.

6.2.2 Step 1.2: Specify the name of the INI file

Step 1.2 is performed in Lines 12 and 13. First, the name of the INI file is determined. In this case, both the path to the file and the filename is pre-determined. This is however not the case in the *loadParameters* function in *cIndirect1* and *cIndirect2*, the location of which is determined by the parameter *relativeIniFileLocation* in configuration *cSubclass*. This explains the dependencies mentioned in Step 1.1 that configurations *cIndirect1* and *cIndirect2* have on *cSubclass*.

Once the path and name of the INI file is determined, it is assigned to the newly created configuration (Line 13) using *cConfiguration/setIniFile*. If the INI file exists, it is loaded; elsewise *cConfiuration/setIniFile* sets the *failLevel* property of the configuration to 3 (missing INI file).

6.2.3 Step 1.3: Pass the configuration to the superclass

In Line 19, the newly created configuration is passed to the *loadParameters* function of its superclass for it to add its parameters. Only after the superclass has added its parameters, the parameters of *cSubclass* is added in Step 1.4. It is typically a good idea to let superclasses add their configuration parameters before subclasses add their own. This is because the properties of the parameters in a subclass, such as their default values, may depend on the values of parameters in the superclass. An example of this is given in Step 1.4 below.

In *cSuperclass/loadParameters*, all steps except Step 1.4 can be omitted, as it does not need to create a new configuration (it uses the one received from *cSubclass*), and it does not have its own ICO classes (only *cSubclass* does).

6.2.4 Step 1.4: Add configuration parameters relating to this class

In Lines 22 to 42, *cSubclass* adds its own configuration parameters to the newly created configuration with *cConfiguration/addParameter*. The parameters titled *dependent* and *ICOClasses* are particularly noteworthy, the latter of which is discussed under Step 1.5 below. The first, named *dependent*, obtains its default value from the parameter *superclass_param1* which is added in *cSuperclass/loadParameters*. Thus, if a value for *superclass_param1* is not provided by the user, the *dependent* is marked as required. This dependency shares similarities with that in CA 2-SIM between the *cSimulator* configuration and *cSimSensor* configurations. The *h* parameter of *cSimulator* configuration is used as a default value for the *h* parameter of the *cSimSensor* configuration. In case the value for the *h* parameter is not given in the *cSimulator* configuration, that for *cSimSensor* configurations are marked as required. A key difference is however that in CA 2-SIM, the dependent and required parameters belong to **different** configurations, while in the case with *cSubclass* and *cSuperclass* the parameters belong to **the same** configuration. Since the latter is an intra configuration dependency, it is not listed in the

second argument to the *create* function in Step 1.1, which only contains inter-configuration dependencies.

In the presence of intra-configuration dependencies the *loadParameters* function may not always be able to specify in a configuration structure all the parameters that belong to a configuration. This is for instance the case with the intra-configuration dependencies found in NavLab described in Section 3.4. Here, the names of some parameters of simulated sensors are determined by the values of other parameters in the same configurations. Thus, the names of the former parameters cannot be determined in the absence of the latter ones. Since all configuration parameters need names, *loadParameters* cannot add the former parameters to the configuration structure. In such cases, it is important that *loadParameters* does **not** "give up" and return, but rather continues adding all the parameters it can the configuration structure. This is because this configuration structure contains the information displayed to the user inspecting the configuration, and the user should be provided with as much information about the configuration as possible. Importantly, the user should be informed about the missing parameters that are required to meet the above-mentioned intra-configuration dependency.

6.2.5 Step 1.5: Call the *loadParameters* function of all ICOs

Lines 45 to 57 implements Step 1.5 of the *loadParameters* function. This step iterates the list of ICO classes specified by the *ICOClasses* parameter. Each class name in the list is used to construct a handle to the corresponding ICO class' *loadParameters* function, which is thereafter executed. As mentioned under Step 1.1, the names of all created configurations must be included in the list returned from *loadParameters*. This includes the names of configurations created by ICOs, which are added to the list in Line 55.

Lists similar to *ICOClasses* are found in CA 2-SIM, 3-PREP and 4-LOAD, where the simulator, preprocessor and estimator need such lists specifying the sensors to use. Instead of class names, these lists hold the names of the sensors' INI files, which in turn contain the name of the sensor classes. Thus, in these cases the class names of the sensor ICOs must be read from the INI files before their *loadParameters* functions can be invoked.

6.3 Stage 2

In Stage 2 a configuration is applied to a CO. At the time Stage 2 begins, we can assume that the configuration(s) of a CO is loaded and has passed verification in Stage 1. Given that *loadParameters* have correctly specified the structure of the configuration(s), including the locations and names of its INI files, its dependencies and its mandatory parameters, we can assume that a working configuration is available in the global configuration registry at the time Stage 2 begins.

Stage 2 can and should be implemented to meet the unique requirements imposed by the specific CO. For this reason, and because the CF is primarily concerned with the user inspection and verification of configurations, the CF does not (and should not) impose any strict

restrictions for how to implement Stage 2 (although it strongly encourages the implementation of Step 2.5 described below). The descriptions in this section will therefore be much briefer than those of Stage 1 above.

In NavLab, the different CO classes apply configurations to their COs in a very similar manner. These activities can generally be divided into the five steps 2.1 to 2.5. Prior to the introduction of the CF, some NavLab classes performed some of their configuration verification in Stage 2 code. This section describes how this code can be modified to avoid repeating the configuration verification that, after the introduction of the CF, was already performed in Stage 1. Actually implementing these modifications is left for future work (described in Section 7).

In Step 2.1, the c*Configuration* object created in Step 1.1 is used to configure a CO. This object is obtained from the global configuration repository using *cConfigurationRegistry/getVerified*. Note that *getApplied* function *should not be used* at this point, since the primary task of Stage 2 is namely to apply a recently verified version of the configuration to a CO. Only at the end of Stage 2 (in Step 2.5), when the configuration has been successfully applied, will the previously verified version be promoted to an applied version. In the example in the *cSubClass* template class, Step 2.1 is carried out with the following statements:

```
registry = setgetAppInstance().pConfigurationRegistry;

configuration = registry.getVerified(mfilename('class'));
```

In Step 2.2, the obtained configuration is first be passed to functions in the superclass to apply the verified configuration there. By passing the configuration upwards in the hierarchy before applying the configuration locally, the configuration is applied in all superclasses in the same order as they were verified in Stage 1. This way, this process closely mirrors that illustrated in Figure 6.2. In the *cSubClass* template class, Step 2.2 is carried out with the following statement:

```
configureObject@cSuperclass(configuration);
```

In Step 2.3, the configuration is applied locally to the current class using the parameters stored in the obtained *cConfiguration* object. Since the configuration has passed the verification in Stage 1, we can safely assume that all mandatory parameters are available, all parameter dependencies are met, and that missing optional parameters are given their default values if they have one. Optional parameters without default values must though be given special attention as they may be missing. The *cSubClass* template class has parameters of all these types, and implements Step 2.3 with the following statements:

```
o.optionalWithDefault =  configuration.getParameter('optionalWithDefault').value;
o.dependent =  configuration.getParameter('dependent').value;
o.lCOClasses = configuration.getParameter('lCOClasses').value;
o.relativelCOIniFileLocation = configuration.getParameter('relativelCOIniFileLocation').value;
if(configuration.getParameter('optionalWithoutDefault').hasValue())
        o.optionalWithoutDefault =  ...
                configuration.getParameter('optionalWithoutDefault').value;
end
```

In Step 2.4, the configured CO calls the functions in all its ICOs that apply their configurations, each of which performs the five steps in Stage 2 outlined here. In the *cSubClass* template class, the list of its ICO classes is stored in the configuration parameter *ICOClasses* which by virtue of being marked as mandatory is ensured to have a value. Stage 2.4 is implemented using the following statements in *cSubClass*:

```
for i=1:numel(o.ICOClasses)
        % The ICOs are listed in terms of their class names.
        ICOClass = o.ICOClasses{i};

        % Since Stage 2 functions are not static, we can only call
        % these after instantiating the ICOs (which is anyway
        % required for further execution of the activity).
        ICOConstructor = str2func(strcat(ICOClass));
        ICOInstance = ICOConstructor();
        ICOInstance.configureObject()
end
```

After completion of Step 2.4, the *cConfiguration* object used to configure the CO are to be considered as *applied*, which is important to signal to other COs depending on this CO. This is done in Step 2.5 using the function *cConfigurationRegistry/registerInApplied*:

```
registry.registerInApplied(mfilename('class'));
```

6.4 Front-End Integration

Correct implementation of the *loadParameters* function is a prerequisite to enable the inspection and verification of configurations via the CF framework. However, access to these facilities requires mechanisms connecting the GUI to the *loadParameters* functions such that (1) the user can choose to inspect all configurations used in a given CA from the GUI, and (2) all configurations used in a CA is proactively verified upon the execution of said CA. These mechanisms are put in place with minor additions to the front-end code, and the use of the *summaryGUI.mlapp* provided as part of the CF.

6.4.1 summaryGUI.mlapp

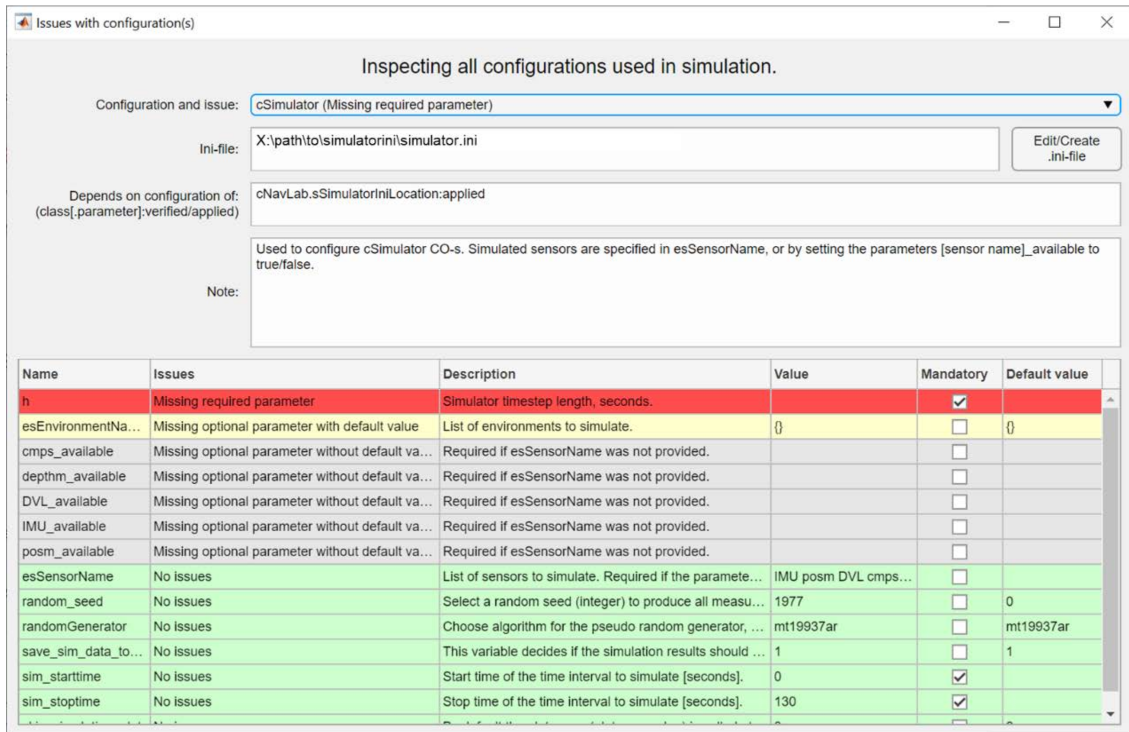


Figure 6.4 Screenshot of summaryGUI.mlapp.

summaryGUI.mlapp is an AppDesigner app with a GUI specifically designed for the purpose of providing an immediate and intuitive overview of a set of configurations – typically all configurations used in a CA. A screenshot of this GUI is shown in Figure 6.4. The GUI is opened by invoking the constructor of the *summaryGUI* class with two function arguments: a cell array specifying the names of the configurations to inspect, and a character array which is displayed in the title bar at the top of the GUI window.

This GUI is primarily designed to be displayed in two circumstances: (1) when the user wishes to inspect the configurations used in a given CA, and (2) whenever a CA is prevented from executing because one or more of its configurations failed proactive verification. In the latter situation, the GUI is displayed to allow the user to quickly identify the missing configuration elements. The names of the configurations given in the first constructor argument are those given to the configurations when they were created with *cConfigurationRegisty/create* (explained in Section 6.2.1). The second argument to the *summaryGUI* constructor is used to inform the user of the circumstance under which the *SummaryGUI* app was opened.

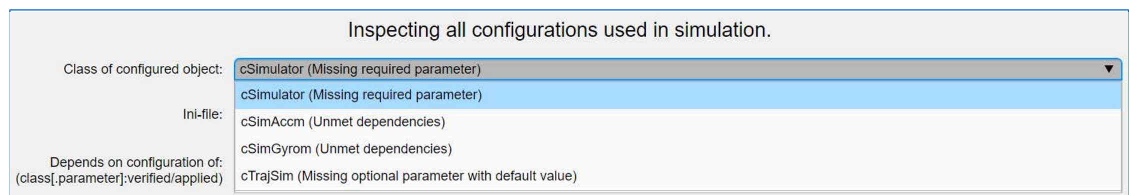


Figure 6.5 The contents of the drop-down lost for CA 2-SIM, showing configurations with issues of varying types.

As can be seen in Figure 6.4, the GUI consists of six overall elements: a drop-down menu at the top, three text fields and a button titled "Edit/Create INI file" in the middle, and a table at the bottom. The drop-down menu at the top has one entry per configuration in the list passed as an argument to the summaryGUI constructor. The text fields and table display information about the currently selected configuration in the drop-down menu. As can be seen in Figure 6.5, each entry in the drop-down menu states the name of the configuration and a textual description of the lowest verification level at which it fails (verification levels are explained in Section 5.3). This verification level at which a configuration fails, and the textual description of this verification level, is obtained from the *failLevel* and es*FailLevelDescritions* properties of the *cConfiguration* class. The three text fields show the INI file used for the configuration, the dependencies of the configuration (as specified when the configuration was created), and a note/description of the configuration (stored in the *note* property of the *cConfiguration* class).

As mentioned above, the GUI is shown either upon explicit user request, or upon one or many configurations failing proactive verification when the user attempts to execute a CA. As seen in Figure 6.5, in these cases the user can quickly identify which configurations fail at which verification levels by simply opening the drop-down menu. Once the configuration is opened, the user can inspect the information in the remaining GUI elements to identify the issues, and press the button labelled "Edit/Create INI file" to edit the corresponding INI file, or create it if it is missing. In the case the configuration has exceptional issues (i.e., fails at Verification Level 1), has unmet dependencies and/or or lacks information about the location of the INI file (i.e., fails at Verification Level 2), or misses the specified INI file (i.e., fails at Verification Level 3), information about this is presented in the respective text boxes.

Any issues with individual configuration parameters are easily identifiable in the table at the bottom. Each row in the table contains information about one of the configuration parameters of the currently chosen configuration. The table has seven columns titled *Name, Issues, Description, Value, Mandatory* and *Default value* and contain the name of the parameter, a textual description of its lowest failed verification level, a description of the parameter (stored in the s*Note* property of *cConfigurationParameter* class), whether the parameter is mandatory or not, and what, if any, its default value is. The rows are sorted first in decreasing order of failed verification level, such that the parameters with most severe issues are presented at the top of the list, and thereafter alphabetically according the parameter name. The background colors reflect the severity of any issues with the configuration parameters. Missing and required parameters are shown with a red background color, to clearly signal to the user which

parameters must be added to ensure a working configuration. Missing parameters with a default value are shown in yellow. This is because the user should be made aware of which parameters affect the behavior of the CA, even if the CA runs without errors. The user may simply have forgotten to include, or was not aware of the presence of such configuration parameters. By clearly highlighting these parameters with a yellow background color, and placing them towards the top of the table, the user is immediately made aware of the parameters and their default values. Missing parameters without default values are shown with a grey background color, suggesting that their omission typically means they have no or little impact on the code. Parameters with no issues, or issues considered to be of low severity, are shown with a green background color.

Once issues are identified, the user can press the button labelled "Edit/Create INI file" to fix them. At the current time of writing, the GUI will not update automatically after modifying the INI file. Thus, to evaluate the effect of the modified INI file, the user needs to close the *summaryGUI* window and re-open it. The GUI will be re-opened whenever explicitly requested by the user or upon failing proactive verification.

6.4.2 Invoking Proactive Verification upon CA Execution

Proactive verification of configurations should occur before the execution of the CA affected by the configurations. A CA is executed upon a GUI request, typically by pressing one of the buttons shown in Figure 3.1. Interaction with any such GUI component invokes its callback function, and such functions constitute the starting point of CAs in NavLab. Therefore, the code to invoke proactive verification must be placed at the beginning of the callback function of the corresponding CA. This is achieved by executing the *loadParameters* functions of all CO classes used in the CA, except for ICO classes whose *loadParameters* functions are executed indirectly via that of their CO class. In case the verification fails, this code should prevent the CA from executing and invoke the s*ummaryGUI* application to inform the user about issues. The user can then fix these issues before re-attempting to execute the CA.

For instance, to achieve the above for CA 2-SIM (performing a simulation) the following code must be added to the beginning of the callback *navlab/pbSimulator_Callback* for the button labelled "Simulator":

```
esConfigurations = cSimulator.loadParameters();
if(app.pConfigurationRegistry.getMinFailLevel(esConfigurations) <= ...
        app.pNavLab.preventTaskUpon)
        pSummaryApp = summaryGUI(esConfigurations, ...
                ['Defect configuration(s) for simulation - aborting task. '...
                'Try again after fixing the configuration(s).']);
        return;
end
% Below this point is code that executes CA 2-SIM
```

In the first line, the static function *cSimulator/loadParameters* is invoked, which results in the verification of the configurations used by the simulator. *cSimulator* also invokes the *loadParameters* function of all its ICO classes, i.e., that of the trajectory simulator (*cTrajSim*) and those of all simulated sensors (inheriting from *cSimSensor*). Therefore, the configurations of all COs involved in CA 2-SIM is proactively verified as a result of the single statement in the first line in the code above. As described in Section 6.2.1, the names of all configurations subjected to verification as a result of a *loadParameters* call is contained in the returned list. Note that this list is not known before executing *loadParameters*, as the COs used in a CA may vary depending on the contents of INI files. This list is passed to *cConfigurationRegistry/getMinFailLevel* to determine whether or not all configurations have passed proactive verification. All configurations have passed verification if the lowest level at which any configuration fails exceeds that specified by the *preventTaskUpon* parameter in navlab.ini. If verification fails, the list of configuration names is passed as an argument to the constructor of *summaryGUI* to inform the user of the critical issues with the configurations, and to let her/him fix the issues before re-attempting CA execution. One example of this is shown in Figure 6.4. Here, the configuration for the *cSimulator* CO fails since the user omitted the required parameter "h" from simulator.ini.

6.4.3 Inspection of Configurations upon User Request

The user may wish to inspect the configurations used in a CA to get an overview of all available configuration parameters, their properties, and their current values as specified in INI files or via default values. *summaryGUI.mlapp* is designed to provide such an overview, not only upon failed proactive verification, but also at any time the user explicitly wishes to inspect the configurations.

The ability to inspect configuration used in a CA can easily be added via (1) a GUI element to request the inspection, and (2) a few lines of code in its callback. The code in the callback will be almost identical to that in Section 6.4.2. First, it must obtain the names of all configurations affecting the CA by invoking the *loadParameters* functions of the CO classes implementing the CA. This list is then passed to the constructor of the summaryGUI class for user inspection.

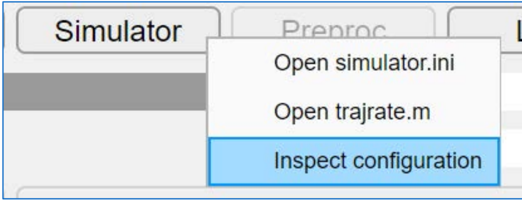


Figure 6.6 The entry added to the context menu of the simulator button to let the user inspect the configurations used in CA 2-SIM (perform a simulation).

In our application of the CF, the context menu was chosen as the type of GUI element allowing the user to request the configuration inspection. One such context menu was added to all GUI elements used to execute a CA. For instance, for CA 2-SIM (performing a simulation), a context

menu was added to the button labelled "Simulator", as shown in Figure Figure 6.6. The callback function of this menu item is as follows:

```
esConfigurations = cSimulator.loadParameters();
if(~isempty(esConfigurations))
    pSummaryApp = summaryGUI(esConfigurations, ...
    'Inspecting all configurations used during simulation.');
end
```

We see that the code is identical to that in Section 6.4.2 with one exception. In the code in Section 6.4.2 it was necessary to determine whether or not proactive verification was passed, to determine whether or not to proceed to execute the CA. This is not necessary when the user only wants to inspect the configurations used in a CA. An additional minor difference is in the text passed as a second argument to the *summaryGUI* constructor to inform about the reason *summaryGUI* is shown.

7 Summary and Possible Future Work

This document presents the NavLab 4 configuration framework (CF), including its requirements, goals, design, implementation and usage.

NavLab 4 draws much if its strength from being highly configurable. This has however also resulted in the configuration of NavLab to become quite complex, requiring the user to handle a large number of parameters and various dependencies between them. Prior to the CF, there was no way to obtain a structured and intuitive overview of available parameters, making NavLab configuration a difficult task even for advanced users.

The CF is designed to reduce the difficulty of creating well-functioning and complete configurations in NavLab. This is achieved by via three core facilities. First, it makes the NavLab configuration easier for the user via a GUI providing structured information about available configuration parameters. This includes information such as parameters' names, descriptions, default values, whether they are mandatory, their dependencies on other parameters, which NavLab objects they configure, and which INI file they belong to. Second, the CF enables automatic and proactive verification of user provided configurations. Whenever the user requests the execution of a NavLab activity, such as simulation, preprocessing or estimation, all configurations affecting that activity is first subjected to verification. If verification fails due to incomplete configurations, the activity is prevented from executing to avoid potentially non-recoverable errors and error messages that may be difficult to understand. Instead, the CF provides the user with intuitive information about which elements are missing and the possibility to add these via a button that opens the relevant INI file. Third, developers are provided with the concepts, API and instructions necessary to provide CF support for their code. Using the CF API, the developer can specify the structure and requirements of their configurations in a global CF configuration registry. This makes their configurations easily understandable by the user, via the above-mentioned GUI, and amenable to proactive verification so as to prevent configuration errors.

CF support has been added to all eight NavLab 4 activities maintained by FFI, including simulation, preprocessing, estimation, DVL calibration and summarizing and exporting analysis results. Therefore, the above mentioned CF facilities are already available to the user in the extended version of NavLab 4. The CF is also designed to make it easy to add CF support to future extensions of NavLab 4.

7.1 Possible Future Work

There are several possible directions for future work. Steps 2.1 to 2.4 in Stage 2 described in Section 6.3 has not yet been implemented for the eight FFI-maintained activities. Therefore, there is still remaining code from prior to the introduction of the CF that performs verification tasks that are now redundant. Since the CF provides a more generic and structured alternative

for many of these tasks, we propose as part of future work to re-write such code according to the instructions in Section 6.3.

Currently, the automatic verification detects only missing configuration elements. We propose to extend the CF with the capability to verify the values of configuration parameters. This can be realized at different levels of sophistication. As a starting point one could simply test whether user-provided values fall within an acceptable range/domain as specified by the developer.

The current CF GUI provides the user with structured information about configurations and a button to open the respective INI files to make adjustments. A possible future extension is to allow the user to modify configurations directly in the CF GUI.

The CF data structure contains information about dependencies between configuration parameters both within and between configurations. The verification mechanisms currently only detects unmet dependencies between configurations. The ability to detect unmet intra-configuration dependencies are left for future work.

Finally, automating the work required by the developer to implement Stage 1 would save time and decrease the probability of programming errors. This would require automatic extraction of information about configuration parameters from MatLab code. Exploring different ways in which this could be achieved, either partly of completely, is proposed as part of future work.

Appendix

A Overview of Configured Code Supported by the CF

Table A.1 presents an overview of all CO classes and INI files that may be used as part of the execution of each CA in NavLab. Whether any given CO class is used depends on the configuration of the particular run. The *loadParameters* function is implemented for all COs listed in this table as part of the application of the CF. Classes that are superclasses of other classes in the table are marked with «(superclass)».

Configuration Activity (CA)	**CO classes**	**Configuration invoked by** (if ICO)	**INI files used**	**Depends on configurations**
CA 0-RUN: Run NavLab	cNavLab		navlab.ini	*None*
CA 1-SMR: Select mission and/or run	cNavLab		navlab.ini	*None*
CA 2-SIM: Simulate	cSimulator		simulator.ini	cNavLab
	ICO classes:			
	cSimAtrm	cSimulator	*User defined*	cSimulator
	cSimAccm	cSimulator	*User defined*	cSimulator
	cSimBVelm	cSimulator	*User defined*	cSimulator
	cSimCmps	cSimulator	*User defined*	cSimulator
	cSimDepthm	cSimulator	*User defined*	cSimulator
	cSimEVelm	cSimulator	*User defined*	cSimulator
	cSimEnvironment	cSimulator	*User defined*	cSimulator
	cSimGyrom	cSimulator	*User defined*	cSimulator
	cSimImu	cSimulator	*User defined*	cSimulator
	cSimPosm	cSimulator	*User defined*	cSimulator
	cSimSimPosmEllipticError	cSimulator	*User defined*	cSimulator
	cSimRange	cSimulator	*User defined*	cSimulator
	cSimSensor *(superclass)*	cSimulator	*User defined*	cSimulator
	cSimSoundSpeed1D	cSimulator	*User defined*	cSimulator

	cSimVelm cTrajSim	cSimulator cSimulator	*User defined* simulator.ini	cSimulator cNavlab
CA 3-PREP: Preprocess	cPreproc		preproc.ini	cNavlab
	ICO classes:			
	cPreAcc	cPreproc	*User defined*	
	cPreAttitude	cPreproc	*User defined*	
	cPreDepth	cPreproc	*User defined*	
	cPreGeneric	cPreproc	*User defined*	
	cPreGps	cPreproc	*User defined*	
	cPreGyro	cPreproc	*User defined*	
	cPreHeading	cPreproc	*User defined*	
	cPreImu	cPreproc	*User defined*	
	cPreImuWithCounter	cPreproc	*User defined*	
	cPreLocalHipap	cPreproc	*User defined*	
	cPrePos	cPreproc	*User defined*	
	cPreUnPressure	cPreproc	*User defined*	
	cPreUsbl	cPreproc	*User defined*	
	cPreVelocity	cPreproc	*User defined*	
	cPreZupt	cPreproc	*User defined*	
	cPreAposClockAhead	cPreproc	*User defined*	
	cPreAttitudeWriteHeading	cPreproc	*User defined*	
	cPreWriteInitialNavigationValues	cPreproc	*User defined*	
CA 4-LOAD: Load data	cKalmanFilter		estimator.ini	cNavLab
	ICO classes:			
	cAtrm	cKalmanFilter	*User defined*	
	cAttitude	cKalmanFilter	*User defined*	
	cBiasedSensor *(superclass)*	cKalmanFilter	*User defined*	
	cCmps	cKalmanFilter	*User defined*	
	cDepthm	cKalmanFilter	*User defined*	
	cDvl	cKalmanFilter	*User defined*	
	cImu	cKalmanFilter	*User defined*	
	cKfAtrM	cKalmanFilter	*User defined*	
	cKfAttitudeM	cKalmanFilter	*User defined*	
	cKfDepthM	cKalmanFilter	*User defined*	
	cKfDvlWaterTrackM	cKalmanFilter	*User defined*	
	cKfHeadM	cKalmanFilter	*User defined*	
	cKfMeas	cKalmanFilter	*User defined*	
	cKfPosM	cKalmanFilter	*User defined*	
	cKfRange	cKalmanFilter	*User defined*	
	cKfVelM *(superclass)*	cKalmanFilter	*User defined*	

	cKfVelRelWaterM cKfZuptM cNavEq cPosm cRange cSeaCurrent cSensor *(superclass)* cStateCorrelation cTraditionalSensor *(superclass)* cTvIntervalSensor *(superclass)* cZupt cSensorWithMeasurement *(superclass)* cSensorWithOutlier *(superclass)*	cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter cKalmanFilter	*User defined* *User defined* *User defined* *cov_matrix.ini* *User defined* *User defined* *User defined* *User defined* *User defined* *User defined* *User defined* *User defined* *User defined* *User defined*	
CA 5-SUM: Create summary report	cSummarySensor *(superclass)*		estimator.ini	cNavLab
CA 6-CAL: Calibrate DVL	cDvlCalibration		DVL_pre.ini	cNavLab cPreVelocity cDvl
CA 7-EXP: Export data	cExportParam configureExport_App		export.ini export.ini	cNavLab cExportParam

Table A.1 An overview of the activities in NavLab, the COs, ICOs, and INI files they may use, and which configurations depend on which others.

References

Gade, K. (2003). *NavLab - Overview and User Guide November 2003 (FFI-Rapport 2003/02128).* Forsvarets Forskningsinstitutt (FFI).

Gade, K. (2004, November). NavLab, a Generic Simulation and Post-processing tool for Navigation. *European Journal of Navigation*.

Kristiansen, S. (2022). NavLab 4: Migrating to App Designer (FFI-notat 22/01154). Forsvarets Forskningsinstitutt (FFI).

The MathWorks, Inc. (1994-2022). *Define Class Properties with Constant Values*. Retrieved from MathWorks: https://se.mathworks.com/help/matlab/matlab_oop/properties-with-constant-values.html

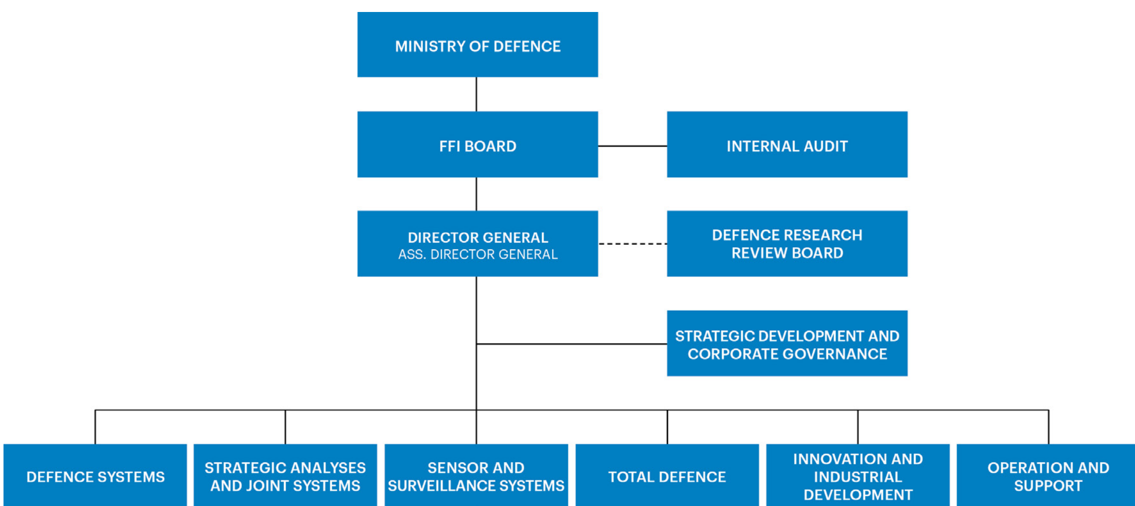
The MathWorks, Inc. (1994-2022). *Handle Classes*. Retrieved from Mathworks: https://se.mathworks.com/help/matlab/handle-classes.html

About FFI
The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's mission
FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's vision
FFI turns knowledge and ideas into an efficient defence.

FFI's characteristics
Creative, daring, broad-minded and responsible.



Forsvarets forskningsinstitutt (FFI)
Postboks 25
2027 Kjeller

Besøksadresse:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telefon: 91 50 30 03
E-post: post@ffi.no
ffi.no

Norwegian Defence Research Establishment (FFI)
PO box 25
NO-2027 Kjeller
NORWAY

Visitor address:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telephone: +47 91 50 30 03
E-mail: post@ffi.no
ffi.no/en