SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Setup of an Embedded System for a Flying Testbed

Benedict Dresel

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Setup of an Embedded System for a Flying Testbed

Aufsetzen eines eingebetteten Systems für einen fliegenden Prüfstand

Author:	Benedict Dresel
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	M. Sc. Tobias Neuhauser
Submission Date:	16.10.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.10.2023 Benedict Dresel

Acknowledgments

I would like to thank the following people:

Thanks to Prof. Dr. Bungartz for supervising this thesis.

Many thanks to Tobias Neuhauser for being my advisor at Airbus, reading through the thesis and making many useful suggestions for improvement; thanks to the whole team at Airbus Defence and Space for an interesting couple of months!

Thanks to Benedikt for reading through the text and making suggestions for improvements.

To Alex for enduring my rants about the MATLAB programming language.

Also, huge thanks to Lily, who read through the whole text and corrected many grammar mistakes!

Abstract

Software integration is a crucial aspect of complex project development, enabling disparate software systems to function seamlessly together. This thesis addresses the challenges of integrating navigational software into middleware for aerospace projects, focusing on achieving a high degree of automation within the MATLAB development environment in terms of a C++ autogeneration workflow.

The complexity of today's technological landscape necessitates the collaboration of specialized teams working on different aspects of a project. Integrating their contributions can be labor-intensive and time-consuming, particularly when dealing with evolving revisions and requirements. To address this, automated integration techniques have gained popularity for enhancing development speed and efficiency.

This thesis presents a (semi-)automated integration process designed for navigational software within an aerospace project, utilizing a proprietary middleware called Si^2.

The target platform for deployment is the DuraCOR 311 Mission Computer, manufactured by Curtiss-Wright, and destined for a flying testbed.

The development process of the navigational software adheres to the V-Model, with all activities centered around an Interface Control Document (ICD) that defines the interface, and hence, the data structures exchanged between the navigational software and its sensors. The integration procedure leverages these ICD-based assumptions to achieve, among others, these goals: A high degree of automation, integration of the workflow within MATLAB, maintainability and adaptability, separation between development and target platforms.

The final result is a user-friendly "drop and forget" solution for the integration process, streamlining the generation of Si^2 projects on the target platform. However, this high level of automation relies on specific assumptions that must hold true and are discussed in the thesis.

Additionally, the thesis presents a simple validation method using a specialized model inside Si^2 to replay test vectors and compare expected values against actual output. Finally, a field test involving real sensors demonstrates the viability of the integration approach in a real-world scenario.

This thesis provides insights into (semi-)automated software integration for aerospace projects, offering a practical solution that balances automation and flexibility in the development process.

Contents

1 Introduction

Software integration is essential for the successful development of larger projects, as it enables separate software (sub-)systems to work harmoniously together. In today's complex technological landscape, no single application can fulfill all the requirements of a comprehensive project – especially not if highly specialized domain knowledge and focus on key functionality are split to different teams with different skill sets.

One key challenge when developing large (software) projects is putting the single pieces together so that they can run in tandem as planned. This can be a cumbersome and time-consuming process, especially if this step has to be performed multiple times in the course of the development process under changing revisions of implementations or even requirements.

This is the reason why techniques like automated integration up to so-called continuous integration are getting more and more popular (see: [Hil+16]). Additional benefits of an integration process which can be performed ad-hoc as needed is that it supports quick validation of the system as a whole. All of this increases development speed and efficiency, taking tedious and repetitive tasks out of the equation, enabling developers and engineers to focus on their actual work.

This thesis describes a (semi-)automated integration process for navigational software in the area of an aerospace project into middleware called Si^2 developed by Airbus Defence and Space GmbH which is then installed and run on a DuraCOR 311 Mission Computer, which, in turn, is going to be deployed to a flying testbed.

1.1 Problem Statement

A (semi-)automated integration procedure for navigational software by Airbus shall be developed which can be triggered ad-hoc by the developers of the mentioned software on the development machines without leaving MATLAB, which is their main development environment. Additionally, a method for validation and a final demonstration test incorporating real sensors on the target platform had to be developed and conducted, respectively.

Now follows a short overview of the development process into which the integration procedure itself is to be integrated, then the requirements, restrictions, and goals are

defined, followed by short introductions of the middleware Si^2 and the DuraCOR 311 target platform. Finally, the results of this thesis are summarized.

The whole project of the navigational software follows the well-known V-Model which is commonly used in larger companies, especially when an interplay between hardware and software is part of the project. For a thorough description of the activities in the V-Model, see [BD04, p. 653f].

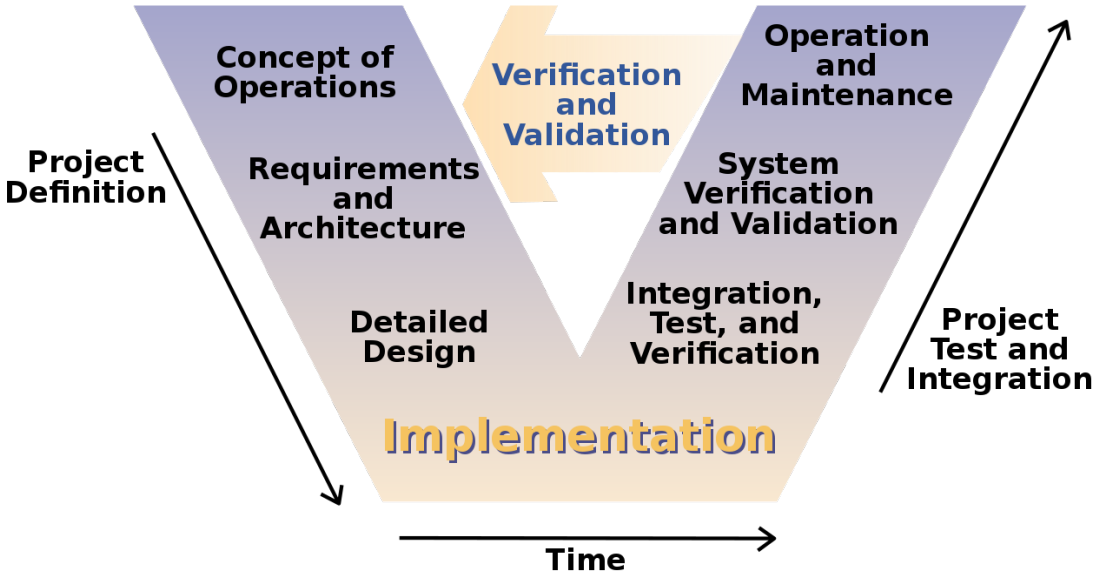

Figure 1.1: This thesis can be located on the "right arm" of the V – it incorporates the integration and validation parts. Source: [Osb+05]

All development activities are centered around an Interface Control Document (ICD): A document serving as the "single source of truth", describing data structures, signals, sensors, and their parameters etc. All artifacts – be they code or other documents – are derived from the ICD and must adhere to the information specified there. This is also called ICD driven development.

This style of development not only benefits the fulfillment of requirements in all parts of the final system, but also an automated integration process since the definitions stemming from the ICD can be assumed to always hold. The procedure described in this thesis makes heavy use of assumptions on the structure of the ICD in order to reach, hold and/or maximize the following goals and restrictions:

- **High Degree of Automation:** The integration procedure shall be executable without much manual labor

- **Integration into the already existing development process:** The integration procedure shall be executable alongside the already existing MATLAB code generation workflow inside MATLAB
- **Maintainability / Adaptability:** If changes to the ICD or the code of the navigational software are made such that the assumptions are violated, and hence, the automated integration process breaks, it shall be possible for third parties to adapt it to the new situation; C++ code has to adhere to the C++11 standard since it has to be supported by different compilers on different platforms (e.g., Microsoft Visual C++ (MSVC), GNU Compiler Collection (GCC) on x86 platforms or others in the future)
- **Separation between development and target platforms:** The integration process shall run on the development machines in order to prevent switching between the former and the target machine. Actions like copying files back and forth between the development and target machines should therefore be minimized; the need for such a separation arises due to constraints by company internal security rules which prohibit the execution of arbitrary code outside of MATLAB on the development machines
- **Correctness**

Some of these goals compete – for example, an increase in the degree of automation usually leads to a decrease in maintainability since more complicated code might be necessary to support the former. Finding a satisfactory balance between these goals is a challenge ever to be passed before and after each design decision. Later on in this thesis, some of these decisions are discussed (e.g. in Subsection 3.2.10 or Chapter 5).

1.2 Si^2

> "The Si^2 simulation framework is a set of software tools that provides a development, integration and execution platform for distributed simulation environments." [23a, p. 4]

In this thesis, only the Si^2 runtime environment is used, which offers a set of services for (potentially) distributed applications in the form of "simulation models". These are abstractions implemented “as shared objects and dynamically loaded by the runtime during [the runtime’s] main initialization” [23b, p. 40] in Ada, C or C++ adhering to one of a few model interface standards like AP2633, EXAP, ARINC653 or FMI2 in order to be loadable by Si^2.

Services offered by the runtime environment to the models include [23a, p. 4]:

- Model Loading
- Communication between different models via messaging channels
- Scheduling
- Logging

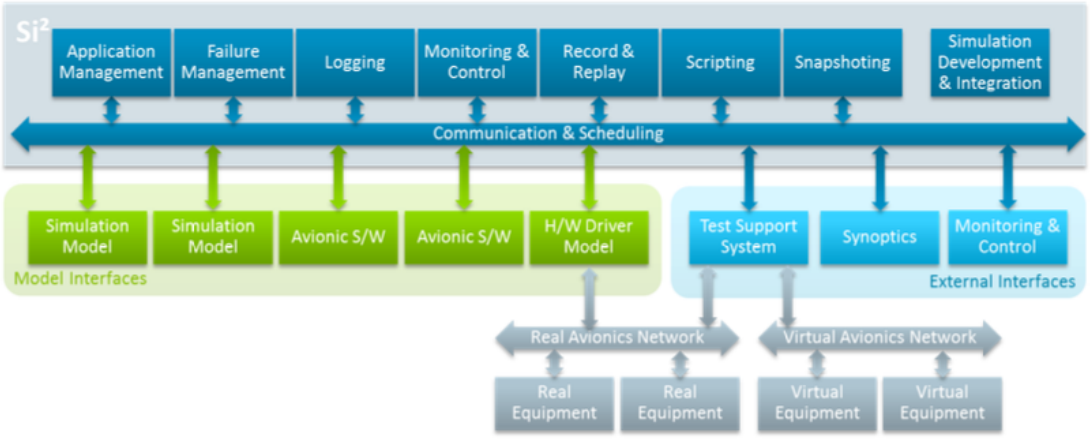

Figure 1.2: Overview of the architecture of Si². Source: [23a, p. 5]

Therefore, by definition, Si² can be understood and used as a middleware[1].

It was decided to use the AP2633 interface standard throughout the whole project. A short definition of such models is provided in Chapter 2. A thorough description of their artifacts follows in Chapter 3.

1.3 DuraCOR 311 Embedded Computer

The DuraCOR 311 tactical mission computer by Curtiss-Wright features a quad-core Intel Atom CPU [20, p. 8] and is used as the target platform for the integration process. An advantage of featuring an Intel CPU is that it provides a standard x86 platform, which, in turn, allows it to run commonly used operating systems such as Ubuntu Linux 20.04.6 LTS and to be targeted by standard compilers such as GCC.

During the integration, necessary additional software such as CMake or GNU Octave and startup scripts running Si² after booting the computer have been set up. As the final step of the integration, a field test has been conducted with real sensors connected to the DuraCOR, gathering real data in order to validate the whole process.

[1] See Chapter 2

1.4 Overview of the Results

The final result of this thesis is a "drop and forget" solution for the integration process: A user can simply copy a set of input files to a directory, call the integration scripts and then copy the resulting complete Si^2 project – containing templates for every sensor, full implementations of the two main models as well as configuration files for each of them – to the target platform. There, the build process is triggered and the project can be run after optionally changing some settings of the models. After that, the main step of the integration is complete and the project is then ready for the (manual) integration of actual sensors by filling the templates of their models with appropriate code.

However, this high degree of automation is only possible if a certain set of assumptions about the input and Si^2's build system hold true – if, for example, the syntax of C struct definitions in the code of the navigational software changes, the integration scripts have to be adapted accordingly[2].

This is also true if the structure of the ICD changes – this requires adaptation of the integration scripts as assumptions on the ICD structure are violated. However, it can be assumed from domain knowledge that this most certainly won't be the case.

Due to this, the integration script supports adding of new sensors, signals, etc. or their removal to or from the ICD.

Later during the development of the integration process, it became clear that the goal of complete separation into code generation on the development machine and just running the project on the target machine is not completely achievable: Parts of the generation of C++ code (specifically: The generation of JavaScript Object Notation (JSON) macros providing conversion functions between JSON and Si^2 data types) depends on header files which are generated by the Si^2 build system which is called on the target machine. This problem could be mitigated without decreasing the amount of automation by leveraging GNU Octave which runs the necessary scripts prior to building the affected models automatically via a custom target in CMake. Other options, possibly providing a cleaner design, are discussed in Chapter 5.

Another key result is a method for validation of the integration process: A special AP2633 model has been implemented which supports replaying test vectors and recording of the input passed to the navigational software as well as its output. The resulting binary file can then be used to compare expected against actual values via a MATLAB script, validating the flow of data.

Finally, a field test has been conducted, integrating real sensors and showing that the approach works for a real scenario.

[2]The reason for this will be discussed in Chapter 3

2 Terms and Definitions

2.1 ICD: Interface Control Document

An ICD defines data structures, their types, signals, and their relationships, and, as a result, the interfaces to and from a given system. It is a document serving as the "single source of truth" inside a project, following a certain structure and letting developers of different subsystems know how to interoperate with each other.

In the case of this thesis, the ICD is an Excel file defining and describing the interface of the navigational software.

2.2 Middleware

A middleware is a software layer running on top of an operating system. It offers various services to applications running on top of it, e.g., scheduling, communication or other services. It serves as a common base for the applications so that they can be developed independent of the underlying operating systems.

Its position in a distributed system, like it is planned for the navigational software, can be seen in Figure 2.1.

2.3 AP2633 Models

The Airbus Procedure 2633 (AP2633) is a model standard which "[...]defines the entry point, control variables, inter-model communication and life cycle"[1]. These models can be in different states during execution: Load, Initialize, Reinitialize, Running, Holding and Unload; each of those requires a corresponding input and result variable.

In Si[2], the model consists of a set of artifacts, which will be described in Chapter 3.

[1] See [23b, p. 41]

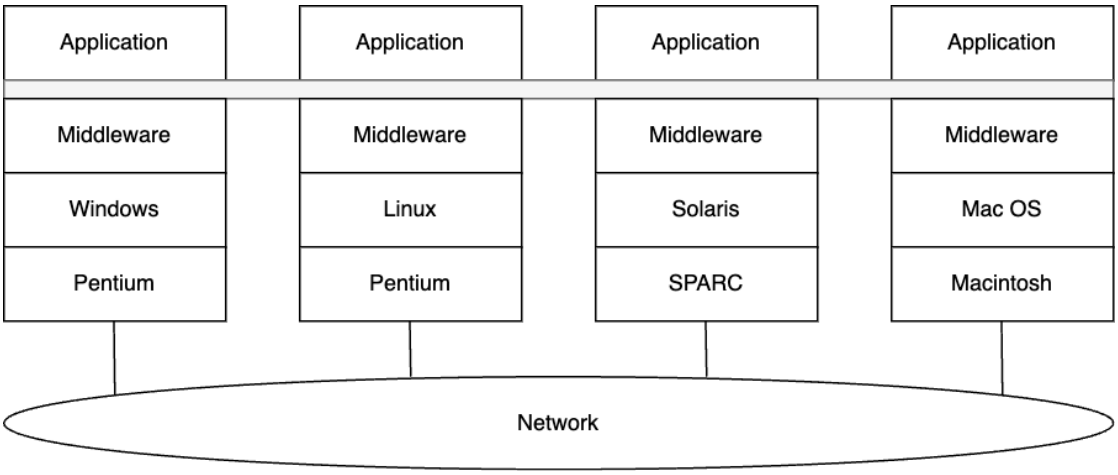

Figure 2.1: Position of a middleware in a distributed system. Adapted and translated from [TB16, p. 685]

3 Approach

First, this chapter gives an overview of the integration process developed during this thesis in Section 3.1 alongside a description of the input files as well as of the flow of events before, during and after executing the integration scripts. In Section 3.2, a detailed description of the developed code as part of the integration process is given.

3.1 Overview of the Integration Process

The integration process can roughly be decomposed into three distinct steps:

1. Based on the ICD, all required files for a Si^2 project are generated.
2. Based on these new files and the input configuration files, C++ code is generated, supporting the conversion between data structure definitions in the navigational software and those deduced from the files based on the ICD from step one (Si^2 communication channels).
3. Si^2's build system generates necessary header files ("model adapter") and compiles the models, which then can be executed inside its runtime environment.

A visual overview of the integration process and the artifacts taking part in and resulting from the integration can be seen in Figure 3.1.

3.1.1 Artifact Overview

Now follows a description of the integration artifacts resulting from the integration scripts. These can be grouped into three categories:

- **Configuration files for Si^2:** Like the manifest files (manifest.mf) and their includes (default.models, ...)
- **Global data descriptions:** Like the Global Simulation Data Description (GSDD) (data.gsdd) or Data Definition (DD) (data.dd) files, which contain data structures usable by all models

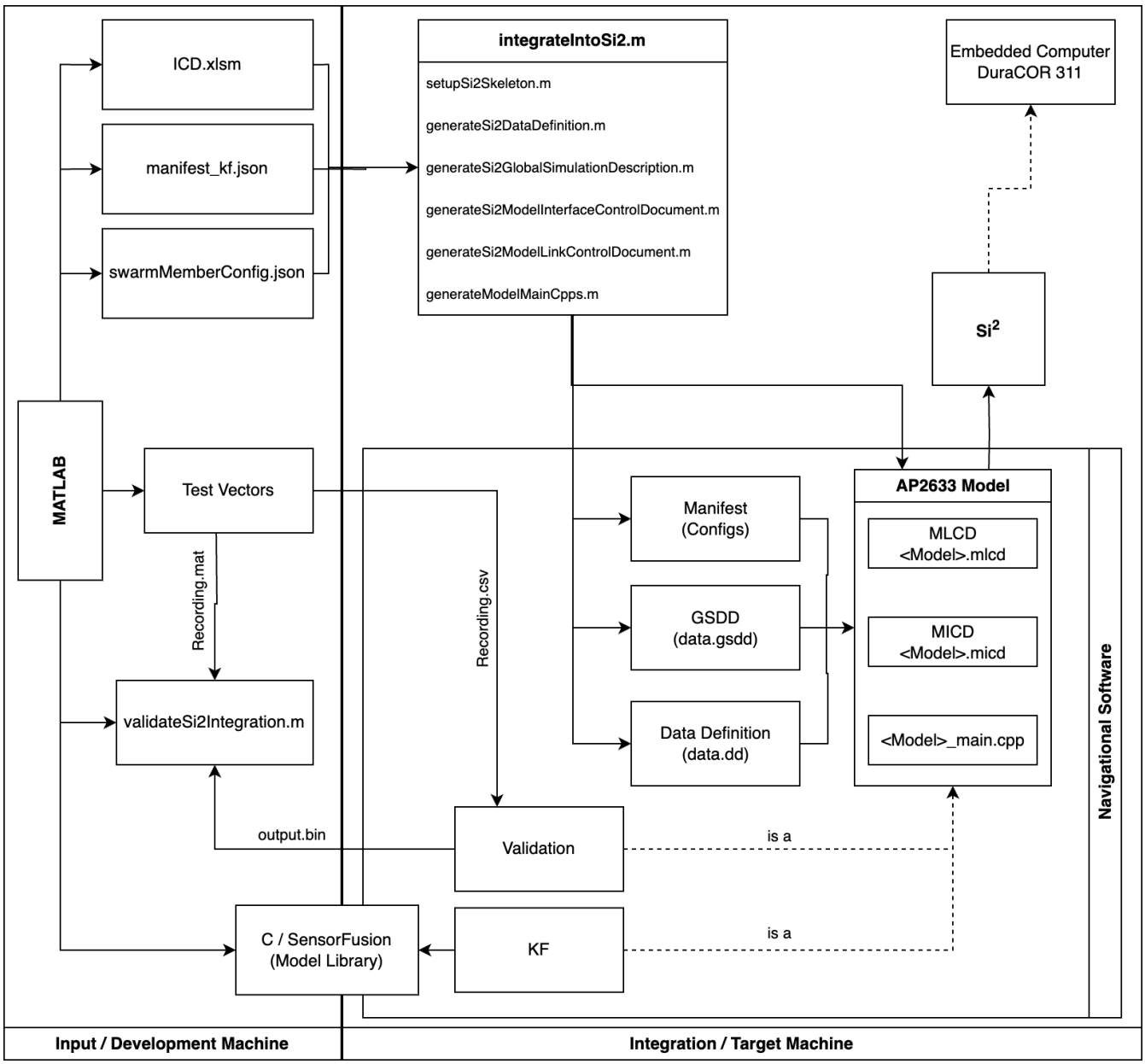

Figure 3.1: Overview of the Integration Artifacts. Note that `integrateIntoSi2.m` is executed on the development machine but belongs to the integration.

- **AP2633 models:** A set of files making up a model for each sensor or data processor (e.g., the model running the navigational software) defined in the ICD

•**Manifest files:** These are Extensible Markup Language (XML) files specifying "which plugins, services and models are loaded within a Si^2 Runtime"[23b, p. 14]. It is also possible to specify preferences for the runtime environment regarding the log level, clock, scheduling, transportation etc.

Two different manifest files are generated by the scripts: `default.mf` which will be loaded if no other manifest file is specified when starting up Si^2 and loads all models apart from the `validation` model through `default.models`. The other one, `validation.mf`, loads only the `validation` and `KF` models via `validation.models`. Its use is described in Chapter 4.

•**The DD:** This is an XML file with the file extension `.dd` and contains data structure definitions. "This[sic] definitions can be reused in a GSDD or an MLCD to avoid multiple, redundant specifications of interface data"[23b, p. 55]. The DD supports different data types, including primitive types like Boolean, Integer, FloatingPoint, String and arrays. Number types are supported in different bit-widths. Multiple data types can be grouped together in the form of StructureElements, which then map to C structs later on.

A sample `data.dd` is shown in Figure 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<DataDefinition xmlns="http://www.airbus.com/si2/icd" name="data">
        <IntegerElement name="id" type="UINT8" value="1"/>
        <StructureElement name="struct">
                <FloatingPointElement name="t" unit="s"/>
                <BooleanElement name="valid" value="true"/>
        </StructureElement>
</DataDefinition>
```

Figure 3.2: Example `data.dd`

•**The GSDD:** The GSDD is an XML file with the file extension `.gsdd`. It contains "the definition of communication channels for data exchange between models and/or devices"[23b, p. 56]. The GSDD supports three different kinds of channels: Signal channels which can reference elements defined in the DD, protocol channels, where the exchanged data is described via a separate protocol and message channels which exchange raw data. In the project generated via the integration scripts, only the first type of channels is used.

A sample `data.gsdd`, referencing elements from a `data.dd`, is shown in Figure 3.3.

•**The AP2633 Models:** For a general description of what an AP2633 model is, see Chapter 2. Here, the artifacts comprising an AP2633 model in Si^2 are explained. Such a

```
<?xml version="1.0" encoding="utf-8"?>
<GlobalSimulationDataDescription xmlns="http://www.airbus.com/si2/icd"
    name="gsdd">
        <Imports>
                <DataDefinition href="data.dd#//@data"/>
        </Imports>
        <Channels>
                <Channel name="struct">
                        <ReferenceElement name="struct" reference="struct"/>
                </Channel>
        </Channels>
</GlobalSimulationDataDescription>
```

Figure 3.3: Example `data.gsdd`

model consists of:

- **The Model Interface Control Document (MICD):** An MICD file is an XML file with the extension `.micd` and contains "[...] the definition of a model component with all its input or output variables or ports"[23b, p. 58]. Following Si^2 runtime manual's description of the DD, it can also reference elements defined there, which is what all MICDs in the generated project do in order to reduce redundancy.

  A sample `model.micd`, referencing elements from a `data.dd`, is shown in Figure 3.4. The output variable specified by the `ReferenceElement` will be mapped to a C variable named `SENSOR1_out` with the type `struct` by the Si^2 build system.

- **The Model Link Control Document (MLCD):** An MLCD file is an XML file with the extension `.mlcd` and contains "[...] the mapping of model inputs and outputs defined in an MICD to channel signals from signal channels or protocol channels defined in the GSDD"[23b, p. 59].

  A sample `model.mlcd`, connecting the output defined in a `model.micd` to a channel defined in the `data.gsdd` can be seen in Figure 3.5.

- **The <Model>_main.cpp:** This C++ file contains the implementation of a model. It includes header files generated by the Si^2 build system based on the other artifacts. Input and output mapped to channels is easily accessible via C variables defined in those header files. For a more thorough description of these header files and of the control variables, states etc., see Chapter 2. An example `IMU_main.cpp` with a

```
<?xml version="1.0" encoding="utf-8"?>
<ModelInterface xmlns="http://www.airbus.com/si2/icd" name="SENSOR1">
        <Imports>
                <DataDefinition href="../data.dd#//@data"/>
        </Imports>
        <Inputs/>
        <Outputs>
                <ReferenceElement name="SENSOR1_out" reference="struct"/>
        </Outputs>
</ModelInterface>
```

Figure 3.4: Example `model.micd`

```
<?xml version="1.0" encoding="utf-8"?>
<ModelLink xmlns="http://www.airbus.com/si2/icd"
    globalSimulationDataDescription="../data.gsdd#
    GlobalSimulationDataDescription" modelInterface="model.micd#SENSOR1"
    name="SENSOR1">
        <Receives/>
        <Sends>
                <Link signal="SENSOR1" variable="SENSOR1_out"/>
        </Sends>
</ModelLink>
```

Figure 3.5: Example `model.mlcd`

full implementation, hooking up a real device providing Inertial Measurement Unit (IMU) and Global Positioning System (GPS) data, is shown in Chapter 4.

Another sample `model_main.cpp`, lacking an integration of a sensor, is shown in Figure 3.6.

- **The CMakeLists.txt:** Each model needs a `CMakeLists.txt` file describing how it should get built by CMake / Si²'s build system. The latter provides a function, `add_ap2633_model()`, which adds a new CMake target based on the provided MICD and source code, generating necessary header files containing data and control variables. If external dependencies have to be built alongside the model, these must be added here. A simple example of an `CMakeLists.txt` is given in Figure 3.7. More complex ones (e.g., for the `KF` or `validation` models) will be shown when they are discussed later in the thesis. Note that, due to the naming of some data structures, the compiler option `-fpermissive` has to be passed to each target so that compilers like GCC accept variable declarations like `FormationSensing FormationSensing;` where the name of a data type is equal to the name of a variable. The MSVC compiler accepts such declarations even if they actually violate the C++ standard.

All artifacts regarding the validation process (the `validation` model, test vectors, `Recording.mat`, `Recording.csv` and `output.bin`) are discussed in Chapter 4.

3.1.2 Description of the Input

Now follows a description of all files serving as the input of the integration process.

ICD.xlsm: This Excel spreadsheet contains abstract definitions about all sensors, data structures, models, and their channels. Each sheet is converted to its own AP2633 model; all data structures are mapped to corresponding Si² data types. All these definitions serve as the "single source of truth" throughout the whole project. A valid `ICD.xlsm` contains a set of sheets which can be categorized into two types: "definition sheets" which must contain the columns `Parameter` and `Value` and define constants and the like – these are currently not used for the integration process but may be in the future. The other category of sheets describes models. These must contain the columns `structure name`, `signal name`, `type`, `dim`, `Unit` and `comment`.

manifest_kf.json: This file contains configuration parameters for the Kalman Filter (KF) model, which serves as the "processing unit" of the navigational software. It is read during runtime by the KF model.

swarmMemberConfig.json: This file contains configuration parameters for the entire system. It describes participating aircraft and their configuration: Which sensors they have installed, etc. Additionally, it tells which sensor signals the KF model expects

```
#include <DATA_C_SENSOR1.h> // input/output variables
#include <CTRL_C_SENSOR1.h> // control variables

extern "C" void SENSOR1_main()
{
	R_LOAD = R_INIT = R_REINIT = R_RUN = R_HOLD = R_UNLOAD = 0;

	if(F_LOAD) {
		R_LOAD = 1;
	}
	if(F_INIT) {
		SENSOR1_out.t = 3.1415;

		R_INIT = 1;
	}
	if(F_REINIT) {
		R_REINIT = 1;
	}
	if(F_RUN) {
		SENSOR1_out.t += 0.1;
		SENSOR1_out.valid = true;

		R_RUN = 1;
	}
	if(F_HOLD) {
		R_HOLD = 1;
	}
	if(F_UNLOAD) {
		R_UNLOAD = 1;
	}
}
```

Figure 3.6: Example `model_main.cpp`

```
add_ap2633_model(SENSOR1
        model_main.cpp
        ICD model.micd
        OUTPUT .
)
target_compile_options(SENSOR1 PRIVATE -fpermissive)
```

Figure 3.7: Example `CMakeLists.txt`

to read. It is used to generate channel assignments to pipe the correct data to the KF model.

SensorFusion4Coder: This directory contains the actual navigational software. It is C++ code generated via MATLAB Coder out of its native MATLAB code. This code will be tightly integrated into and compiled with the KF model which calls its entry point function and writes its output to the output channel of KF. It also contains header files defining all C data structures corresponding to information in the ICD. Conversion functions between these and those generated from Si^2 are necessary and are created during the integration process.

3.2 Description of the Code

In this section, all relevant code for the integration process is explained in detail.

3.2.1 integrateIntoSi2.m

This script is the main entry point to the automated integration process described in this thesis. Its purpose is to read in the input described in Subsection 3.1.2, perform the required generation of artifacts by calling the other scripts and store the resulting Si^2 project in the `output` directory. It also generates additional code required by the used C++ libraries for JSON and Comma-Separated Values (CSV) support which is needed by the `KF` and the `validation` models.

It first declares some constants containing paths as well as `jsonifyStructs` which is an array telling the scripts what data types (structs) to recursively generate `to_json()` and `from_json()` functions for (if they're not automatically targeted later on). These are structs (defined in `SensorFusion4Coder/SensorFusion4Coder_types.h`) the call to the entry point function of the `SensorFusion4Coder` code uses as parameters and contain the C/C++ equivalent of the JSON configuration files from the input. Since the output

of the entry point function (in the form of a `output_T` struct) also needs to be stored later on, it also requires JSON compatibility.

A call to `setupSi2Skeleton()` prepares the output directory structure while creating "static" files which will be the same or similar for each possible input satisfying the requirements imposed on the input, as well as empty files which will be filled by the other functions later on.

Then, it reads in the sheets from `ICD.xlsm` and separates the two categories of sheets mentioned earlier into two maps, `datastructures` and `ms` ("Models"), mapping their name to their tabular data.

Subsequently, it proceeds to calling the other functions in sequence, generating and filling all necessary files for the resulting Si^2 project:

- `generateSi2DataDefinition()` which generates a `data.dd` based on the data structures and models passed to it, returning an XML root node describing the DD, which is reused in the following function calls.
- `generateSi2ModelInterfaceControlDocuments()` which generates `MODELNAME-.micds` for each model.
- `generateSi2GlobalSimulationDataDescription()` which generates the `data-.gsdd` file for the project.
- `generateSi2ModelLinkControlDocuments()` which generates `MODELNAME.mlcds` for each model.
- `generateModelMainCpps()` which generates `MODELNAME_main.cpps` for each model and fills in supporting code (JSON conversions, assignments between channels and internal data types of the integrated navigational software, etc.) for the navigational software. Specifically, it also generates the logic for initializing and calling the navigational software, as well as recording its data and reading in test vectors in the two models `KF` and `validation`.

After all these calls have returned successfully, a fully functional Si^2 project is found in the directory `output`.

3.2.2 setupSi2Skeleton.m

This file contains the `setupSi2Skeleton()` function which is responsible for setting up the directory structure and necessary files for a Si^2 project based on the given input parameters `project_name`, `sensorFusionPath`, `structDefs` (not used now but may be

```
output/
        config/
                manifest_kf.json
                swarmMemberConfig.json
        src/
                MODELNAME/
                        CMakeLists.txt
                        MODELNAME.micd
                        MODELNAME.mlcd
                        MODELNAME_main.cpp
                SensorFusion4Coder/
                        (all SensorFusion4Coder files)
                validation/
                        CMakeLists.txt
                        generateJSON_macros.m
                        generateJSON_macrosForValidation.m
                        parseCStruct.m
                        validation.micd
                        validation.mlcd
                        validation_main.cpp
                util/
                        conversions.h
                        csv.hpp
                        json.hpp
                        jsonStub.h
                CMakeLists.txt
                data.dd
                data.gsdd
        CMakeLists.txt
        default.mf
        default.models
        validation.mf
        validation.models
```

Figure 3.8: Structure of the `output` directory

in the future) and `models`. After a call to this function, `output` should contain the directory structure as seen in Figure 3.8.

First, the function appends `validation` to the models list because it is a separate model not defined in the ICD and therefore not present in the list yet. Then, the function checks whether the `output` directory already exists and, if not, creates it. It proceeds by generating the static (in the sense that they need to be present and depend solely on the names of the models) files `output/CMakeLists.txt`, `output/default.models` and `output/validation.models`. The latter two are (as all XML files) created by leveraging MATLAB's XML support (via the `com.mathworks.xml.XMLUtils` Java interface) in the form of creating and manipulating Document Object Model (DOM) nodes and their attributes, building an XML tree in the process and finally writing them to the files via `xmlwrite()`. An advantage of using this functionality is that these DOM root nodes representing a file can (and will) be reused later on without the need of parsing the XML files themselves again.

The function then proceeds to generate `default.mf` and `validation.mf`, which define the runtime configurations for Si^2 available for this project[1]. Then, it creates the `output/src` directory which will contain all models in the form of subdirectories alongside their source code as well as `data.dd` and `data.gsdd`.

Then, the function creates a `CMakeLists.txt`, `model.micd`, `model.mlcd` and `model_main.cpp` for each model by iterating over the list of models passed to the function. The `CMakeLists.txt` is also filled out with additional information if the current model is `KF` or `validation`, like including the `SensorFusion4Coder` code. The `CMakeLists.txt` of the `validation` model also needs a custom target for executing some MATLAB scripts on the target machine before being built. This is discussed in Chapter 4. All `model.micds`, `model.mlcds` and `model_main.cpps` are filled with content in later function calls.

Finally, the function copies the `SensorFusion4Coder` directory in place as well as some libraries (`csv.hpp` and `json.hpp`) and stubs (`conversions.h` and `jsonStub.h`) alongside the configuration files which also serve as input of the integration process.

3.2.3 generateSi2DataDefinition.m

This file contains the function `generateSi2DataDefinition()` which expects a target path (for the `.dd` file), a map of data structures (`ds`) and a map of models (`ms`) as input and generates a Si^2 DD file as output, returning the XML root node (which is constructed in `ddRootNode`) describing that file. It is responsible for mapping all signals, structures, models etc. from the ICD to suitable representations available in Si^2.

[1] These can be used as parameters for the Si^2 executable.

Again, the XML utilities of MATLAB are used to build the DOM tree by first iterating over each key-value pair in `ds`, which represent data structures not directly stemming from a sensor and which are possibly reused by many sensors. In each iteration, a new node (`curNode`) is created and appended to the root node if it does not exist yet. From each value (which is the table read from the current sheet in the ICD), information like the struct name and all its signals are retrieved. All signals with the same structure name are put together into a single `StructureElement` in Si^2 by creating appropriate child nodes for them, depending on their type in the ICD. This also applies for "substructs" (structures inside a structure) and arrays (if the dimension of the signal is greater than one).

Thereafter, a separator in the form of a comment node is appended to the root node so that the nodes stemming from `ds` and those from `ms` are clearly separated in the resulting file. The latter ones are now generated, again by iterating over key-value pairs (now of `ms`). These represent models, primarily sensors. For each such model, a new `StructureElement` is generated by the name of the current sheet ("`sensorName`").

At this point, mitigating a potential issue within the ICD was necessary: There were two classes of discrepancies in it, redefinitions of structures (if one is defined in `ds` and one in `ms`) or the usage of similar names (e.g., "DataPackage" and "datapackage") which conflict with the current implementation of this function or with the idea of an ICD per se. However, these all had equal definitions inside it, so the mitigation of this problem was to just create a `ReferenceElement` to the already existing `StructureElement` instead of creating another one by the same name which Si^2 would not accept. It should be noted that this approach would break if there would indeed be multiple definitions of structures by the same name allowed. However, there would be no way of automatically deciding which one to use without additional information.

If the current row is not of one of these discrepancies, the function continues by appending the new `StructureElement` node to the root node and adding the appropriate child nodes to it, depending on the data type of the signal. If the current row's `structureName` is not empty, it checks if a node with the same name already exists within the current node. If so, it sets `curSubstructNode` to point to it; otherwise, it creates a new `StructureElement` node with that name and appends it to the current node and the new signal element to `curSubstructNode`. These represent nested structures.

After the last iteration, the generated XML document is finally written to the specified `ddPath` using the `xmlwrite()` function.

3.2.4 generateSi2ModelInterfaceControlDocuments.m

The `generateSi2ModelInterfaceControlDocuments()` function accepts two arguments, `models` (a map containing sheet names as keys and their corresponding tables as

values) and `dataDefinition` (a DOM node representing a DD as returned from `generateSi2DataDefinition()`) and generates an MICD for each model, returning a cell array of DOM nodes representing these MICDs.

From domain knowledge we know the following facts: Every sensor model has exactly one output which will be of the structure type by the name of the sensor as defined in the DD. The model `validation` only contains outputs and no inputs since its sole purpose is to read in test vectors from a CSV file, writing the read data to the channels of the sensors, essentially emulating them[2]. The model `KF` receives all outputs of the sensors as inputs and has one single output.

Keeping this in mind, the function iterates over each key-value pair in `models`. Again, each key represents the name of an Excel sheet corresponding to the name of a model. In each iteration, a new `micdRootNode` is created, which receives an `Imports` node as a child, importing the DD created earlier. Then, empty `Inputs` and `Outputs` child nodes are created and appended to the `micdRootNode`. If the current model is not named `validation`, a `ReferenceElement` by the name of `micdName_out` is created and appended to the `Outputs` node, referencing the structure of the model as defined in the DD. Afterwards, the new MICD DOM node is added to the `micds` cell array and saved to the path of the current model.

After processing all models in that way, the function iterates over the `micds` cell array to add each output of the sensors to `KF`'s `Inputs` node by the name of `micdName_in` and to `validation`'s `Outputs` node. This step establishes the required connections between all models on a semantic level. The "real" links between channels and models are implemented in `generateSi2ModelLinkControlDocuments()` which allows the actual sending and receiving of data via the channels. After modifying the root nodes for `KF` and `validation`, it saves the updated XML documents again.

3.2.5 generateSi2GlobalSimulationDataDescription.m

The `generateSi2GlobalSimulationDataDescription()` function is responsible for generating a GSDD XML document for the Si[2] project. It expects one input argument: `micds`, a cell array of DOM root nodes describing MICDs returned from the call to `generateSi2ModelInterfaceControlDocuments()`. It returns a cell array containing all channels created, which are accumulated in the variable `channels`.

It starts by creating a new XML document root node by the name of `gsRootNode` and appending an `Imports` child node referencing the DD created earlier. Subsequently, it creates a `Channels` child node and starts iterating over each MICD in the `micds` cell array.

[2] See Chapter 4

For each MICD, it extracts the name of the model from its attributes and searches for the `Inputs` and `Outputs` nodes within it. Then, it creates a `Channel` node and sets its name to match the model's name. It proceeds by iterating over each child node of the retrieved `Outputs` node, deep copies them[3] via the function `copyXmlNode()`, removes the "_out" suffix from the name of the element and appends them to the `Channel` node.

Finally, the `Channel` node is appended to the `Channels` node and added to the `channels` cell array. Then, the XML file gets saved.

3.2.6 generateSi2ModelLinkControlDocuments.m

This function creates the MLCD XML documents for each model based on the previously generated GSDD channels. It has two parameters: `models`, as in previous functions, and `gsddChannels`, a cell array of channels from the GSDD.

At first, an empty cell array named `mlcds` is created which is going to hold all the MLCD's root nodes again; additionally, `validation` is added to the cell array of keys again, because it is a separate model not defined in the ICD.

The function then iterates over all keys, creating an MLCD for each model. This is accomplished by first creating a new XML root node with the name of the model, a reference to its MICD and a reference to the GSDD as its attributes. Then, a `Sends` child node is created if the name of the current model is not `validation` since it does not have its own channel but will reuse the channels of all sensors. When such a child node is created, the function proceeds by searching the corresponding `channel` node for the current model inside `gsddChannels`. After that node has been found, the function begins iterating over the child nodes of that `channel`, creating `Link` nodes and appending them to the `Sends` node. Their attributes connect signal names to variable names with the suffix "_out" which are accessible in the model's C++ code later on. Finally, the new MLCD root node is added to `mlcds` and the new MLCD is saved to its location in the directory of the model.

After processing all models in that way, the function starts creating links from all other MLCDs – apart from `KF` itself – to `KF`. For that, it first retrieves the MLCD root node of `KF` from `mlcds` and iterates over all created MLCDs. For each MLCD (except `KF` and `validation` for which the iteration is simply skipped), it retrieves the new `Sends` node, iterates over its children, and creates corresponding `Receives` nodes, linking signal names to variable names with the suffix "_in", which, again, are accessible in the C++ code of `KF` later on. This step connects the output of all sensors as input for `KF`. After that, the updated MLCD root node for `KF` is replacing the old one in `mlcds` and is stored to `KF`'s directory as `KF.mlcd`.

[3] This is necessary because the XML implementation MATLAB uses binds all nodes to the XML document in which they were created initially and cannot be shallow copied to other documents by default

As final post-processing, the function repeats this process for the `validation` model. But instead of creating `Receives` nodes, it deep copies all the `Sends` nodes from the MLCDs of all sensors as well as their child `Link` nodes without changing their names. This is done so that the `validation` model has access to the variables and channels of all sensors and can emulate them later on without the `KF` model knowing. Finally, the old MLCD root node in `mlcds` is replaced by the new one and the new MLCD is stored to `validation`'s directory as `validation.mlcd`.

3.2.7 generateModelMainCpps.m

This function fills the `*_main.cpps` for every model as well as supporting header files like `conversions.h` and `jsonStub.h` by generating appropriate C++ code. Each sensor model's `*_main.cpp` will consist of an empty template for an AP2633 model, which compiles but needs to be implemented manually depending on the physical sensors used (or other data sources). An example for such an implementation is given in Chapter 4. The `KF` and `validation` models consist of a full implementation after a call to this function. Descriptions of their C++ code can be found in Subsection 3.2.15 and Section 4.1, respectively. A high level description of how this C++ code is generated follows.

The function expects three arguments: `models`, which, again, is a map of model names to their corresponding tables in the ICD Excel file, `sensorFusionPath`, a path to the directory containing the code of the navigational software, and `jsonifyStructs`, a cell array / list containing the names of structures for which `from_json()` and `to_json()` functions should be generated[4].

It starts by retrieving the number of sensors used in the current revision of the navigational software by looking up the length of the `SensorList` array contained in `swarmMemberConfig.json`[5]. This number is used as the size of an array called `isNewSensorData` which contains an entry for each sensor indicating whether new sensor data is available to the navigational software ("SensorFusion4Coder").

Then, it defines several type names as constants which the SensorFusion4Coder code exposes for and with its entry point function (`SF_NAV::SensorFusion4Coder()`). They are defined by the developers of the navigational software and may or may not change. Here, their names can be adapted, if needed. These constants are later inserted into the C++ code at appropriate places via calls to `sprintf()`.

After that, the function begins iterating over each name of the models (contained in `ks`) and starts creating C++ code for each one of them by assembling strings containing

[4]These establish compatibility to the used JSON library explained in Subsection 3.2.13

[5]It should be noted that this number could be retrieved more easily by simply using `sizeof(isNewSensorData)`

that code.

When creating the C++ files for the models is done, the function proceeds with a few post-processing steps regarding the automatic generation of JSON macros, conversion functions between Si^2 data types and those defined in the SensorFusion4Coder code which are used to assign sensor data to the equivalent structs representing sensor measurements.

Finally, the function generates the required assignments from CSV data representing test vectors to the appropriate sensor types inside the `validation` model. All these generated functions and stubs are inserted at special comments inside the C++ code serving as markers.

3.2.8 generateJSON_macros.m

This file contains functions responsible for generating macros establishing JSON compatibility for the C++ code to allow serialization and deserialization of data. In order to achieve this, macros are generated which expand to `to_json()` and `from_json()` functions as required by the used JSON library discussed in Subsection 3.2.13.

The first function, `generateJSON_macros()` takes in three parameters: `headerFile`, a path to a C++ header file to search structs in; `struct`, the name of a struct for which JSON macros need to be (recursively) generated; and `lookInStub`, a boolean flag indicating whether to look into `jsonStub.h` for already existing manual definitions of JSON functions for `struct`. This function does not perform the actual work of generating JSON macros but delegates it to a recursive helper function, `generateJSON_macroRec()`.

The second function, `generateJSON_macroRec()`, takes in the same parameters, with an additional fourth one, `acc` which is a string accumulating the resulting macros. It returns a string of the assembled macros. This function first checks via calls to `isJsonFunctionDefined()` if the current `struct` already has manual JSON functions defined in `jsonStub.h` or automatically generated ones in `acc`. If so, it returns early. If not, it parses the current `struct` using the `parseCStruct()` function which extracts information about its members as triples of `[ memberName, memberType, memberDim ]` as well as additional struct types used in the `struct` definition. It then proceeds to assemble the macro call by concatenating the string `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(` with the name of the `struct` followed by the comma-separated names of its members and appends the resulting string to the cell array `ret` which is going to be returned by the function. Finally, the function iterates over the cell array `additionalStructs`, recursively calling itself on each identified struct type, constructing additional macro calls for each of them.

The third function, `isJsonFunctionDefined()` takes in three parameters: `struct`, the name of a struct for which the function searches already existing JSON functions; `acc`,

the accumulator which is being filled by `generateJSON_macroRec()`; and `lookInStub`, a boolean flag indicating whether to search in `jsonStub.h`, too.

It uses regular expressions to seek specific patterns within the `jsonStub.h` file (if `lookInStub` is `true`) as well as in the accumulator `acc`. If it finds such a pattern, the function returns `true`, otherwise it returns `false`.

3.2.9 generateJSON_macrosForValidation.m

In this file, only one function is defined: `generateJSON_macrosForValidation()`. It is specifically used to update `validation_main.cpp` for the `validation` model by generating additional JSON macros for converting test vectors to Si[2] data types and is not called directly from the integration script but is executed during the build process on the target machine (see also Section 4.1).

This function takes in two parameters: `pathToKFDataHeader`, a path to `DATA_C_KF.h` which is generated by Si[2] during the build process of the `KF` model[6]; and `pathToValidationMainCPP`, a path to the `validation_main.cpp` of the `validation` model.

It first reads the contents of the C++ header file specified by `pathToKFDataHeader` and then extracts all struct type names serving as the input for `KF` since these will be the outputs of `validation`. This is done by matching everything between comments `/* input symbols */` and `/* output symbols */` inside the header file via a regular expression and transforming the result such that only a list of struct types remains. These are then passed to the `generateJSON_macros()` function in order to generate the required JSON macros.

Finally, the function inserts these macros at the right place inside the `validation_main.cpp` of the `validation` model, denoted by the comment `/// MARK: autogenerated_macros_si2` and saves the updated file.

3.2.10 parseCStruct.m

This file contains a function, `parseCStruct()` which accepts two arguments: `filePath`, the path to a C/C++ source or header file; and `structName`, the name of a struct of which the members shall be parsed into a list of triples `[ memberName, memberType, memberDim ]` . The function returns these triples as well as a list of custom struct type names encountered during this process (i.e. type names not considered as "basic C types").

It first defines a list of what is considered basic C/C++ data types (like `char`, `int`, `uint8_t` and the like). Thereafter, it checks for possible typedef definitions that could alias the specified struct name to another one. This is done to resolve the "true" name

[6]Which is also the reason why this script needs to be run during build time.

of the struct, so the function can match the actual structure definition. If such a typedef is found via a regular expression, it updates `structName` accordingly.

The function then uses regular expressions to find and extract the definition of the struct with the given `structName` from the provided file. It considers two cases: First, C++ type struct definitions like `struct structName { ... };` , second C type "anonymous" struct definitions like `typedef struct { ... } structName;`. It then makes sure, accidental matches due to a reuse of a struct's name inside another struct, are removed from the results in order to get only the relevant definition.

After that, it prepares the resulting string for tokenization by removing special characters like curly and square brackets and splits the string on each occurrence of a semicolon. Each element remaining is considered a member declaration, which is then tokenized by splitting at spaces such that the type, name, and dimension (if it is an array) of the member remain. If the current member is not an array, and thus has no dimension, a dimension of 0 is added to the triple for consistency. It follows a check whether the type of the current member is found inside `ctypes` and, if not, adds its type to the cell array `structNames`.

Finally, it removes the name of the initial struct from `structNames` and removes any potential duplicates.

There are a couple of facts that should be noted about the parser implemented in this function. First, it only parses a subset of C++, namely definitions of structs generated by MATLAB Coder or Si[2], making the following assumptions:

- The code is valid C++.
- The struct members are POD (Plain Old Data).
- No qualifiers and specifiers (like `__PACKED__`, `static`, `const`, `volatile`, `constexpr` etc.) are used.
- Inside a struct definition, no other aggregate types (like other structs, enums, unions etc.) are defined. To be more precise, no curly brackets must be used inside a structure definition.
- Members are never pointers – with the exception of arrays using "syntactic sugar" syntax like `double arr[5];`.
- No multiline comments are used inside the definitions.
- Inside the parsed source or header file, typedefs must be at most one level deep; id est, there must not be "transitive" typedefs like `struct v; typedef v w; typedef w x;`.

- No structure definitions should be commented out.

This parser breaks if code generation by MATLAB Coder or Si[2] changes such that any of the assumptions no longer hold true. Correct and more generalized parsing requires a way more sophisticated parser; if it should work in all circumstances, a full parser with access to the AST (Abstract Syntax Tree) would be necessary. C and – above all – C++ have extremely complex syntax which cannot be parsed without enormous and time consuming effort which is out of scope of this thesis. The method used in this parser – based on (extended due to the use of lookarounds) regular expressions – is, by definition, not powerful enough to work on arbitrary C++ code. However, due to certain properties of the generated code which is parsed by this function, it should work fine by greatly simplifying the grammar of the language this parser understands.

Finally, a simple grammar of that language in Backus-Naur form could be similar to Figure 3.9, where `NAME` produces all valid names in C++, `CTYPE` produces all basic types in C++ (i.e. int, unsigned int, double, ...) excluding user-defined types, and `DIM` produces a positive integer (including 0).

```
<S> ::= 'struct' <NAME> '{' <MEMBER> '}' ';'
       | 'typedef' 'struct' '{' <MEMBER> '}' <NAME>';'

<MEMBER> ::= <TYPE> <NAME> ';' <MEMBER>
       | <TYPE> <NAME>'['<DIM>']'';' <MEMBER>
       | ϵ

<TYPE> ::= <CTYPE>
       | <NAME> ';'
```

Figure 3.9: A simple grammar producing the language parsed in `parseCStruct.m`

3.2.11 generateStructAssignment.m

This file contains the function `generateStructAssignment()` which generates C++ code converting and assigning Si[2] struct variables to their equivalents from the SensorFusion4Coder code and takes in three parameters: `toName`, the name of the target member within the `allSensorMeasurements` struct of the navigational software to which data is to be assigned; `toType`, the data type of the `toName` member from which the function extracts the appropriate channel name; and `sensorIdx`, an integer representing the index of the sensor inside the `isNewSensorData` array.

The generated code checks whether the time `t` of the sensor is newer than the time `b_time_t` from the `KF` model[7], and, if so, sets the appropriate slot in the `isNewSensorData` array to `true`, then proceeds to convert the data from the channel to its equivalent from the SensorFusion4Coder code and assigns it.

3.2.12 generateStructConversion.m

In `generateStructConversion.m` a function of the same name is defined, accepting three parameters: `sensorFusionPath`, the path to the SensorFusion4Coder directory; `namespace` the name of the namespace used in the latter; and `toType`, the name of the struct type containing the substring `Measurement_T` defined in the SensorFusion4Coder code to which the function generated by `generateStructConversion()` should convert a Si[2] type to.

First, the function calls `parseCStruct()` to parse the struct definition specified by `toType` from the file `SensorFusion4Coder_types.h` at the given path and stores the result in the variable `parsedStruct`. Afterwards it extracts the variable `fromType` from the string `toType` by finding the index of the substring `Measurement_T` within it and taking the part of the string before that index. This works because all structures inside the ICD representing a sensor have a corresponding struct by the same name with the suffix `Measurement_T`.

The function initializes a string in the accumulator `acc` and starts filling it with the code for a C++ function converting `fromType` to `toType`. For that, it iterates through the members of the parsed struct and generates one-to-one assignments to the members of the struct `toType` as they are defined the same way and only differ in naming (they are basically redefinitions of the same structure but a compiler needs such a conversion in that case since it sees both types as different, and thus, cannot be assigned directly to each other. This also has to happen on a per-member level since some of them differ in type, like `boolean_T`, which is actually an `unsigned char` in the case of a SensorFusion4Coder struct and `bool` in the case of a Si[2] struct; both can be converted implicitly, however). If the member is an array, it generates code to copy the contents by calling `std::memcpy()` since assigning an array member to another would simply let both arrays point to the same memory instead of copying the data.

Finally, code for returning a copy of the newly built struct is generated and appended to the accumulator, which is then returned.

[7] This will be explained in Subsection 3.2.15

3.2.13 Includes - JSON and CSV Libraries

This subsection discusses, why JSON and CSV compatibility was needed and how it was implemented.

First, the navigational software / SensorFusion4Coder code requires some initialization which is supplied in the form of `*_.json` files are read during runtime. Second, the `validation` model has to read in test vectors and it was decided to supply those in the form of `*_.csv` files containing rows with cells in JSON format (since JSON is already used for initialization). However, C++ does not natively support JSON nor CSV. For this reason, third party libraries were used with the following requirements: They have to be header-only for ease of integration; they have to be written in plain C++, requiring at most the C++11 standard; they must be licensed under the MIT license (or similar) such that they are free to use within commercial products.

CSV Compatibility. For CSV compatibility, the choice has fallen on Vincent La's CSV Parser [La21] which is able to read in arbitrarily long files with streaming behavior without loading the complete file into memory. The library can also directly convert rows to JSON, which might be useful in the future.

JSON Compatibility. The choice of which JSON library to use fell on Niels Lohmann's "JSON for Modern C++" library [Loh22]. This library not only turns JSON into data types usable just like standard C++ containers, but it also offers an easy way to make user-defined types compatible to its full range of functionality. This is done via macros which provide a good balance between ease of use and the need to parse structs in order to automate the calls of these macros due to the lack of reflection capability in C++11.

For that, macros like `NLOHMANN_DEFINE_TYPE_NON_INTRUSIVE(name, member1, member2, ...)` are available which simply require the name of a struct or class as well as the names of its members as arguments. These macros expand to the required `from_json()` and `to_json()` functions. Working together with the struct parser introduced in Subsection 3.2.10, the automated generation of these macros is possible. The manual creation of these macros would be an error-prone and tedious process since there are numerous structures with many members in the need of JSON compatibility in the SensorFusion4Coder code. Furthermore, all structs generated by Si[2] needed such compatibility as well.

jsonStub.h In order to make the code of the models requiring JSON support more clean, most of the code establishing such compatibility is put into a file called `jsonStub.h`. It contains manual definitions for certain types created by MATLAB Coder (explained shortly) as well as markers after which `generateModelMainCpps()` places the auto-generated JSON macros.

As a representative example for a data type not directly supported by the JSON macros, `emxArray_char_T_1x64` shall serve. This is a struct, generated by MATLAB Coder, representing basic strings / `char` arrays with a maximum length of 64. These are used inside structs like `swarmMemberConfig_T` which represent configuration data the navigational software needs during initialization. For some reason, the developers of MATLAB decided to use such esoteric structs instead of using standard C++ types. However, this means a basic JSON string – as found in, e.g., `swarmMemberConfig.json` – cannot be simply converted via the "standard" `from_json()` function since `emxArray_char_T_1x64` carries semantics the library cannot know about. This implies that the user of the integration scripts has to manually provide implementations of these functions for all MATLAB types relevant to the call to the entry point function of SensorFusion4Coder. For persistency, these functions go into this file. All types encountered during the evaluation steps of this thesis, have already been supplied with such implementations. In Figure 3.10 an example implementation is illustrated.

```
void from_json(const json& j, emxArray_char_T_1x64& emx) {
        auto data = j.template get<std::string>();
        std::copy(data.begin(), data.end(), emx.data);
        emx.size[0] = 1;
        emx.size[1] = 64;
}
```

Figure 3.10: Example `from_json()` function

3.2.14 CMake Build Files

The basic `CMakeLists.txt` has already been explained in Subsection 3.1.1. However, the CMake files for the `KF` and `validation` models require some further explanation because they have additional tasks to perform.

The `CMakeLists.txt` of the `KF` model includes the complete SensorFusion4Coder directory by finding all source and header files inside its folder via the `file()` function, and also adds that directory to the include directories of the target.

The `CMakeLists.txt` of the `validation` model is a bit more elaborate because it needs to call `generateJSON_macrosForValidation()` before building the `validation` target. This is done by adding a custom target via `add_custom_target()` called `generateMacros` which depends on the target `KF` (because the `DATA_C_KF.h` of the `KF` model is read by the `generateJSON_macrosForValidation()` function) so that it is

built after KF. Another dependency of validation on generateMacros is added so that validation is built after that custom target.

Depending on the host platform, the target generateMacros executes either MATLAB (on WIN32) or GNU Octave (on UNIX) running the required functions generating the necessary JSON macros for validation to compile correctly.

3.2.15 The KF Model

Finally, a brief description of both the models for which a full implementation is automatically generated by the integration process is provided. These are the KF and validation models. The latter one is discussed in Section 4.1 to avoid redundancy.

The file KF_main.cpp first includes the necessary libraries, including jsonStub.h discussed earlier. Moreover, variables are declared which are used throughout the lifetime of the program, which is the reason they are marked as static (since they must survive multiple calls of KF_main()). These involve both configuration variables for the KF, swarmMemberConfig and manifest_kf; the inputs to the KF: b_time_t which stores the last time of validity of the IMU sensor / model; isNewSensorData indicating which sensors delivered new data; and allSensorMeasurements which is a structure containing the structs of all sensors representing their measurements. It also declares static variables for the output of KF: output and datapackageOutput, as well as necessary variables for the data recorder[8] (which can be turned on and off via a preprocessor definition). After that, three utility functions are defined: printOutput() which converts the output of the KF to JSON and pretty prints it to the console; loadSwarmMemberConfig() which loads the file swarmMemberConfig.json from disk and parses its JSON content directly to the type swarmMemberConfig_T; and loadManifestKf() which does the same for the file manifest_kf.json.

Thereafter follows the main function of the model, which consists of the different states explained in Section 2.3. In the LOAD state, all variables are first initialized with zeros and the data recorder – if enabled – is set up, i.e., an output file by the name of output_datarecorder_<timestamp>.bin is created. Both JSON configuration files are loaded and parsed using the utility functions described earlier.

The INIT state initializes the data recorder's file, sets b_time_t to 0.0 and calls SensorFusion4Coder(), the entry point function of the navigational software. It should be noted that MATLAB Coder merges the required initialization steps and those for actually running the code into one single function. The first call to it simply initializes everything; further calls to that function then actually run the code[9]. The REINIT state

[8] Which is a class provided with the navigational software.

[9] The code of the original SensorFusion4Coder.cpp is modified in such a way that it returns after initialization in the first call to its entry function; subsequent calls then run the code. This modification

(as well as the `HOLD` and `UNLOAD` states) does nothing as of now.

Finally, the `RUN` state actually executes the model. It does so by first checking whether the sensor channels contain new data (by checking whether the time is more recent) and updates `isNewSensorData` as well as `allSensorMeasurements` accordingly. It then updates `b_time_t` to be equal to the time of validity as provided by the IMU model (since the IMU is acting as the clock of the whole system). A check for `b_time_t` being greater than 0.0 is made to prevent the code of the KF from executing prematurely without any actual sensor data. This is required from domain knowledge. If it exceeds 0.0, the actual call to `SF_NAV::SensorFusion4Coder()` is made, which executes the KF using the new sensor data received from various Si^2 channels. If the data recorder is enabled, it appends a new record for this iteration with the inputs and outputs of the navigational software to the already existing recording file. Before returning, the function resets `isNewSensorData` and `output` to zeros.

is necessary because internal buffers of SensorFusion4Coder would otherwise be initialized with erroneous values and stems from inconsistent behavior in the C++ code generation of MATLAB Coder

4 Evaluation

This chapter describes the two ways how the integration process introduced in this thesis has been validated, indicating that the navigational software is integrated correctly by the developed automated integration process. For one, a special `validation` AP2633 model is implemented which allows the user to check whether the data flow functions properly by replaying test vectors[1] and – in addition – whether MATLAB Coder has generated a working C++ version of the navigational software. The second way of validation is the integration of a real sensor by plugging it into the DuraCOR 311 embedded computer and implementing the corresponding sensor models. After that, a final field test has been conducted using this setup, showing that the integration method indeed works.

4.1 The Validation Model and validateSi2Integration.m

The `validation` model is a special AP2633 model which is loaded into Si^2 by using the `validation.mf` configuration and is automatically generated by the integration process. The `validation` model emulates all the sensors for which the navigational software has been configured (as found in `swarmMemberConfig.json`. Its purpose is to validate the correctness of the data flow through the generated channels for Si^2 by reading in test vectors from a CSV file, piping the data to the various sensor channels which are received and processed by the `KF` model which then writes the output of the navigational software into a binary file.

It follows a short description of the implementation of the model, respectively of the structure of the `validation_main.cpp` file. As with all AP2633 models, the file first includes all necessary header files, including those generated by Si^2. It then declares necessary variables for CSV compatibility using the library mentioned in Subsection 3.2.13 as well as static variables for each sensor output.

The main function sets up the CSV reader and the format of the test vectors in its `LOAD` state. If the model is in the `RUN` state, each call of the main function reads one row of the CSV file. It proceeds to convert and assign the values to the corresponding

[1] These are CSV files with one column for each sensor measurement for a certain time step; each cell contains a JSON encoded structure holding the sensor data as defined in the ICD

channels for each sensor. All other states do nothing.

The user can copy the binary file saved by the KF model mentioned earlier to the development machine, and perform the validation in MATLAB with the help of the function `validateSi2Integration()`.

This function is contained within `validateSi2Integration.m` and takes in three parameters: `refFile`, the path to a reference `.mat` file[2]; `si2recordingFile`, the path to a binary recording generated by the data recorder[3] of the `KF` model; and `stopOnError` which tells the function to abort if a discrepancy between expected and actual recorded values is detected.

First, the reference data is loaded from the `.mat` file via the `load()` function provided by MATLAB as well as the recording data via `binary2matRecording()`, a function provided by the developers of the navigational software. A tolerance threshold is set which defines the maximum allowed difference (due to numerical errors or conversion losses) between reference and values from Si^2 to be considered equal.

The function then proceeds to compare the input data from both the reference and recording files by iterating through the retrieved data structure fields. This is accomplished by making use of the reflection capability of MATLAB using the `eval()` function, putting together the "chain of accessors" throughout the structs, appending their field names retrieved by a call to the function `getFieldNamesRecursive()`[4]. For each field, `validateSi2Integration()` then iterates through the elements of the field and checks if the absolute difference between the reference and recording values lies within the set tolerance threshold. If a mismatch is found, an error message is printed and the boolean flag `hasPassedTest` (which is also the return value of the function) is set to `false`.

After processing the input data in that way, the function performs the same steps on the output data.

Using this function, the user can determine the correctness of the integration process (if no mismatch has been found during the validation of the input[5]), and the correctness of the C++ code of the navigational software generated from native MATLAB code by MATLAB Coder (if no mismatch has been found during the validation of

[2]Which is the recording of a previous test flight, also containing the expected output of the navigational software

[3]This parameter is currently unused due to the implementation of the function `binary2matRecording`, which searches for files containing the string "datarecorder" in their name instead.

[4]Which also has been supplied by the developers of the navigational software.

[5]Which indicates that either the conversion between data types has an issue or the piping of the input data through Si^2 went wrong and therefore the relevant code for model generation or the assignments to the channels need to be corrected

the output).

4.2 The Field Test

The final part of the thesis and its evaluation is a field test – for this, a real sensor, the SBG Systems ELLIPSE2-N is plugged into the DuraCOR 311 via USB and integrated into an AP2633 model via the manufacturer's C library based on example code supplied with it [22b]; the example was extended to also read out acceleration data from the IMU and GPS data regarding position and velocity. All information regarding the sensor and library protocol is found in its firmware manual [22a].

Due to the SBG Ellipse internally having an IMU as well as a Global Navigation Satellite System (GNSS) receiver which is reflected also by the library and, respectively the sensor protocol, the IMU and GNSS AP2633 models had to be merged – this just meant copying the entries of the MICD and MLCD of the GNSS model to those of the IMU model and removing the GNSS model's entry in `default.models`.

A photo of the final setup is shown in Figure 4.1.

Figure 4.1: The DuraCOR 311 with the Ellipse sensor and a GPS antenna plugged in.

A short overview of a `IMU_main.cpp` integrating the Ellipse sensor: After including the necessary header files, the path to the USB device the sensor is bound to by Linux, as well as the used baud rate are defined in preprocessor definitions. A couple of static variables regarding the sensor library and holding the data for error codes, an interface handle and a communication handle in memory are declared.

After that, a callback function, `onLogReceived()`, is defined, which is called every time a new data frame is available from the sensor. First, this function checks the message type of the frame according to the sensor protocol and assigns the read values to the correct Si[2] channels depending on the message type (in our case these may be either IMU or GPS data) after converting them to the correct units.

The `IMU_main()` function is straight-forward. In its `INIT` state, it establishes a serial connection to the sensor device via appropriate calls to the sensor library, configuring the refresh rate to 50 Hz for the IMU and 5 Hz for the GPS. It also hooks up the callback function described earlier. In its `RUN` state, the AP2633 model tries to read a new data frame from the sensor and simply sleeps if no frame is available. The `UNLOAD` state deinitializes the sensor's library. All other states do nothing.

Finally, this setup of the embedded computer, the sensor device with the IMU including the GNSS receiver as well as an GNSS antenna and power supply, has been installed in a car[6] in order to gather real world data of a dynamic trajectory. The IMU has been mounted on the car roof with duct tape, the GPS antenna had an integrated magnet for attaching it on the car roof.

The field test has shown that the developed integration method works for a real scenario using physical sensors and may be used as a starting point for adding additional physical sensors in the future. It should be noted that no manual changes to the KF model have been necessary; the auto-generated AP2633 model worked as intended.

[6] Due to time constraints, the initially planned hexacopter flight could not be conducted

5 Outlook

There are a couple of improvements and additions the integration process introduced in this thesis could benefit from.

Regarding the generation of the C++ JSON macros, leveraging a complete compiler with access to the Abstract Syntax Tree (AST) or `libclang` would enable the use of arbitrary valid syntax for the structure definitions inside `SensorFusion4Coder_types.h` and other header files parsed during the integration process instead of relying on certain characteristica of the code generated by MATLAB Coder and Si[2]. This approach also mitigates certain threats to the validity of the process: If the syntax of the generated structure definitions changes, the parser has to be adopted accordingly. However, using `libclang` or a similar solution introduces many dependencies which must be installed on the target platform, increasing the complexity of the build process. Also, using such a solution is a lot more complex compared to the parser introduced in this thesis.

Moreover, if at some point in the future, the restriction on using C++11 will drop, one could make use of the reflection capability of C++17 or newer which allows a kind of metaprogramming, e.g. iterating over the members of a struct at compile time which makes the generation of the JSON macros easier if even necessary.

Another possible improvement regarding the generation of the C++ main files of the models is to separate the C++ code from the MATLAB scripts; right now, they are built "insitu" inside large strings; the static parts of the code could literally be outsourced to their own files containing placeholders which the scripts then replace and/or fill. A small downside which comes with this approach is that changes to these static parts – possibly – require another editor besides MATLAB itself. It is up to the developers what solution is more beneficial to them.

A significant, however time-consuming, improvement is to implement unit tests for the integration scripts. This decreases risks of breaking the integration process if changes to the scripts are made since any error can be quickly identified by running the test suite.

6 Conclusion

In this thesis an automated integration process for navigational software into the middleware Si^2 has been developed and introduced, pathing the way of putting the navigational software onto an embedded computer and exposing it to the real world.

Looking back, such an endeavor requires understanding and knowledge of the whole system; a lot of domain knowledge has been obtained by communicating with the team responsible for the development of the navigational software. It also became clear that there is no such thing as a "blueprint" for integration processes like this: Each one has to be tailored to the specific needs and requirements by the developers as well as the development and target systems.

However, certain patterns (like the need for parsing code and generating new code out of it) emerged which could be promising targets for future research, possibly also increasing the efficiency of developing such integration processes themselves.

Abbreviations

AST Abstract Syntax Tree

CSV Comma-Separated Values

DD Data Definition

DOM Document Object Model

GCC GNU Compiler Collection

GNSS Global Navigation Satellite System

GPS Global Positioning System

GSDD Global Simulation Data Description

ICD Interface Control Document

IMU Inertial Measurement Unit

JSON JavaScript Object Notation

KF Kalman Filter

MICD Model Interface Control Document

MLCD Model Link Control Document

MSVC Microsoft Visual C++

XML Extensible Markup Language

List of Figures

Bibliography

[20] *DuraCOR 311 User Manual*. Available at https://www.curtisswrightds.com/sites/default/files/2021-12/Manual-DuraCOR-311.pdf. Curtiss-Wright. 2020.

[22a] *Ellipse, Ekinox & Apogee Firmware Manual*. Available at https://support.sbg-systems.com/sc/dev/latest/firmware-documentation, version SBGFWM.2.3. SBG Systems. 2022.

[22b] *sbgECom Library*. Version 3.2.4011-stable. Available at https://github.com/SBG-Systems/sbgECom. Nov. 2022.

[23a] *Si2 Simulation Framework Extension Manual*. Airbus Defence and Space GmbH. 2023.

[23b] *Si2 Simulation Framework User Manual*. Airbus Defence and Space GmbH. 2023.

[BD04] B. Brügge and A. Dutoit. *Objektorientierte Softwaretechnik mit Entwurfsmustern, UML und Java*. 2nd ed. Pearson Studium, 2004.

[Hil+16] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. "Usage, costs, and benefits of continuous integration in open-source projects." In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, pp. 426–437.

[La21] V. La. *Vince's CSV Parser*. Version 2.1.3. Available at https://github.com/vincentlaucsb/csv-parser. July 2021.

[Loh22] N. Lohmann. *JSON for Modern C++*. Version 3.11.2. Available at https://github.com/nlohmann/json. Aug. 2022.

[Osb+05] L. Osborne, J. Brummond, R. Hart, M. Zarean, and S. Conger. *Systems Engineering Process*. 2005. URL: https://commons.wikimedia.org/wiki/File:Systems_Engineering_Process_II.svg.

[TB16] A. S. Tanenbaum and H. Bos. *Moderne Betriebssysteme*. 4th ed. Pearson Deutschland, 2016.