DAISI

Driving Assistance Interface for Simulated Interventions

Developer Guide

Universiteit Utrecht

Contents

Chapter 1

Introduction

1.1 Requirements

This section lists all of the requirements needed to develop and work on our packages. The assumption is that the reader already has a C++ and CMake capable IDE installed like Visual Studio, Visual Studio Code or CLion.

1.1.1 Compiler and build tools

For DBeaver the last version of Java and Maven is required, as DBeaver is a Java application that is build with Maven. Furthemore you will need an IDE for Java, like Eclipse or IDEA. For Eclipse follow this guide `https://github.com/dbeaver/dbeaver/wiki/Develop-in-Eclipse`.

Windows

For Windows we suggest using the MSVC compiler, which should be shipped with Visual Studio (if installed). The project was compiled with the MSVC v143 with specific build numbers 14.30, 14.31 and 14.32 versions of the compiler and should (in theory) compile with later versions of the compiler as well. If needed, Visual Studio can install older versions of the C++ compiler by modifying the Visual Studio install and selecting a specific version under `installation details -> Desktop development with C++ -> MSVC v142` (or other version). Furthermore we suggest having the ninja build tool installed, which also ships with Visual Studio, as this should be the fastest build tool with CMake for Windows.

To be able to run code coverage over the project, OpenCPPCoverage should be installed.

Linux

For Linux we suggest using the GCC-11 compiler. Normally GCC is already installed in most Linux distros, but the compiler may be of an older version. You can check the GCC version by typing `gcc --version`. If you want to install a newer version run the following commands `sudo apt update`, `sudo apt install gcc-11 g++-11`, `sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-11 110 --slave /usr/bin/g++ g++ /usr/bin/g++-11 --slave /usr/bin/gcov gcov /usr/bin/gcov-11`. Be careful with copy-pasting these commands, newlines might be copied, while these should not be pasted in the command.

You can install CMake by downloading it from the Ubuntu package store, or by using snap with the following command: `sudo snap install cmake --classic`. Furthermore to install ninja, use `sudo apt install ninja-build`.

GCOVR can also be installed for code coverage on Linux. This can be done by using the python installer pip `sudo pip install gcovr`.

1.1.2 IPCLib

Windows

No additional packages are needed to build IPCLib.

Linux

To build IPCLib you need to have PThread installed, which should be shipped with the pre-installed version of the GCC compiler.

1.1.3 DAISI and SDALib

For DAISI and SDALib you need the same packages as IPCLib. Furthermore you need Boost (version 1.79 or later). The libraries 'libcrypto' and 'libssl' are also required for MySQL. These libraries are usually pre installed on Windows and Linux, but if they are not installed you can easily download the dlls or an installer from the internet.

Windows

For Windows we created a libraries folder which already contains all of the additional dlls needed to build the project.

Linux

You need to have IPCLib installed. This can be done manually by installing from the source code, or (for debian) can be done by downloading and installing the debian package from the releases tab of IPCLib. Then you need to install the following packages (for Ubuntu use `sudo apt-get install` {`package name`}):

- libplib-dev
- libexpat1-dev
- libopenscenegraph-dev
- freeglut3-dev
- libvorbis-dev
- libsdl2-dev
- libopenal-dev
- libenet-dev
- libjpeg-dev
- libpng-dev
- libcurl4-openssl-dev
- libmysqlcppconn-dev
- libmsgpack-dev

After that both packages can be build correctly.

1.2 Program Structure

1.2.1 Introduction

DAISI has two main modules: the front-end and back-end modules. Besides these modules, the system uses two libraries made by the developer team. These libraries are named SDALib and IPCLib. In Figure 1.1 a simplified version of the program structure is displayed. Furthermore, Appendix A includes extensive visual representations of the program structures. These include all classes, functions, and variables of DAISI, SDALib, and IPCLib.

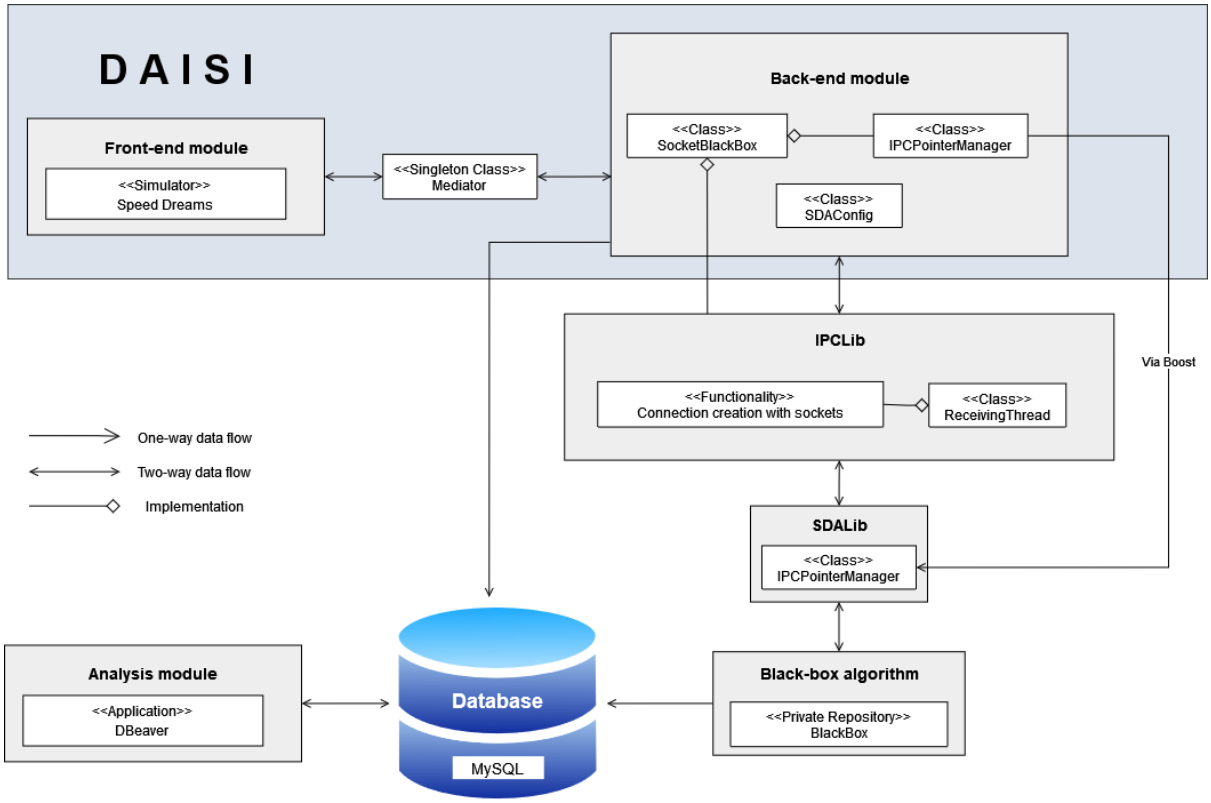


Figure 1.1: An simplified visual representation of the program structure.

1.2.2 Front-End and Back-End Modules

These main two modules of the system are connected with one another through the **Mediator** singleton class, and therefore also communicate with one another through the mediator. The **Mediator** class is included in Figure 1.1. The back-end module is connected to two more modules: the database and SDALib. With the help of the back end's connection with SDALib, it sends the intervention data to the front-end module so that the decisions of the AI can be visualised in the simulation. Via the front-end module, the intervention data is sent to the simulator. The front end can be seen as an encapsulation of the simulator, the intervention signals display and the menus, which manages all the sounds and visuals of the system. The driving and intervention data is sent from the back end to the database.

Front End: Saving Simulator Settings

The front-end module sends the driving data to the back-end module with the help of the **Driver** class so that the AI can make a decision based on the driver's behaviour. Via the menus (researcher menu, data compression menu, developer menu, etc.) the settings for the simulator are determined. The menus share their information to the mediator.

With the help of the **DatabaseSettingsMenu**, settings such as the user's username and password are collected. The **DatabaseConnectionManager**, as the name suggests, manages the connection with the database by applying the settings from the menu.

Back End: the Decision Maker

In order to save how the decisions are stored, we use a decision tuple data structure, named the **DecisionTuple** class. This decision tuple contains the different kinds of interventions. When receiving the data from the **SocketBlackBox** class, the values in these decisions are updated.

Thereafter, the **InterventionExecutor** class is called which contains the current intervention type and, for example, what icons to show. In this class, the **ExecuteIntervention** method is called first which updates all decisions on whether or not an intervention should be taking place. Depending on the settings in the front end's researcher menu, the decisions are filtered based on their threshold and allowed action. Next, the decision tuple is updated again in order to update the visualisation of the icons.

Adding a new intervention type and decision
In the back end, adding a new intervention type is achieved with the help of the factory design pattern by having the intervention executor create a new class. Each class has a different set of information on what to do when intervening and what icons to show.
In order to add a new decision, such as steering, a new class will have to be implemented in the **Decision** class. The **Decision** class contains the actions that should be taken.

IPCLib and SDALib

The back end is connected with SDALib via IPCLib. IPCLib can be interpreted as a way to simplify the inter-process communication between the back end and SDALib. IPCLib sends a ping and returns the result of the decision maker over the socket. Boost manages the pointers. Both the back end and SDALib have a pointer manager so that IPCLib communicates the correct pointers. The back end's pointer manager is the **IPCPointerManager** class. The **IPCPointerManager** class manages the boost pointers. One of SDALib's many advantages is that it provides more flexibility to add support for other programming languages. In order to add support for other programming languages, you can easily create a new class in SDALib with similar functionality to the **PythonDriver** and **SDADriver** classes. SDALib uses the driver class linked to a particular programming language to read the output from the black box. As visualised in Figure 1.1, SDALib currently includes a **PythonDriver** class so that the output of a black box written in Python can be read.

1.2.3 Analysis Module

The analysis module is connected with the database, and therefore can visualise the structure of the database in its interface. With the help of queries, the analysis module can specifically request particular data from the database. As is shown in Figure 1.1, the analysis module is a different application and a separate entity from DAISI because this module is only used for post analysis of the data. The analysis module has no direct connections with DAISI's modules, as also shown in Figure 1.1. The analysis module is comprised of the open-source multi-platform database tool named DBeaver, which includes functionality to connect to the database.

1.2.4 Black-Box Algorithm

The back end functions as a (flexible) framework for the black-box algorithm that is used to deliver interventions in the actual simulator. In order to pass the intervention data from the black box on to the back end, the data (in Figure 1.1 visualised as *Black Box Algorithm*) is first sent to SDALib. Then, the intervention data will be sent to the back-end module via IPCLib.

If the black box is developed in such a way that it calls functions from Speed Dreams directly, it will have to be changed in such a way that it will call Speed Dreams' functions contained in SDALib. This is possible because SDALib contains the core of Speed Dreams so that the black box can use Speed Dreams' functions.

1.2.5 Internal Meta Data

As shown in Figure 1.1, both SDALib and the black box can send internal meta data to the database. It is also possible for SDALib and the black box to not send the internal meta data. During the simulation,

the internal data is written to a buffer file. The **FileDataStorage** class is used for this functionality. Every 'tick', the **Save** function in this class is called. In case you want to store extra data, you will have to add more parameters to which represent the objects you want to save. After the simulation, the buffer file is saved to the database by opening a connection toward the database with the help of the **SQLDatabaseStorage** class. This buffer file system was implemented in order to increase the speed of saving the data to the database as MySQL is fairly slow. In contrast, buffer files are really fast. The simulator writes a trial ID to the buffer file containing the metadata. The MySQL library has already been included in the system.

Database

It is important to note that the structure of the database should be changed accordingly in order to save the data according to how these functions are structured and/or what parameters are given. Lastly, the database works with MySQL which allows for query functionality.

Chapter 2

Research

2.1 New Research Options

The research options provided in DAISI are numerous, but not limitless. Perhaps you wish to control some aspect of the vehicle which is not accessible, or you wish to research an alternative way of indication. You might also have two or more algorithms, controlling different aspects, that you now want to test as a singular algorithm. Luckily, you're not out of luck. By editing some code files, you can accomplish these things.

2.1.1 Adding Decisions

The `Driver` class contains the simulation's car struct. It will send this car over to the `Mediator` class each drive tick. The `Mediator` will then ask the `DecisionMaker` to make a decision. This class will communicate with the black box and send its response to an `InterventionExecutor` to evaluate the response. Decisions are the cornerstone of DAISI. A decision goes into a singular aspect of the vehicular control of the driving simulation. To add a decision, you will need to edit the source code of both SDALib and DAISI.

This guide assumes the new decision you wish to add cannot be categorized as a decision related to steering or speed. If your new decision is related to one of these categories, you can ignore any steps surrounded by `*`'s.

DAISI

To start, navigate to `source-2.2.3/data/data/indicators/Config.xml`. In the section named `indicators`, add new sections for your new decision, as demonstrated in Figure 2.1. Values that you should certainly edit are marked with a red box. *Make sure that the `xpos` value you use is not the same as the `xpos` value of another set of decisions, as that would have the indicators overlap.*
In the same directory as this config file you can find the folders `sound` and `texture`. Add your sound files to the `sound` folder and add your indicator images to the `texture` folder, as shown in Figure 2.2.

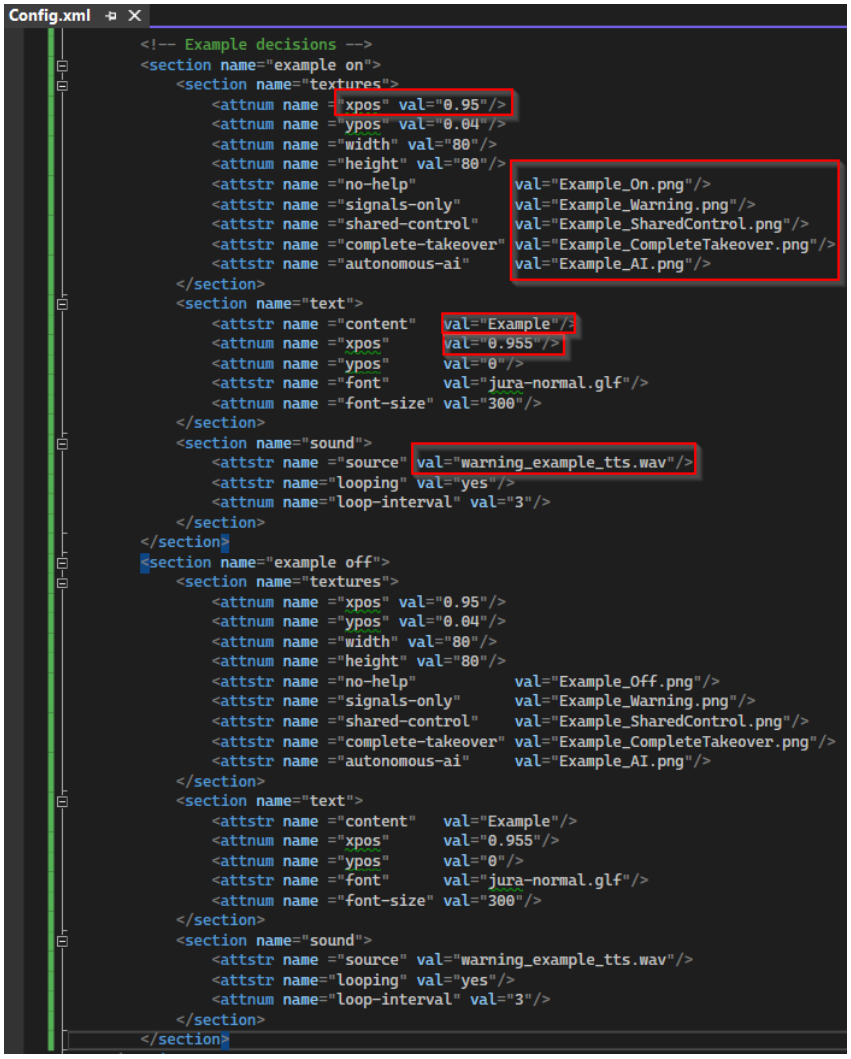


Figure 2.1: Adding sections for the new decision.

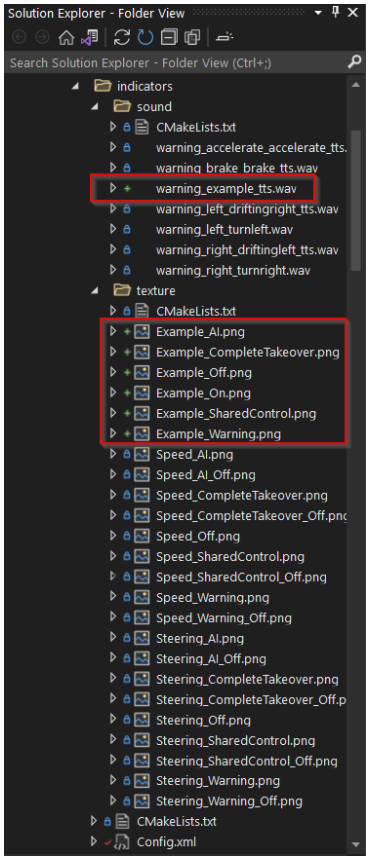


Figure 2.2: Adding the images and sound files for the new decision.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/ConfigEnums.h`. In here, add your decision to the `tAllowedActions` struct, as demonstrated in Figure 2.3. Continuing in this file, add defines for the specific actions related to this decision, as demonstrated in Figure 2.4. Each define should be one higher than the latest one. Increment `NUM_INTERVENTION_ACTION` by the amount of actions you added.

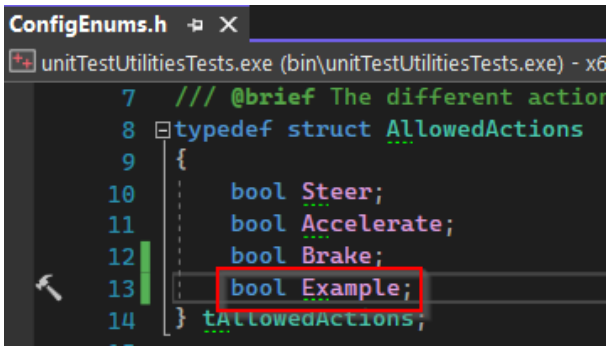


Figure 2.3: Adding the new decision to the allowed actions.

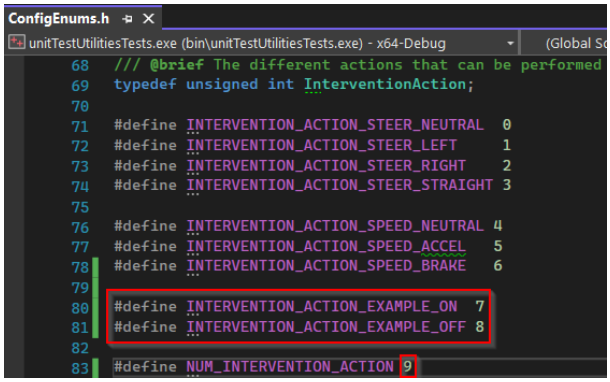


Figure 2.4: Adding defines for the new decision's actions.

Add a define for your decision category, as shown in Figure 2.5. Increment `NUM_INTERVENTION_ACTION_TYPES` by one. Extend the `s_actionToActionType` array, adding the relevant decision category. This is demonstrated in Figure 2.6.

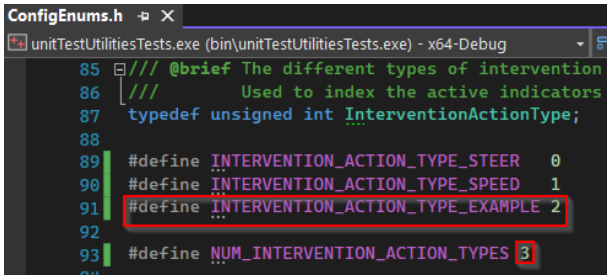


Figure 2.5: Adding a define for the new decision category.

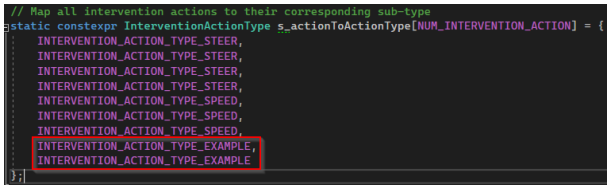


Figure 2.6: Mapping the new decision actions to their decision type.

Add your new decision to the `tParticipantControls` struct, as demonstrated in Figure 2.7. As the last modification to this file, add a threshold for your new decision to the `tDecisionThresholds` struct and add a define for a default value, as demonstrated in Figure 2.8. This threshold will be used if more specific ones are not set.

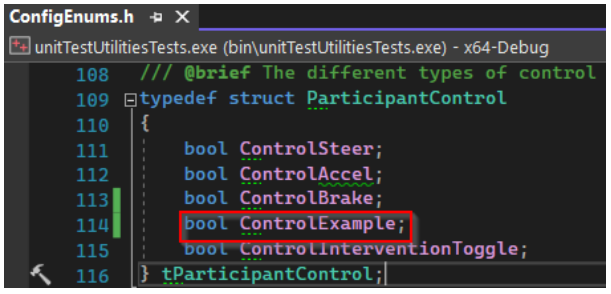


Figure 2.7: Adding the new decision to the participant control struct.

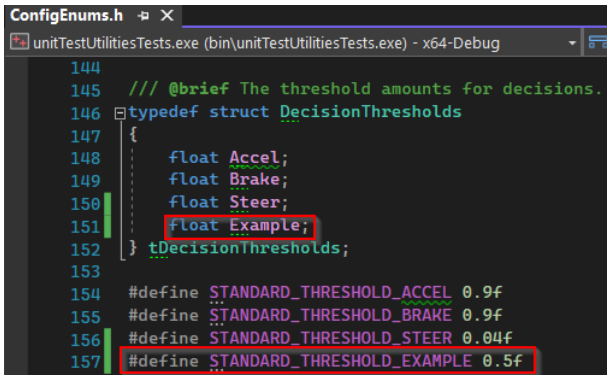


Figure 2.8: Adding the new decision to the thresholds struct.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/frontend/IndicatorConfig.h`. Find the `s_interventionActionString` array. Add the decision names that you added to the config file, as demonstrated in Figure 2.9. Navigate to `source-2.2.2/data/data/menu/ResearcherMenu.xml`. In

the section named `dynamic controls`, add sections for the allowed actions (shown in Figure 2.10) and participant control (shown in Figure 2.11) checkboxes. Make sure the `y` value is different from the `y` values of other sections within the same list (indicated by the XML comment).

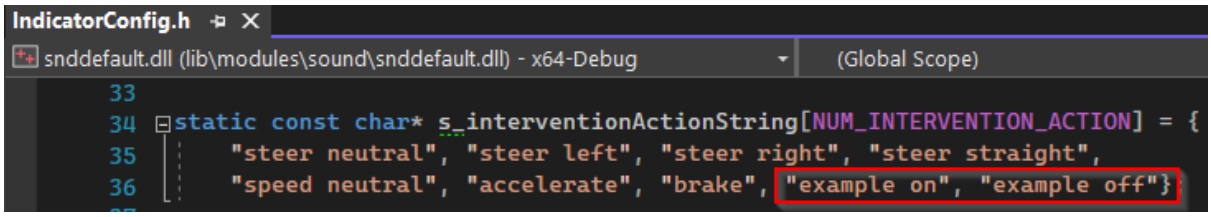


Figure 2.9: Adding the names of the decisions.

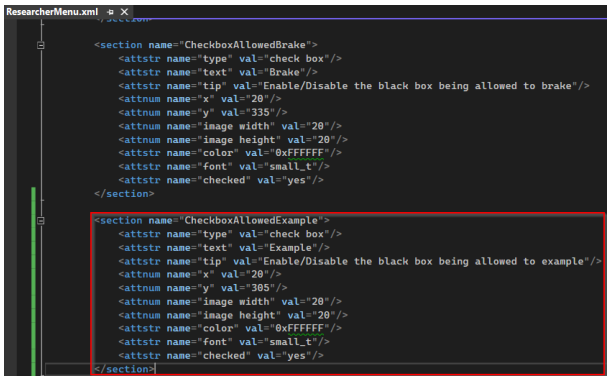


Figure 2.10: Adding the checkbox for the allowed action.

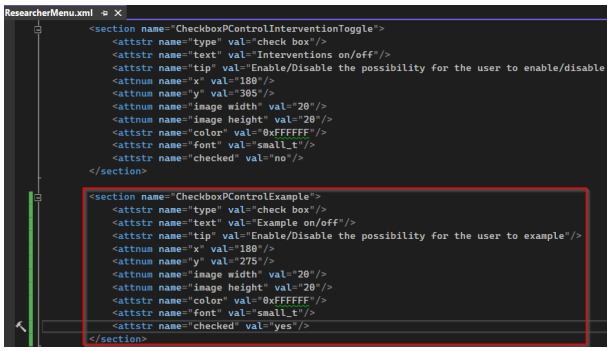


Figure 2.11: Adding the checkbox for participant control.

You are not yet finished with the Researcher Menu. The XML for the check boxes exists now, but the actual check boxes still need to be created. To do this, navigate to `source-2.2.3/src/modules/userinterface/legacymenu/mainscreens/ResearcherMenu.cpp`. Add two defines for the newly created checkboxes, as shown in Figure 2.12. Use the names as you have written them in the XML file. Increment `PCONTROL_AMOUNT` and `ALLOWED_ACTIONS_AMOUNT` by one, as shown in Figure 2.13.

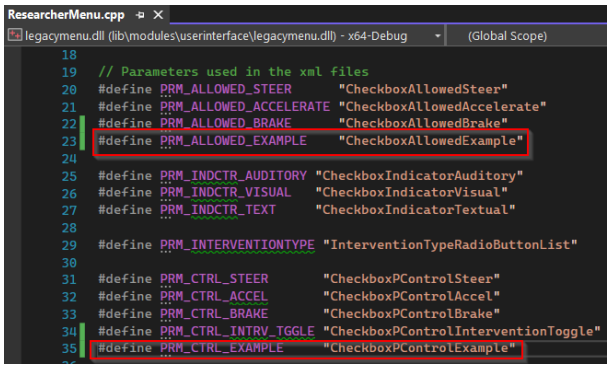


Figure 2.12: Adding defines for the names used in the ResearcherMenu.xml file

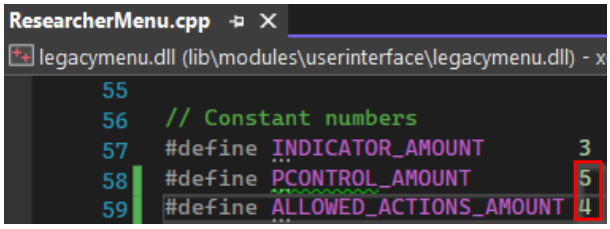


Figure 2.13: Incrementing the defines for amounts of participant control and allowed action options.

Add two functions `SelectAllowedExample` and `SelectControlExample`, as shown in Figure 2.14. The implementation for these functions is provided in this figure, as these functions should only have the purpose of setting a single value. Find the `ResearcherMenuInit` function. Add two calls to

`GfuiMenuCreateCheckboxControl` and store the return values in the relevant arrays of checkbox ids, as demonstrated by Figure 2.15. Because you incremented the defines, there is now room in these arrays.

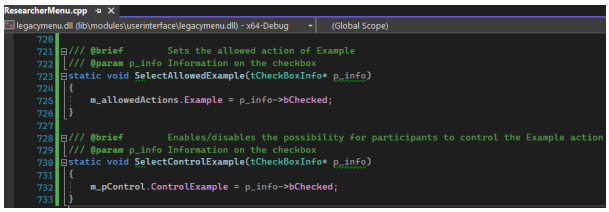


Figure 2.14: Adding functions for setting the allowed action and participant control.

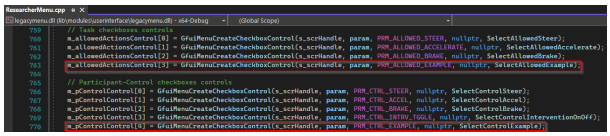


Figure 2.15: Adding the calls to create the new checkboxes.

To make sure the state of the new checkboxes is saved between launches of the system, add saving (shown in Figure 2.16), synchronization (shown in Figure 2.17) and loading code (shown in Figure 2.18).

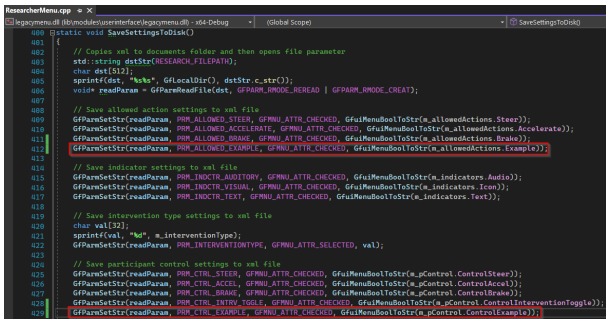


Figure 2.16: Adding saving of settings to the Researcher Menu.

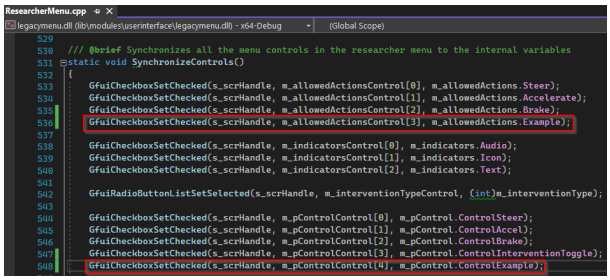


Figure 2.17: Adding synchronization of settings to the Researcher Menu.

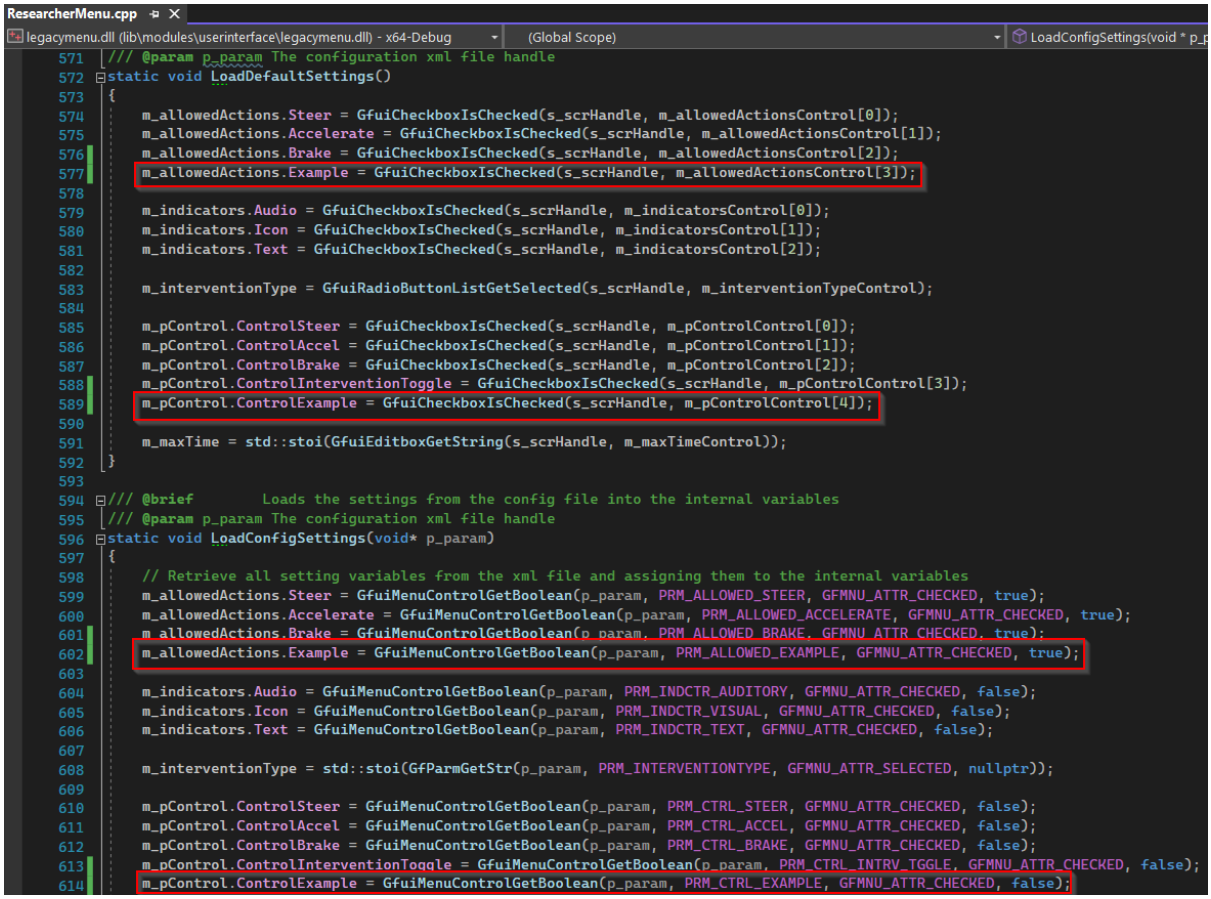


Figure 2.18: Adding loading of settings to the Researcher Menu.

You will now have to add the default thresholds for your new decision. Navigate to `source-2.2.3/data/data/menu/DeveloperMenu.xml`. To the `DefaultThresholdValues` section, add your desired default thresholds for each intervention type, as shown in Figure 2.19. In the `dynamic controls` section, add an edit box for the new threshold, as demonstrated in Figure 2.20. Make sure the `y` value does not overlap with the `y` values of other threshold edit boxes.

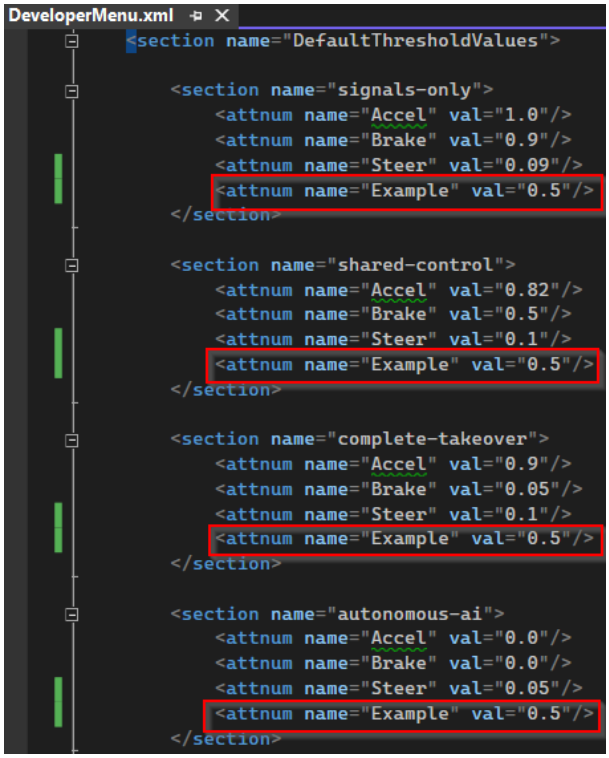


Figure 2.19: Adding default thresholds.

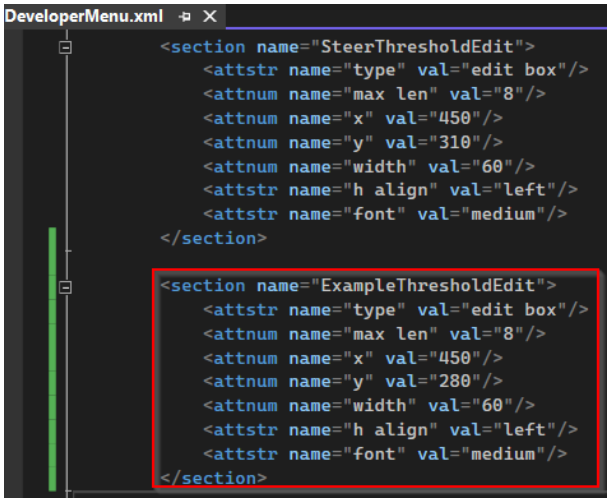


Figure 2.20: Adding an edit box for the threshold.

You will also need to adjust the `y` value of the button resetting the default values, as demonstrated by Figure 2.21. You will also have to add a label for the edit box. You should do this in the `static controls` section of the XML file. The name of this section should be one higher than the name of the latest section. This is demonstrated in Figure 2.22.

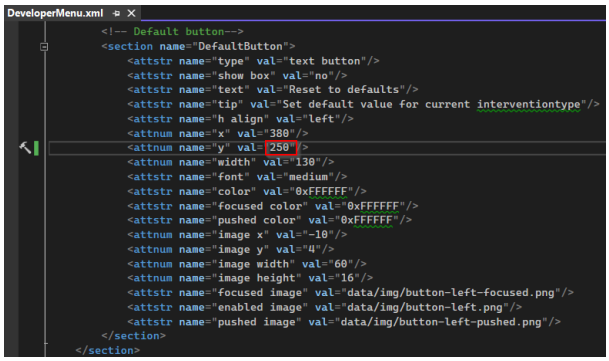


Figure 2.21: Changing height of default button.

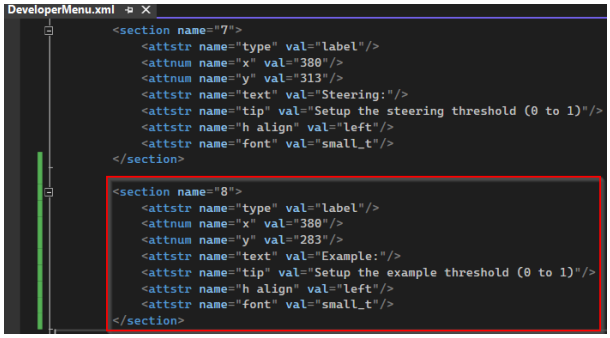


Figure 2.22: Adding a label for the edit box.

Navigate to `source-2.2.3/src/modules/userinterface/legacymenu/DeveloperMenu.cpp`. To be able to actually save the threshold value, you will need to make a multitude of changes. To start, add a define for the decision name, as shown in Figure 2.23. Add an id for the edit box control and add the default threshold value to `m_decisionThresholds`, as shown in Figure 2.24.

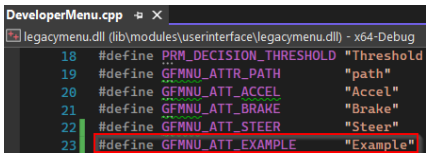


Figure 2.23: Adding define for saving/loading of threshold.

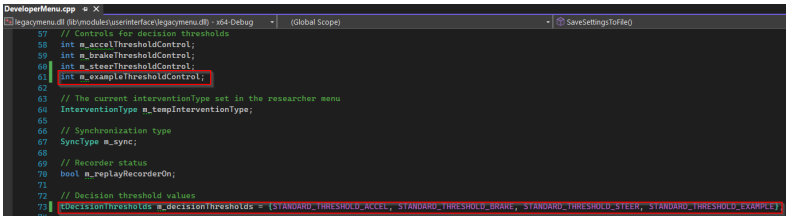


Figure 2.24: Adding default threshold and an id for the edit box.

Add loading and synchronization of the edit box as demonstrated by Figure 2.25 and saving as shown in Figure 2.26.

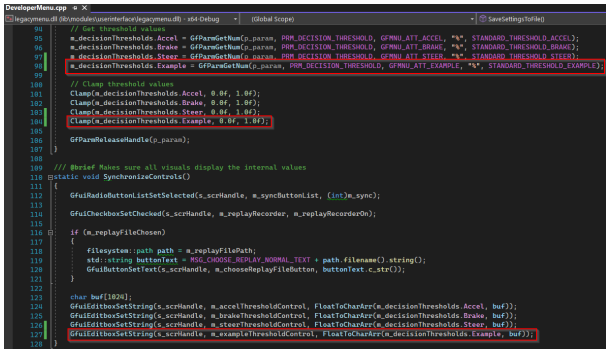


Figure 2.25: Adding loading and synchronization of the threshold setting.

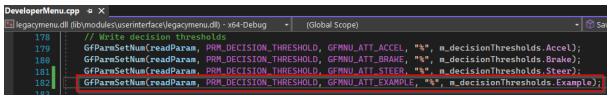


Figure 2.26: Adding saving of the threshold setting.

Add a function that will actually set the threshold value, as shown in Figure 2.27. Use this function to write the default threshold value, as shown in Figure 2.28.

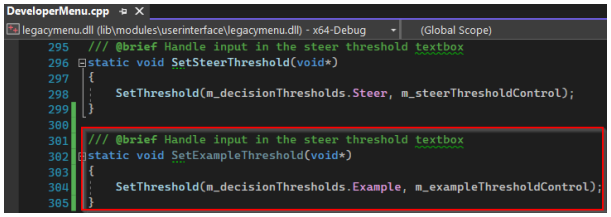


Figure 2.27: Adding method for setting the threshold value.

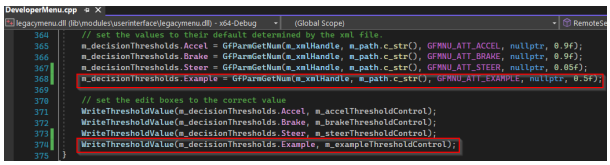


Figure 2.28: Adding setting of default threshold value.

To finish off the Developer Menu, create the actual edit box, as shown in Figure 2.29. Pass the method you just created and set the id equal to the return value of the `GfuiMenuCreateEditControl` method.

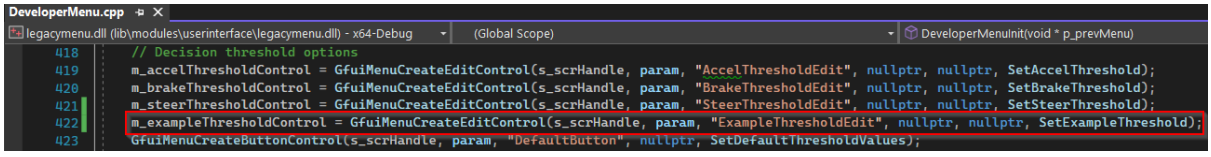


Figure 2.29: Creating the edit box.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/CarController.h`. Add function definitions for `GetExampleCmd` and `SetExampleCmd`, as shown in Figure 2.30. In the associated source file, add the implementation for these methods, as shown in Figure 2.31. Here, one would set control

values that will be used in a later function to determine an action. For this example, the unused field `reserved1` is used, but it is recommended you use a field that is relevant to the decision you wish to implement. The type of these methods is `float` in this example, though you are not restricted to that.

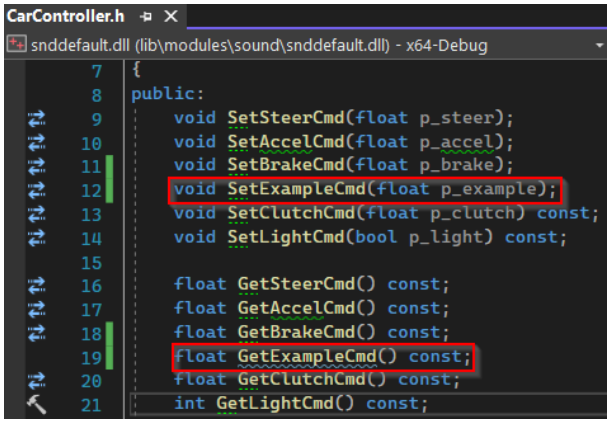


Figure 2.30: Adding functions definitions to the car controller.

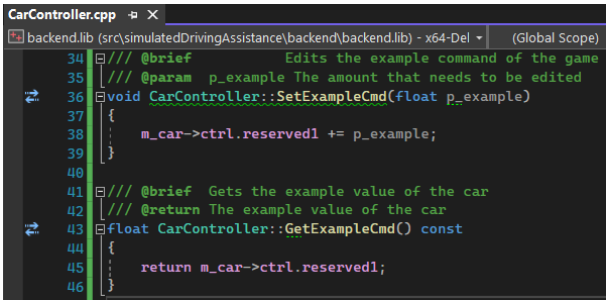


Figure 2.31: Adding function implementations to the car controller.

Navigate to the `source-2.2.3/src/simulatedDrivingAssistance/backend` folder and add the two new files `ExampleDecision.cpp` and `ExampleDecision.h`. You must also add these files to the `CMakeLists.txt` file in the same folder, as shown in Figure 2.32. In these files, create a class that inherits from `Decision`, and implement your desired action. The header file (shown in Figure 2.33) is fairly simple. Unless you wish to add more data structures that will help in the decision process, that is what the file should look like. In the source file (shown in Figure 2.34), you will need to implement these methods. If you want to add additional implementation to the decision process, that is where you would add it.

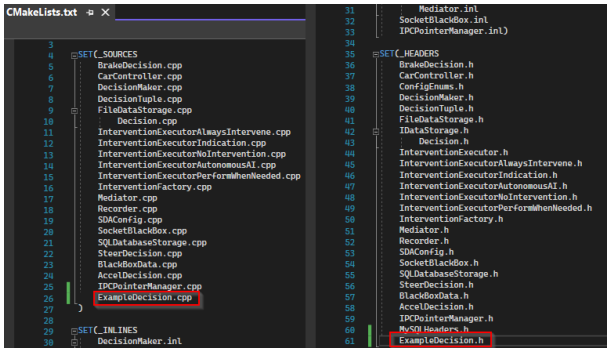


Figure 2.32: Adding the decision files to CMakeLists.txt. The image has been split in half and these halves have been put side by side.

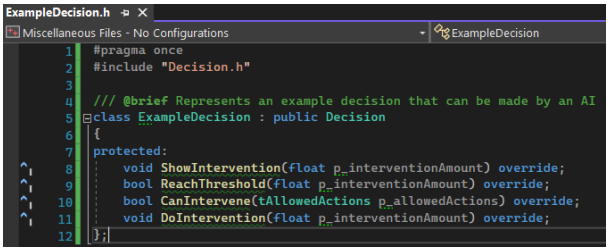


Figure 2.33: Example header for the new decision.

ExampleDecision.cpp

backend.lib (src\simulatedDrivingAssistance\backend\backend.lib) - x64-Del (Global Scope)

```cpp
#include "ExampleDecision.h"
#include "Mediator.h"

/// @brief Shows the intervention on the screen
void ExampleDecision::ShowIntervention(float p_interventionAmount)
{
    SMediator::GetInstance()->CarControl.ShowIntervention(INTERVENTION_ACTION_EXAMPLE_ON);
}

/// @brief Runs the intervene commands
/// @param p_interventionAmount The intervention amount
void ExampleDecision::DoIntervention(float p_interventionAmount)
{
    SMediator::GetInstance()->CarControl.SetExampleCmd(p_interventionAmount);
}

/// @brief tells whether the intervention amount is higher than the threshold
/// @param p_interventionAmount The intervention amount
/// @return whether the threshold is reached
bool ExampleDecision::ReachThreshold(float p_interventionAmount)
{
    float threshold = SMediator::GetInstance()->GetThresholdSettings().Example;
    return threshold < p_interventionAmount;
}

/// @brief tells whether the simulator can be intervened by the decision
/// @param p_allowedActions The allowed black box actions
/// @return whether the simulator can be intervened
bool ExampleDecision::CanIntervene(tAllowedActions p_allowedActions)
{
    return p_allowedActions.Example;
}
```

Figure 2.34: Example implementation for the new decision.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/DecisionTuple.h`. Include your new decision's header file. Increment `DECISION_AMOUNT` by one. Add definitions for `SetExampleDecision`, `GetExampleAmount` and `ContainsExampleDecision`. Finally, add a private variable `m_exampleDecision`. All these changes are demonstrated in Figure 2.35. In the accompanying source file, you will need to implement these new methods. There are also some further changes, shown in Figures 2.36 and 2.37: you need to add a new if statement in the `GetActiveDecision` method for the new decision and you need to add expand the `Reset` method to reset the new decision.

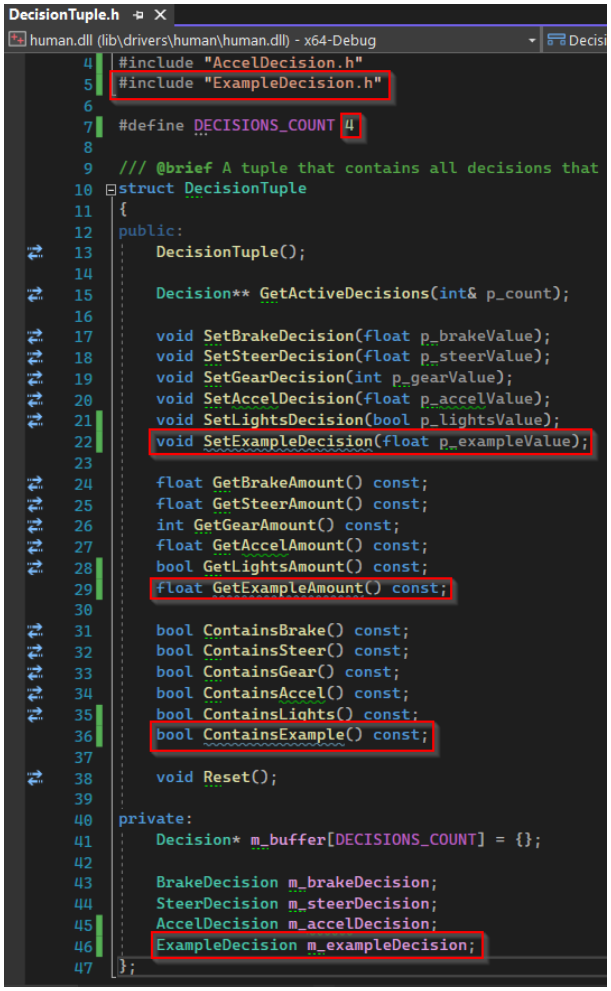


Figure 2.35: Adding definitions, an include, and a variable to DecisionTuple.h.

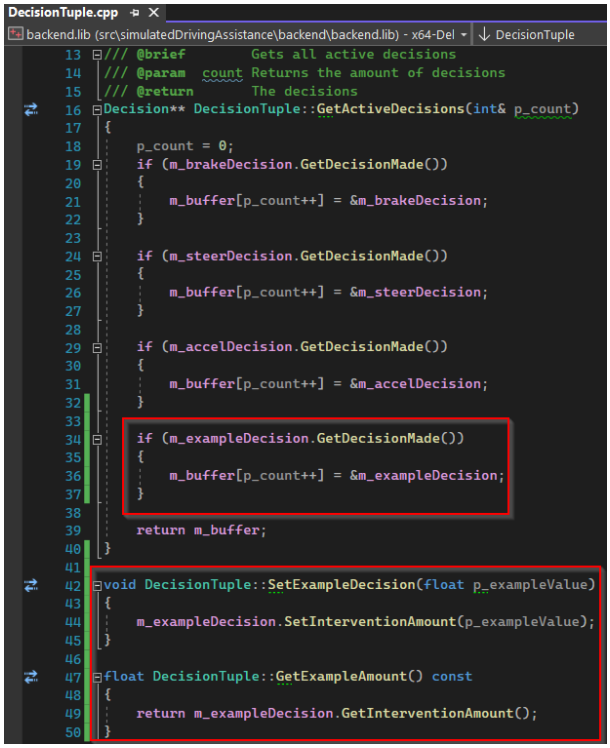


Figure 2.36: Adding if statement in GetActiveDecisions and implementing two methods.

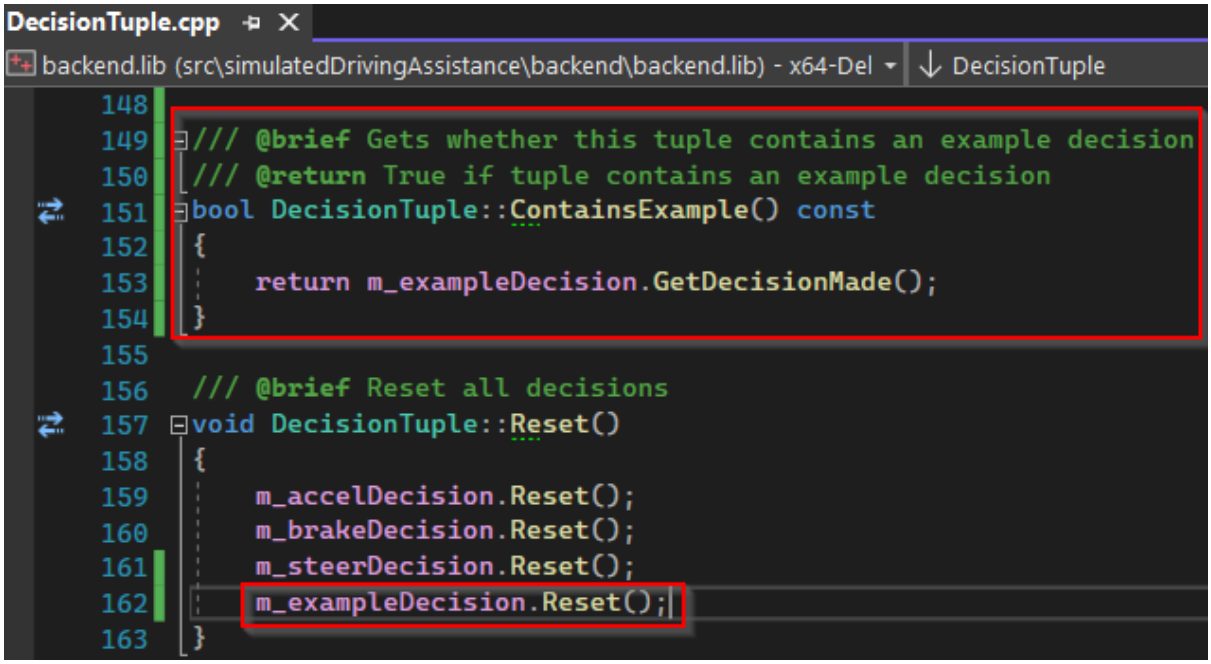


Figure 2.37: Implementing the third method and extending the Reset method.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/Mediator.h`. Add the function definition for `CanUseExample`, as shown in Figure 2.38. The implementation for the `Mediator` is found in the `Mediator.inl` file. In the define at the top, add the definition for the `CanUseExample` method, as shown in Figure 2.39. Implement the function in this file, as demonstrated by Figure 2.40.

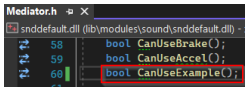


Figure 2.38: Adding function definition to the Mediator.h file.

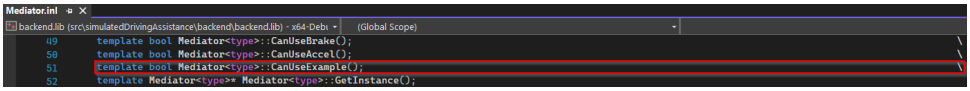


Figure 2.39: Adding function definition to the Mediator.inl file.

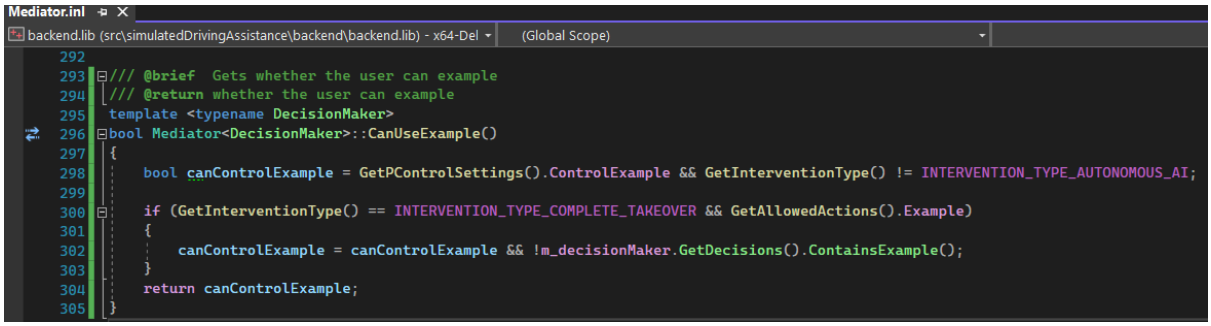


Figure 2.40: Implementing the function in the Mediator.

Navigate to `source-2.2.3/src/libs/robottools/rthumandriver.cpp`. Find the `common_drive` method. In this method, control is handled. Find the section of this function that uses the control that you set in `CarController::SetExampleCmd` and add modifications as you please for how this control should be affected. You should surround this code with an if block, which checks whether the participant is allowed to control this aspect. To check this, access the `Mediator` (`SMediator::GetInstance()`) and call its `CanUseExample` method, which you have just added.

To save your new data, you will need to make some final changes in DAISI. Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/FileDataStorage.h`. To the header defines, add your new decision (and control), as shown in Figure 2.41. For this example, we will assume our variable is only ever equal to the user input (potentially modified by the black box) and not dependent on the environment, so we do not add it to the gamestate headers. In the same file, add private variables `m_exampleDecision` and `m_exampleValues`, as shown in Figure 2.42.

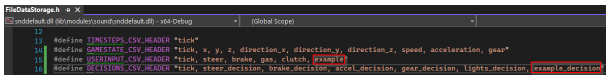


Figure 2.41: Adding the decision to the CSV headers.

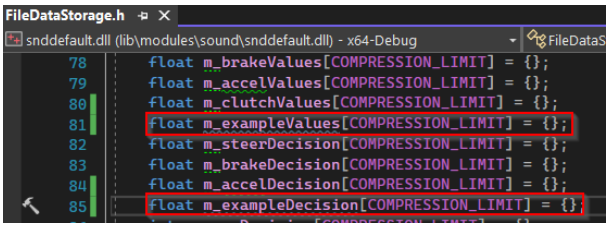


Figure 2.42: Adding arrays for storing the values of the new decision.

In the accompanying source file, you will need to change the `SaveHumanData` and `WriteHumanData` methods (as shown in Figures 2.43 and 2.44), and the `SaveInterventionData` and `WriteInterventionData` methods (as shown in 2.45 and 2.46). If your aspect is furthermore a variable related to the car, you will need to also to change the `SaveCarData` and `WriteCarData` is a similar manner. It is important that you keep all decisions in the order that they appear in the headers. When changing the `Write` methods, take care that only the last write ends in a newline, while the other entries ends in a comma. You need to decide on a sampling method as well. These are documented in Section 3.2.1.

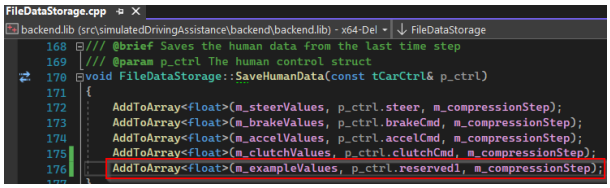


Figure 2.43: Saving the data of the example human control.

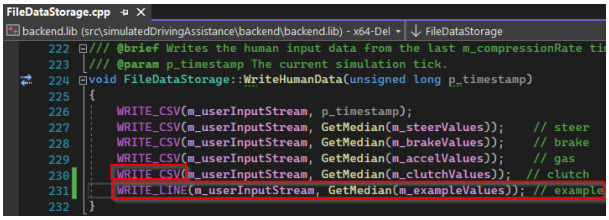


Figure 2.44: Writing the data of the example human control.

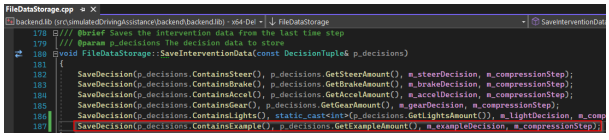


Figure 2.45: Saving the data of the example decisions.

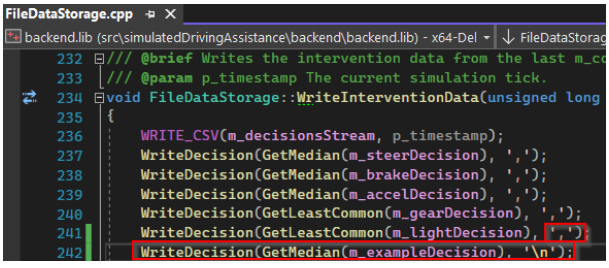


Figure 2.46: Writing the data of the example decisions.

You will further need to change a significant amount of the tables and methods in `source-2.2.3/src/simulatedDrivingAssistance/backend/SQLDatabaseStorage.cpp`, as you will need to add a new table for your decision, as well as new columns for your decision in other (temporary) tables. How you can change this is described in Section 3.1.

You will also need to add deserialization of the values. You can do this by navigating to

`source-2.2.3/src/simulatedDrivingAssistance/backend/SockerBlackBox.inl`. In this file, you will need to a a define for `CONVERT_TO_EXAMPLE` and add your new decision to `m_variableDecisionMap`, as shown in Figure 2.47.

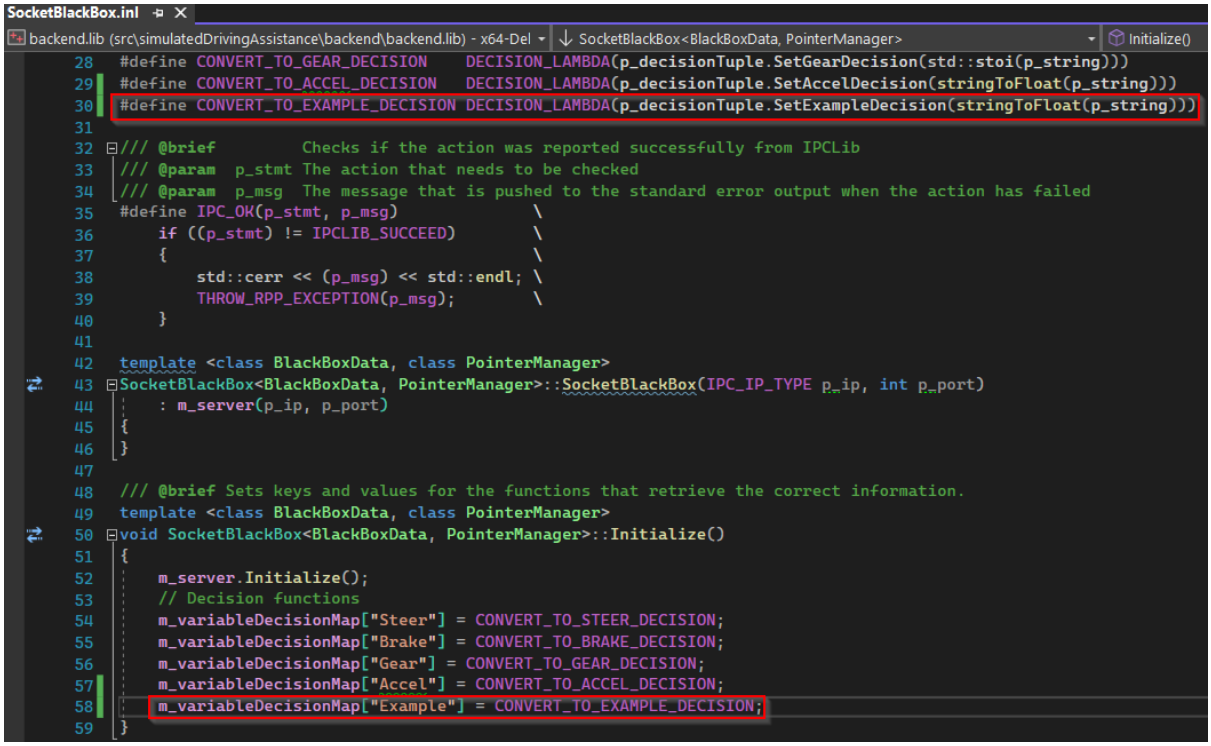


Figure 2.47: Adding deserialization for the new decision.

SDALib

Navigate to `SDALib-CPP/SDALib-CPP/SDAAction.hpp`. Add a private variable for your example decision, as shown in Figure 2.48. In the accompanying source file, add pushes for your new decision, as demonstrated by Figure 2.49. To allow simulating ahead, navigate to `SDALib-CPP/SDALib-CPP/SDASpeedDreams.cpp` and modify the `SDASpeedDreams` function as shown in Figure 2.50. Use the control value for the aspect that you also used in DAISI.

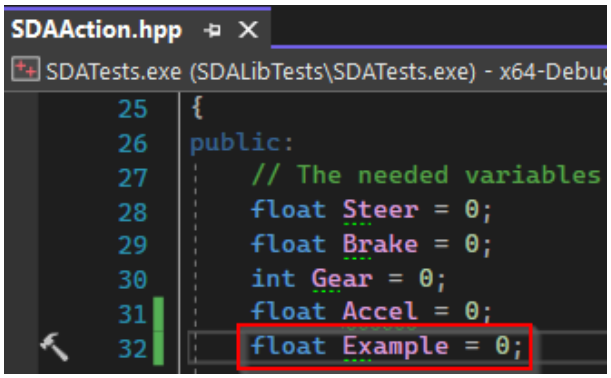


Figure 2.48: Adding the new decision to SDAAction.

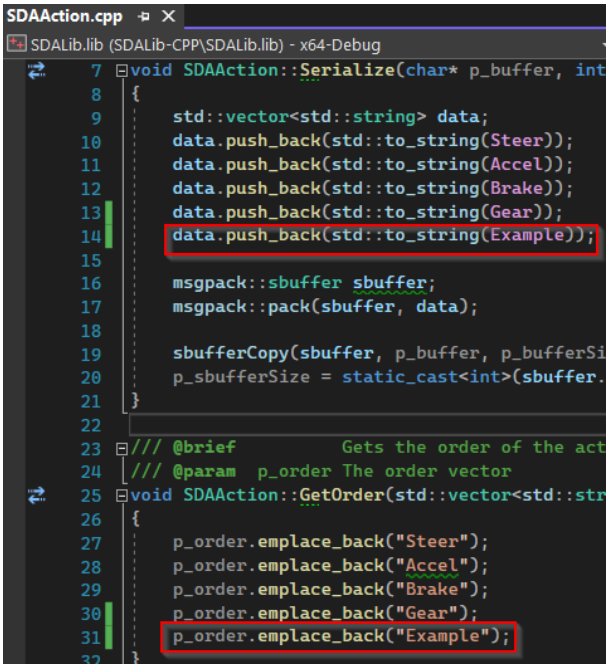


Figure 2.49: Making sure the new decision is sent over.

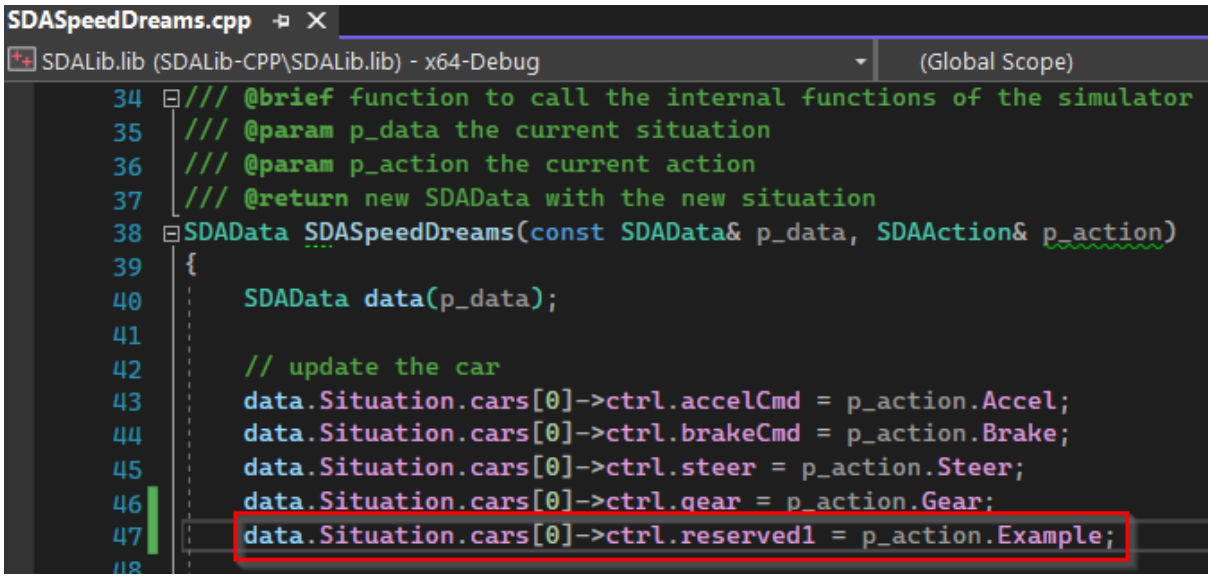


Figure 2.50: Making sure the new decision is used in lookahead.

Integration Testing

For the integration tests, you should also store and be able to read the new actions and decisions. To do so, navigate to `SDALib-CPP/SDALib-CPP/SDAReplay/ReplayDriver.h`in SDALib. Add two lines reading the new decision from the recording, as shown in Figure 2.51.

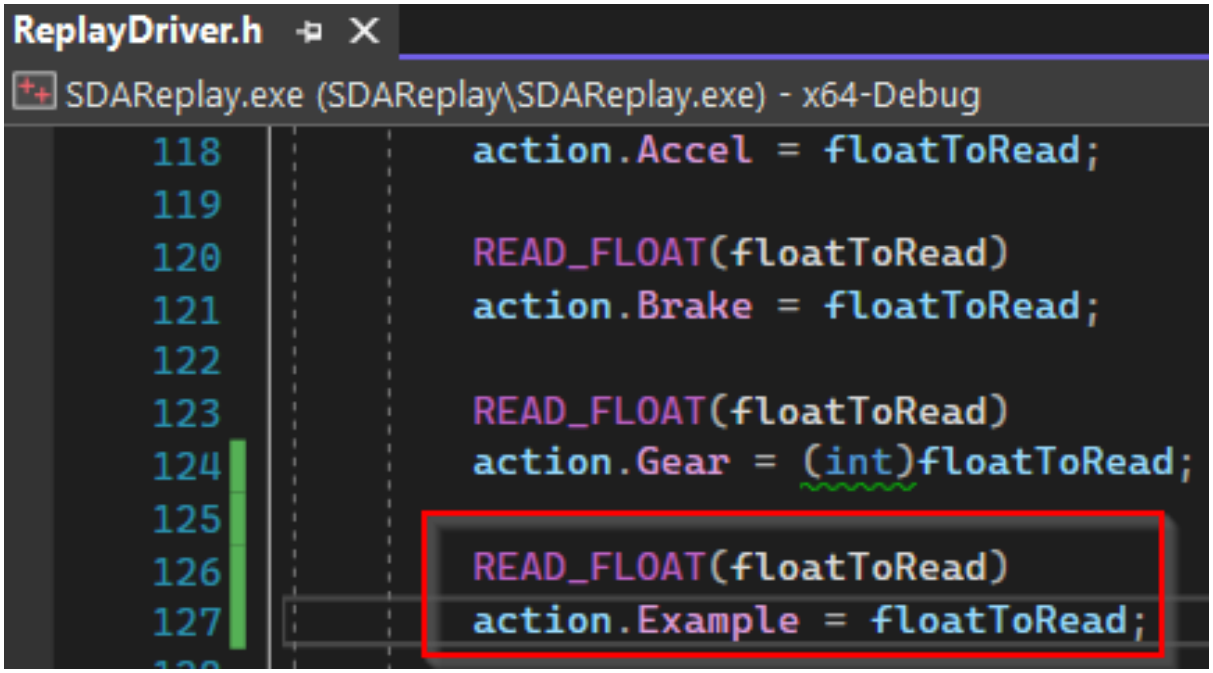


Figure 2.51: Adding the new decision to the replay driver in SDALib.

You will also have to make changes in DAISI. Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/Recorder.h`. Add defines for your new decision, as shown in Figure 2.52. Increment `DECISION_RECORD_PARAM_AMOUNT`, as shown in Figure 2.53. You also need to increment the recorder version.

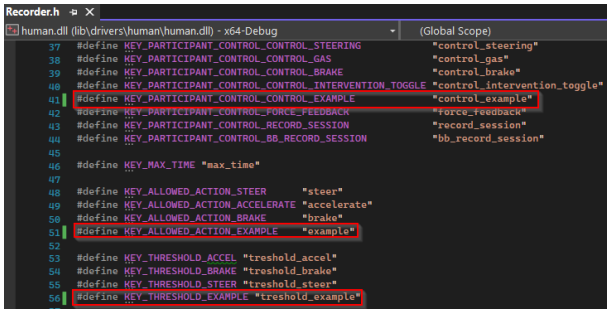


Figure 2.52: Adding defines for saving the decision settings.

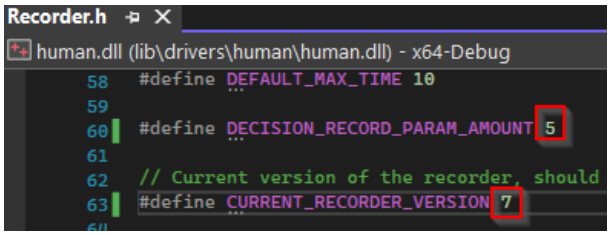


Figure 2.53: Incrementing the amount of stored decisions and the recorder version.

In the accompanying source file, write the settings in the `WriteRunSettings` method, as shown in Figure 2.54. You should write the actual decision in `WriteDecisions`, as shown in Figure 2.55.

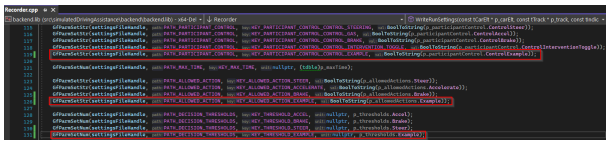


Figure 2.54: Writing the new decision settings.

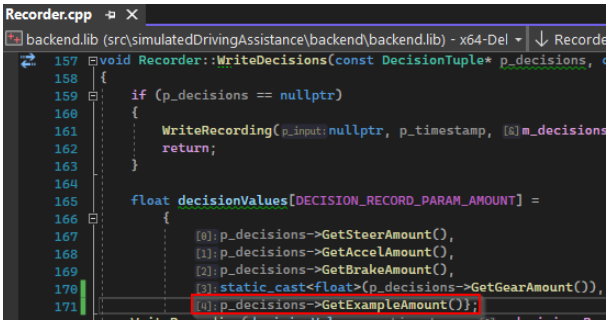


Figure 2.55: Writing the decision.

You should add a function to add these new simulation settings to old recordings. For functionalities that were not previously limited, the default participant control setting should be `true`. The allowed actions entry can have either value: the decisions of that recording would not contain any new decisions anyhow. The same goes for the threshold. You should also call this method. This is demonstrated in Figure 2.56. Finally, you should load the new settings from a recording. This is done in the `LoadRecording` method, as demonstrated in Figure 2.57.

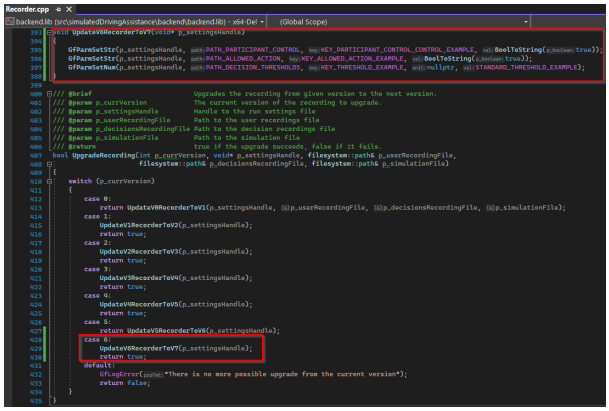


Figure 2.56: Updating the old recordings.

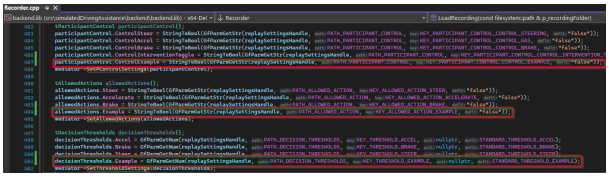


Figure 2.57: Loading the new settings.

Your decision might not be a value that is currently saved for the integration tests. In this case, you should find the references to the `WriteUserInput` and/or `WriteSimulationData` methods and change the arrays passed to this method. You will need to increment the defines that define the lengths of these arrays as well. Furthermore, you should navigate to `source-2.2.3/src/drivers/replayDriver/Driver.cpp` and change the `Drive` and `ValidateSimulationData` methods, reading your newly added values in such a way that the order in the arrays is the same as the order in which the variables are read. After these steps, your new decision should now work. Of course, the newly added decision has not been tested yet. You can expand the existing tests for files that you've edited in the `source-2.2.3/src/simulatedDrivingAssistance/backendUnitTests` and `SDALib-CPP/SDALibTests` folders, or add new tests in the `DecisionsTests.cpp` file for the newly added files.

2.1.2 Adding Intervention Types

Intervention types decide how interventions are applied to the same simulation. Every time a black box returns an action, DAISI packs this action into a `DecisionTuple`. This tuple is passed from the `DecisionMaker` to a specific `InterventionExecutor`. This class will decide how the intervention is applied. To add an intervention type, you will need to edit the source code of DAISI.

To start, navigate to `source-2.2.3/data/data/indicators/Config.xml`. In this file, find the section with the name indicators. Every direct subsection in this section represents a decision. To each decision's `textures` section,

add `<attstr name="example-intervention-name" val="example-intervention-image.png"/>`, as demonstrated in Figure 2.58. You should replace `example-intervention-name` with the actual intervention name you have in mind, and replace `example-intervention-image.png` with the filename of an image you have in mind for this intervention type in conjunction with this decision. Add all images you have newly defined to the `texture` folder, found in the same directory as this config file. This is illustrated in figure 2.59. Navigate to `source-2.2.3/data/data/menu/DeveloperMenu.xml`. In this file, add default threshold for your intervention, as shown in Figure 2.60.

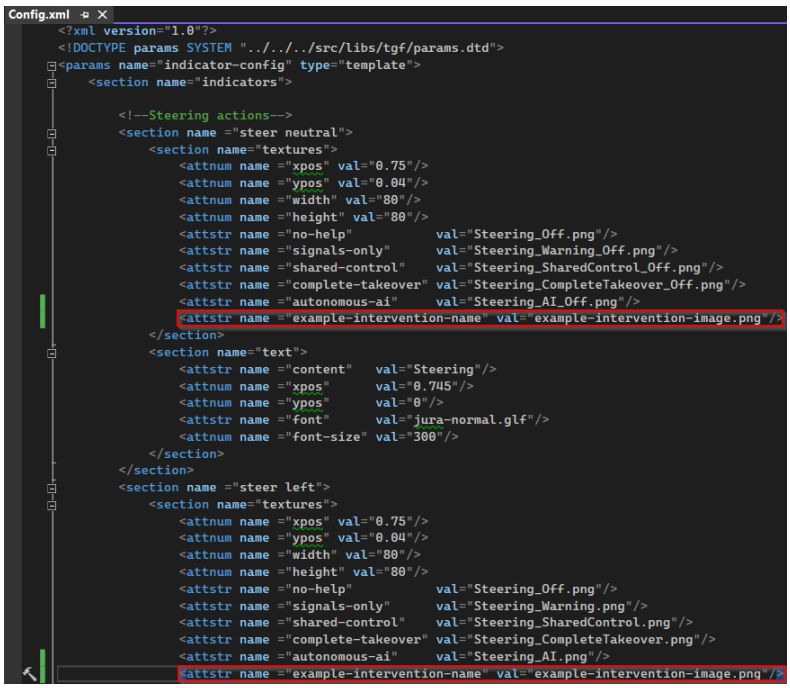


Figure 2.58: Adding the indicators for the new intervention type. You can decide the displayed image for each decision.

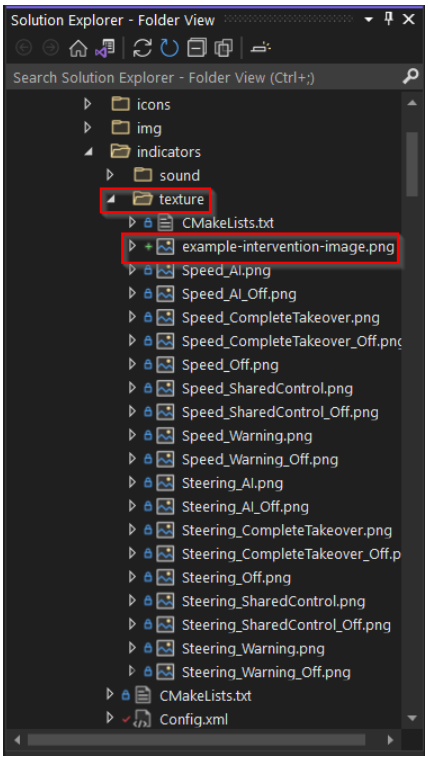


Figure 2.59: Adding the images for the new intervention type.

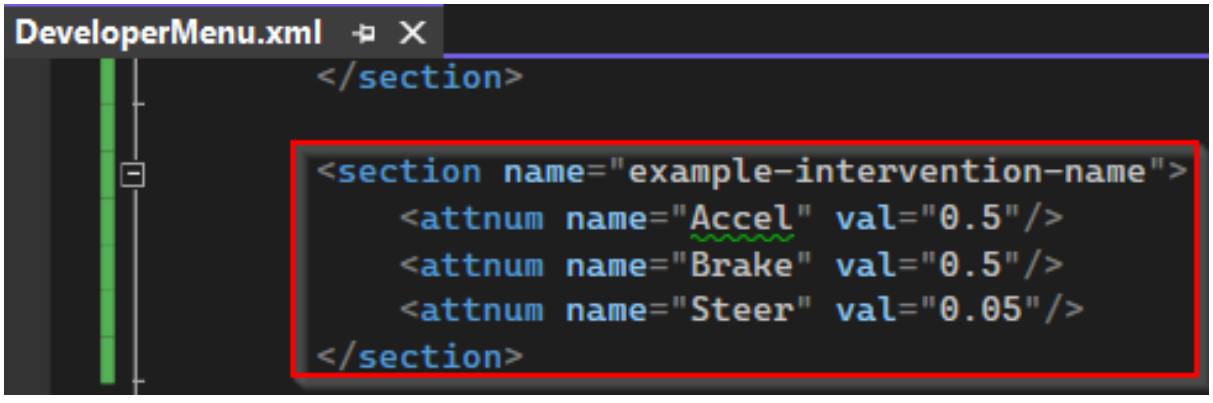


Figure 2.60: Adding the default threshold values.

Navigate to `source-2.2.3/src/simulatedDrivingAssitance/backend/ConfigEnums.h`. Add a new define for your intervention type. Its value should be one higher than the latest intervention type. Increment the value of `NUM_INTERVENTION_TYPES` by one. Both these steps are demonstrated in Figure 2.61.
Navigate to `source-2.2.3/src/simulatedDrivingAssitance/frontend/IndicatorConfig.h`. In this

file, find the `s_interventionTypeString` array, and add to it the name you assigned earlier in the config file, as demonstrated in Figure 2.62.

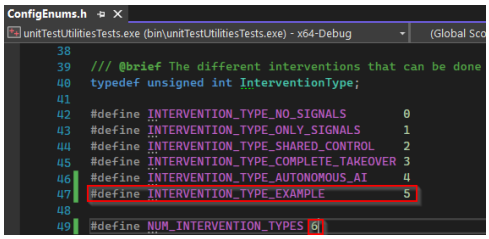


Figure 2.61: Adding a define for the intervention type.

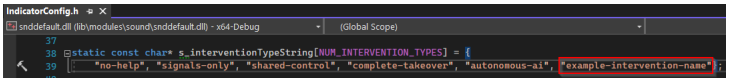


Figure 2.62: Adding the name of the intervention type.

Navigate to `source-2.2.3/data/data/menu/ResearcherMenu.xml`. Find the section named `InterventionTypeRadioButtonList` and add a new section to the section `options`. The name of this section should be the same number as the define you created earlier. In this section, you can add two name-value pairs of type `attstr`: `text` is the displayed name of the intervention type in the Researcher Menu. `tip` is the hint displayed when hovering over the option for this intervention type. This is demonstrated in Figure 2.63.

In the `source-2.2.3/src/simulatedDrivingAssistance/backend` folder, add the files `InterventionExecutorExample.cpp` and `InterventionExecutorExample.h`. These files should also be added to the `CMakeLists.txt` file in this folder, as shown in Figure 2.64. In these files, create a class that inherits from `InterventionExecutor`, and implement your desired intervention method. The header file (shown in Figure 2.65 is fairly simple. Unless you wish to add more data structures that will help in the decision process, this is what the file should look like. In the source file (shown in Figure 2.66, Section 1 is code that gets the set of allowed actions from the `Mediator` and sets the currently displayed images to a neutral image. You could also create the set of allowed actions yourself, but that would ignore the settings in the Researcher Menu. Sections 2 and 3 are specific to this example, and decide how to intervene. Section 4 shows how to enumerate over all decisions, as well as which functions to call to intervene or to indicate an intervention.

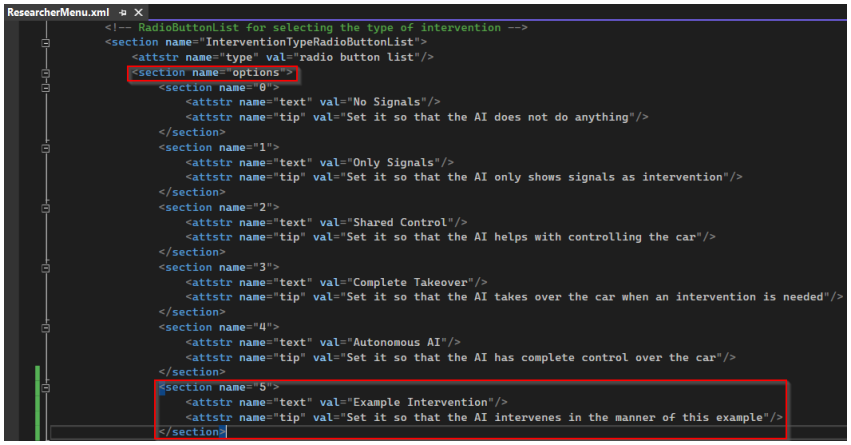


Figure 2.63: Adding the selection option for the new intervention type.

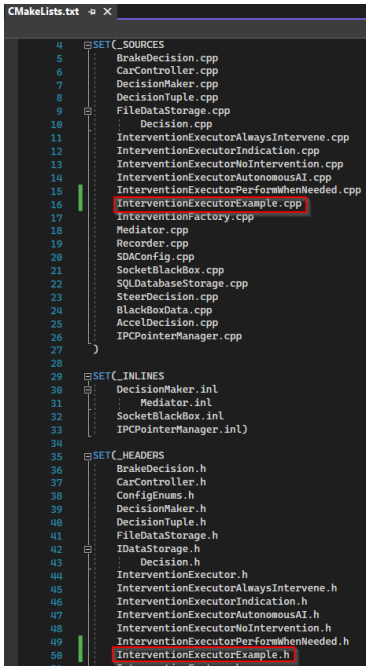


Figure 2.64: Adding the intervention executor files to CMakeLists.txt.

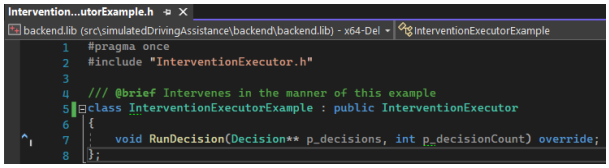


Figure 2.65: Example header file for new intervention executor.

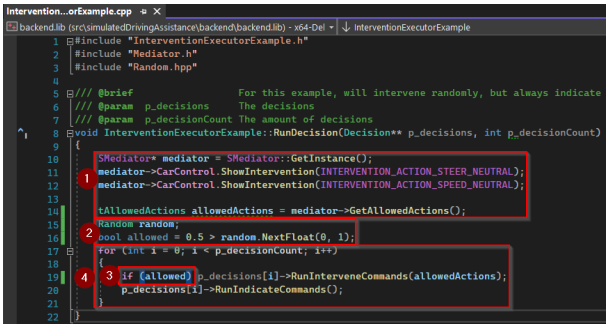


Figure 2.66: Example implementation for new intervention executor.

Navigate to `source-2.2.3/src/simulatedDrivingAssistance/backend/InterventionFactory.cpp`. Include your newly created header file, and add a case for your new executor to the switch statement, as demonstrated in Figure 2.67. Use the newly created define for this.
Navigate to `source-2.2.3/src/simulatedDrivingAssistance/SQLDatabaseStorage.cpp`. Find the `CreateTables` method and add a value for your new intervention type to the ENUM in the statement creating the Settings table, as demonstrated in Figure 2.69. Further find the `InsertMetaData` method and add a case for your new intervention type. Use the define you created earlier, and set values equal to the value you decided on in the ENUM, as demonstrated in Figure 2.68. These last additions are necessary to save the data to the MySQL database. If you already have a `Settings` table, the `ALTER TABLE` statement will update it with the new enum.

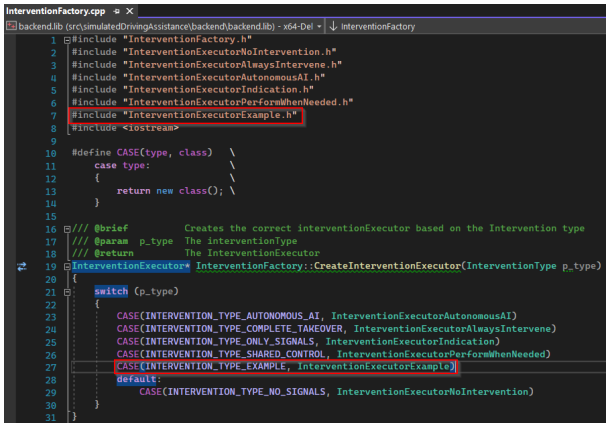


Figure 2.67: Adding a case for a intervention executor to the factory.

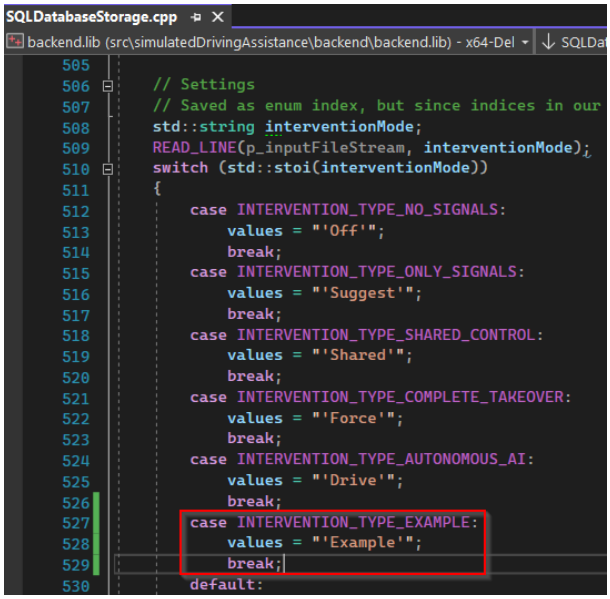


Figure 2.68: Adding a case for the new enum value.

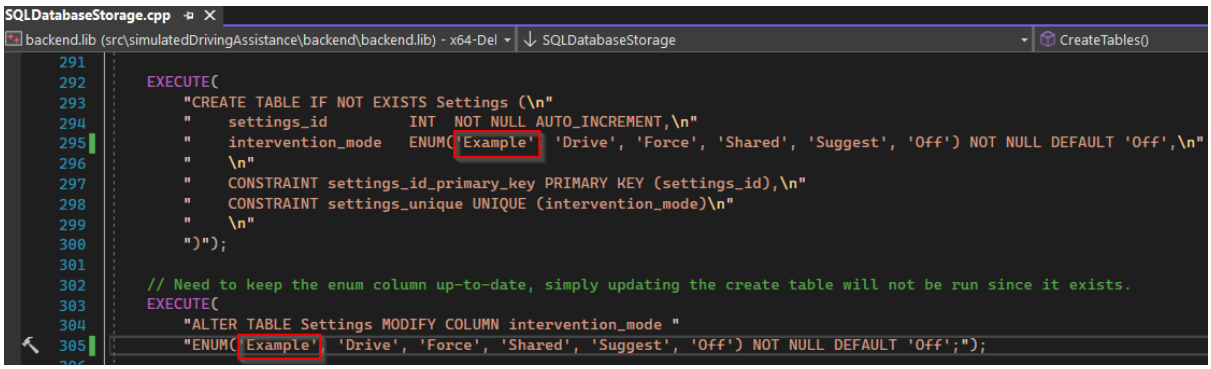


Figure 2.69: Adding an enum value for the new intervention type.

After these steps, your new intervention type should now work. Of course, the newly added intervention type has not been tested yet. You can expand the existing tests for files that you've edited in the `source-2.2.3/src/simulatedDrivingAssistance/backendUnitTests` folder, or add new tests in the `ExecutorTests.cpp` file for the newly added files.

2.1.3 Combining Multiple Decision Makers

It is not possible to have two or more decision makers active at the same time. DAISI would connect with just one of the black boxes. In addition, the order in which interventions are resolved would not be deterministic, but instead dependant on the state of the machine. If you wish to have multiple decision making algorithms influence the simulation concurrently, you will need to create a singular black box.

Create a new decision maker. How you can do this is explained in Sections 2.4 (for C++) and 2.5 (for Python). In your driver's `UpdateAI` method, calculate all actions separately, as you would with a singular decision maker. Afterwards, you must decide on a system to combine these actions. If the algorithms act on different decision types, addition of every corresponding decision will suffice. If there is a conflict, you will need to decide which algorithm's value will have precedence, or the specific algorithm to combine the values (for example, by (weighted) averaging).

2.2 Data From Simulator To AI

In Table 2.1 all data sent to the decision maker from the simulator is shown. It contains all struct members with information about what these attributes entail. Where applicable a reference to another struct is shown. Some attributes are necessary for DAISI to run correctly, hence not all attributes may be useful for your AI. Furthermore, some attributes have been left out of the table completely as they are sure to not be useful for research (e.g. `racenumber`).

`SDAData` **struct members**	**Struct members information**
`Car (tCarElt)`	This is the main car structure and holds all information about the car. More information about this struct can be found in Table 2.2
`Situation (tSituation)`	This is the car its situation. It holds information about the simulation. More information about this struct can be found in Table 2.3
`SimCar (tCar)`	This is the data used to actually simulate the car. More information about the contents of this struct can be found in Table 2.4
`TickCount (unsinged long)`	Amount of simulation ticks since the start of the simulation.

Table 2.1: The members of the SDAData Struct. The complete `SDAData` struct can be found in `SDALib/SDALib-CPP/SDAData.hpp`.

`tCarElt` **struct members**	**Struct members information**
`index (int)`	The car index
`info (tInitCar)`	Holds general information about the car (e.g. `name (string)`, `raceNumber (int)`). The `dimension (t3Dd)` attribute is a 3D vector and holds the cars length, width, and height. The `drvPos (t3Dd)` attribute is a 3D vector and holds the drivers position in the car. The `wheel (tWheelSpec[4])` attribute holds the wheel attributes `rimRadius`, `tireHeight`, `tireWidth`, `brakeDiskRadius`, and `wheelRadius (float)`.
`pub (tPublicCar)`	Holds geometric information about the car (e.g. `speed (float)`) The `DynGC (tDynPt)` and `DynGCg (tDynPt)` attribute are dynamic points. The attribute `pos (3D vector)` in `tDynPt` contains the position. The attribute `vel (3D Vector)` in `tDynPt` contains the velocity. The attribute `acc (3D Vector)` in `tDynPt` contains the acceleration. It holds the global coordinates data; `DynGC` holds the global coordinates data relative to the car axis. `DynGCg` holds the global coordinates data relative to the world axis. The `posMat (sgMat4)` attribute is a 4×4 array containing the position matrix. The `trkPos (tTrkLocPos)` attribute contains information about the current track segment. More info can be found in Table 2.5.
`race (tCarRaceInfo)`	Holds info about the current run (e.g. `curTime (double)`, `wrongWayTime (double)`, `topSpeed (float)`). `distRaced (float)` and `distFromStartLine (float)` contain information about the distance the participant has or should have travelled.
`priv (tPrivCar)`	Contains internal data of the car that is only visible to the driver (e.g. `gear (int)`, The attribute `corner (tPosD[4])` contains the four corners of the car with attributes `x`, `y`, `z`, `xy` (combined 2D xy coordinate), `ax`, `ay`, and `az` (angles)`(float)`.

ctrl (tCarCtrl)	Contains information returned by the driver during the race (e.g. steer (float) [-1.0, 1.0], accelCmd (float) [0.0, 1.0], brakeCmd (float) [0.0, 1.0], clutchCmd (float) [0.0, 1.0]) It has commands for single wheel braking; brakeFrontLeftCmd, brakeFrontRightCmd, brakeRearLeftCmd, and brakeReadRightCmd (float) [0.0, 1.0].
setup (tCarSetup)	Contains the car setup parameters. Each attribute has a value, min, max, and stepsize (float). For each wheel there is a rideHeight, toe, camber, tirePressure, and tireOpLoad.
pitcmd (tCarPitCmd)	Contains information during a pit stop (e.g. fuel).

Table 2.2: The members of the tCarElt Struct. More information can be found in source-2.2.3/src/interface/car.h

tSituation struct members	Struct members information
deltaTime (double)	Time of one time step in the simulation.
currentTime (double)	Current time since the start of the simulation.
raceInfo (tRaceAdmInfo)	Contains information about the current simulation (e.g. totTime (double), fps (unsigned long)). The attribute state (int) contains whether the race is running, finished, ended, starting, prestart, or paused.

Table 2.3: The members of the tSituation Struct. More information can be found in source-2.2.3/src/interfaces/raceman.h

tCar struct members	Struct members information
axle (tAxle)	Holds information about the current state of the front and rear axle of the car and their suspension (e.g. force upon the axles, spring, damping)
wheel (tWheel)	Holds information about the current state of each of the four wheels (e.g. torque, spin, mass, temperature)
steer (tSteer)	Holds the current position of the steering wheel. This can be different from the position the user is trying to put the steering wheel in, as the steering wheel may block or turn slower.
brkSyst (tBrakeSyst)	Holds information about the current state of the braking system inside the car (e.g. brake pressure)
aero (tAero)	Holds information about the current aerodynamic forces acting upon the body of the car (e.g. drag, lift)
wing (tWing)	Holds information about the current aerodynamic forces acting upon the wings of the car (e.g. angle, lift, angle of attack). Only used if the car has a spoiler
transmission (tTransmission)	Holds information about the current state of the transmission, gearbox and clutch (e.g. gear ratio, time until clutch pedal is released, inertia of each gear)
engine (tEngine)	Holds information about the current state of the engine (e.g. output torque, revolutions per second, inertia)

Table 2.4: The members of the tCar Struct. More information can be found in source-2.2.3/src/modules/simu/simuv4/carstruct.h

tTrkLocPos struct member	Information about the struct members
toStart (float)	Distance to the start of the segment

`toRight (float)`	Distance to the right side of the segment (positive when inside of the track, negative when outside of the track)
`toMiddle (float)`	Distance to the middle of the segment (positive to the left and negative to the right)
`toLeft (float)`	Distance to the left side of the segment (positive when inside of the track, negative when outside of the track)
`seg (tTrackSeg)`	Contains information about the current track segment. The attribute `type (int)` shows whether it is a right curve `1 (TR_RGT)`, left curve `2 (TR_LFT)`, or straight `3 (TR_STR)`. The attributes `length (float)` and `width (float)` show the length from the middle of the track segment and width of the track segment. The attribute `lgfromstart (float)` shows the length from the starting point of the experiment to the beginning of the segment. The attributes `radius (double)`, `radiusr (double)`, and `radiusl (double)` show the radius in meters of the middle, right, and left side of the track, respectively. The attribute `arc (double)` shows the arc of the curve of the road if applicable. The attribute `center (3D vector)` shows the coordinate of the center of the curve and `vertex (3D vector[4])` shows the coordinates of the four corners of the segment. The attribute `speedLimit (float)` shows the speed limit of the current segment in km/h.

Table 2.5: The members of the tTrkLocPos struct. More information can be found in `source-2.2.3/src/interfaces/track.h`

2.3 Data From AI To Simulator

In Table 2.1 all data sent to the simulator from the decision maker is shown. It shows the range for each attribute, as well as whether the simulator expects a continuous or discrete value.

`SDAAction` **struct members**	**Range**	**Information about the struct members**
`Steer (float)`	`[-1 (left), 1 (right)]`	The continuous steering value.
`Brake (float)`	`[0 (no brake), 1 (full brake)]`	The continuous braking value.
`Gear (int)`	`[-1 (gear lower), 0 (no gear switch), 1 (gear higher)]` different gear values	The discrete gear value.
`Accel (float)`	`[0 (no gas), 1 (full gas)]`	The continuous accelerate value.

Table 2.6: The members of the SDAActopm Struct. The complete `SDAData` struct can be found in `SDALib/SDALib-CPP/SDAAction.hpp`.

2.4 Create C++ AI

The Simulated Driver AI Library, SDALib for short, can be used to create an AI in C++, that is compatible with the Simulated Driver Assistance software. In this guide we will explain how to create a new AI in C++. You are required to have the SDALib repository cloned.

Create a new folder inside `SDALib\SDALib-CPP`. In this guide we will name this folder `ExampleAI`. In this folder, create a `CMakeLists.txt` file, an `ExampleMain.cpp` file, and an `ExampleDriver.h` file. The last two files can be renamed to fit your use. This newly created folder needs to be included in CMake. To do so, open `SDALib-CPP\CMakeLists.txt`. Here we will add `add_subdirectory("<name of folder>)"`.

The `ExampleAI\CMakeLists.txt` needs CMake code as well. We will not be going into detail about this, but below is shown what needs to be added. Note that every case of `ExampleAI` needs to be changed to the name of your folder. The `ExampleMain.cpp` and `ExampleDriver.h` also need to be renamed to your file names.

```
project(ExampleAI)

add_executable( ExampleAI
        "ExampleDriver.h"
        "ExampleMain.cpp")

target_link_libraries(
        ExampleAI
        SDALib
)

IF(UNIX)
    TARGET_LINK_LIBRARIES(ExampleAI rt)
ENDIF(UNIX)

target_include_directories(ExampleAI PUBLIC "../SDALib-CPP")

IF(WIN32)
    add_custom_command(TARGET ExampleAI POST_BUILD
            COMMAND ${CMAKE_COMMAND} -E copy
            "${IPCLIB_BIN_DIR}/IPCLib.dll" $<TARGET_FILE_DIR:ExampleAI>
            COMMAND_EXPAND_LISTS
    )
ENDIF(WIN32)
```

Now we are ready to create our AI. Create a new class in `ExampleDriver.h`. We will call this class `ExampleDriver` and it will derive from `SDADriver`. This means we inherit functions from `SDADriver`. The important functions are `Run()`, `InitAI()` and `UpdateAI(SDAData p_data)`. `Run()` will be called to start our AI. It will connect to DAISI and call `InitAI()`, where your AI can be initialized. During a run in DAISI `UpdateAI(SDAData p_data)` will be called. This function allows your AI to create an `SDAAction` struct, which will be explained later. You will have to implement `InitAI()` and `UpdateAI(SDAData p_data)` by overriding them, as shown in the code snippet below. The `Run()` function does not need to be changed.

```
#pragma once
#include "SDADriver.hpp"

class ExampleDriver : public SDADriver
{
    void InitAI() override
    {

    }

    SDAAction UpdateAI(SDAData &p_data) override
    {
        SDAAction action;
        return action;
    }
};
```

In this guide we will explain how to create an AI that will press the brake when the car is going faster than 100 km/h. This is a very simple example, but it will explain the basics of using SDALib.

In the `UpdateAI(SDAData& p_data)` function we can define the logic of the AI. This function will be called every time that DAISI asks for input from the AI. The `SDAData` contains all the simulation data from DAISI. This includes the speed of the car, which we will be using. More details about this data can be found in 2.2.

You may have noticed this function returns an `SDAAction` struct. This is a data structure used to set the actions of the AI. More details can be found in 2.3. It is important to know that each value in an SDAAction has a default value of 0. In the DAISI software, a value of 0 will not affect the car. This means nothing will happen if you return an SDAAction without changing its values.

To figure out how fast the car is going, we can look at the `p_data.Car.pub.DynGC.vel.x`. This is the speed of the car in metres per seconds. We will convert this to kilometres per hour by multiplying the value with 3.6. If this value is greater than 100, we will press the break. We can do this by setting `action.Brake` to 1.

```
SDAAction ExampleDriver::UpdateAI(SDAData& p_data)
{
    SDAAction action;

    if (p_data.Car.pub.DynGC.vel.x * 3.6f > 100)
    {
        action.Brake = 1;
    }

    return action;
}
```

Our AI is now functional, however, it is never initialized. In `ExampleMain.cpp` we will create a `main()` function. Here we can create our `ExampleDriver` and call the `Run()` function.

```
#include "ExampleDriver.h"

int main()
{
    ExampleDriver exampleDriver;
    exampleDriver.Run();

    return 0;
}
```

To test your AI in DAISI you need to build your project. A guide about loading a blackbox can be found in the DAISI User Guides. The .exe file is located in `...\cmake-build-release\<your folder name>`.

You can also create more involved AIs, that can plan ahead and react based on possible future events. This can be done by calling the `UpdateSimulator(const SDAData p_data, SDAAction p_action)` function from `SDASpeedDreams.h`. This function returns the `SDAData` of the simulation after `p_action` was done. In the code snippet below, an example is shown on how to call this functions.

```
#include "SDASpeedDreams.h"

SDAAction ExampleDriver::UpdateAI(SDAData& p_data)
{
    SDAAction action;
    SDASpeedDreams sdaSpeedDreams;

    SDAData newSDAData = sdaSpeedDreams.UpdateSimulator(p_data, action);

    return action;
}
```

2.5 Create Python AI

The Simulated Driver AI Library, SDALib for short, can be used to create an AI in Python that is compatible with the Simulated Driver Assistance software. In this section we will explain how to create a new AI in Python, and how to use this AI in the SDA software. You are required to have SDALib installed.

We will be working in the `SDALib-Python` directory. An example of a python decision maker can be found in `Driver.py`. You are free to change this example or create your own file in this folder. When creating your own file make sure that the `m_pythonDriverFileName` variable in `PythonDriver.h` is updated to the name of your new file and update the `INSTALL_PYTHON_FILES` CMake macro in `SDALib-CPP/cmake/SDAMacros.cmake` to install your new file by adding:

```
add_custom_command(TARGET ${TARGET_NAME} POST_BUILD COMMAND ${CMAKE_COMMAND} -E copy
"${CMAKE_SOURCE_DIR}/SDALib-Python/<your file name>.py"
$<TARGET_FILE_DIR:${TARGET_NAME}> COMMAND_EXPAND_LISTS )
```

In the file you decided to work with make sure you `import SDATypes`. This module contains the Python version of the datastructure of the `SDAData` struct. The complete `SDAData` struct can be found in 2.2 or `SDATypes.py`. This way, your Python module can collect all data from the simulator. To be able to communicate with the simulator it expects an `SDAAction` struct which can be found in 2.3.

In your driver file, create a class named `SDADriver`. In the class create at least the following two functions:

1. `__init__(self)` which returns nothing. This function is called as soon as the experiment is starting. Here, you should set up your decision maker.
2. `UpdateAI(self, sdaData)` which returns the four tuple `SDAAction`. This function is called every time `SDALib` receives data from the simulator. Here, you should return any interventions the AI should do like so:

   ```
   return SDAAction(<steer value>, <accel value>, <brake value>, <clutch value>)
   ```

   All data from the simulator can be found in the `sdaData` variable.

It is important that you do not change the names of these functions, class or the number of parameters, as this is what the `PythonDriver` class in C++ is searching for when calling your AI. If you require this to be changed, make sure you update the `UpdateAI` and `InitAI` functions in `PythonDriver.inl`. For the documentation on the Python/C API used in the `PythonDriver` class see https://docs.python.org/3/c-api/index.html.

Now that you have finished creating your decision maker, your code is ready to be build and to be connected to the simulator. For the latter, start up the simulator and follow the loading black box guide in the user guides. Your AI can be found in the build folder at `SDALib-Python/SDALib-Python.exe`.

2.5.1 Calling Simulator Functions

To call functions from the simulator find the files `SDALib-Python/SpeedDreamsPython.cpp` and `SDALib-Cpp/SDASpeedDreams.cpp`. These files allow you to create a Python module from C++ code. In `SDALib-Python/SpeedDreamsPython.cpp`, the `simulator_update` (do not change the name of this function, otherwise the module will not be able to build) receives three arguments from your Python file in `p_args`. The data (SDAData), the action (SDAAction), and the new data (SDAData) you want to receive. There, you can add a call to `SDASpeedDreams` (for example `UpdateSimulator(SDAData, SDAAction)`). In `SDASpeedDreams.cpp` you can add the functions you want to call from the simulator. When creating the simulator module, remove all build files and reload CMake, then build again. Automatically, it should create the new simulator module.

To add this call to your Python file make sure to `import simulator` module. To make sure this module can be found add `sys.path.append(site.getsitepackages())` to your file before importing, make sure to `import site` and `import sys`. This Python module communicates with the simulator using the

`update(SDAData, SDAAction, SDAData)` function, which expects an `SDAData` struct and an `SDAAction` struct as input. The last `SDAData` struct will be the updated struct. This should be a valid `SDAData` struct. For example, to update the simulator in your `UpdateAI(self, sdaData)` function add the following lines:

```
sdaAction = SDAAction(<steer value>, <accel value>, <brake value>, <clutch value>)
newSDAData = sdaData
simulator.update(sdaData, sdaAction, newSDAData)
```

The `newSDAData` variable contains the updated `SDAData` struct.

2.6 Adding New AI Languages

It is possible to add new decision maker language support to SDALib. This section will provide a global overview of the steps necessary to be taken for SDALib to accept the language of your choice by creating a bridge between your language and C++. This bridge is the easiest way to add new language support due to the way that DAISI shares its memory with the `IPCPointerManager` class. By creating the bridge, you do not have to worry about establishing connections with the simulator itself. This guide will not be going in to depth about certain functions as most of it is language dependent. We will not go into depth about how you should update CMake to build all your files correctly. In SDALib there is a reference project that implements this bridge for Python, you can find it in the `SDALib-CPP/SDALib-Python` folder.

DISCLAIMER: It will take some time, effort, and patience to add this support. So make sure you are absolutely certain you need this functionality before you start working on it.

First and foremost, decide on which language you would want to integrate and find a library that would allow you to communicate between C++ and the language of your choice (e.g. for Python https://docs.python.org/3/c-api/index.html). Add the required library to `SDALib` using CMake.

In `SDALib-CPP` create a new folder `SDALib-<your language>`. You are required to create two classes:

1. A `SDATypesConverter` class that translates the `SDAData` and `SDAAction` struct to a struct in your language and back (e.g. for Python it translated to an `SDATypes.py` module in a `PyObject` data structure).

2. A `<your language>Driver` that connects with your decision maker. This class should inherit from `SDADriver` and override the `InitAI()` (which connects with your decision maker and initializes your type converter) and `UpdateAI()` (which should convert the `SDAData` to the struct in your language, calls the functions from your decision maker, and translates the output back to `SDAAction`) functions. Make sure to add a `main()` function that is the main entry point of the driver.

   When inheriting from `SDADriver`, `SDALib` will automatically register your decision maker as one in your language when inserting the decision maker in DAISI. `PythonDriver` has inherited from `AIInterface<PointerManager>`, such that it can run unit tests. If this is not a necessity for you, we recommend inheriting from `SDADriver`.

If you require to call functions from the simulator, you need to find a way on how to call a C++ function from your language (e.g. for Python it is necessary to create a module (`setup.py`) from a C class (`SpeedDreamsPython.cpp`) using the previously mentioned library. The module should then be imported in the `driver.py` file). The function you need to call is located in `SDALib-CPP/SDASpeedDreams`. It requests an `SDAData` and `SDAAction` struct and returns an updated `SDAData` struct.

Chapter 3

Saving Data

3.1 Changing Database Data

This section will explain how to change the data that is saved to the database by DAISI. It will explain what parts of the system have to be modified to change the data that will be saved to the database.

3.1.1 Data buffer files

Before data is saved to the database, it is first written to a buffer file during the simulation. This is done because constant communication with a database can be slow. Writing of this buffer file is done in the `FileDataStorage` class. One buffer file is used for the metadata (data that is only saved once per trial, such as the participant's ID or the intervention settings), timesteps (which simply includes a list of every tick). The tables TimeStep, Intervention, UserInput and GameState each use their own buffer file.

Data saved every time step is saved using the `Save` function after compression (for more information on data compression, see section 3.2). This function writes the time step to its own buffer file and then triggers writing to every other buffer file if required.

To make it easy to read a buffer file after writing it, every attribute value is written separated by a comma and every time step is placed on a new line.

To add a new attribute to the buffer file, write the attribute by appending it to the list of comma-separated values using the `WRITE_CSV` macro.

To add a new table that will be saved once per tick, first create a buffer file for the new table by opening a filestream in the `Initialize` function and storing it as a member variable of the class. Then, add a function to write the comma-separated values to the filestream and call it in the `Save` function.

3.1.2 MySQL storage

Any functionality relating directly to the MySQL database is located inside the `SQLDatabaseStorage` class. In this class, two functions are responsible for inserting data into the database: `InsertMetaData` handles the storing of the metadata as explained above, and `InsertSimulationData` handles the storing of data that is saved once per tick (such as the user input or interventions made during the trial).

To add, change or remove data saved only once for a simulation, modify the queries inside the function `InsertMetaData`. These queries are written either through a direct MySQL statement or a prepared statement (as documented in the MySQL documentation: `https://dev.mysql.com/doc/refman/8.0/en/sql-prepared-statements.html`) and can easily be changed provided you are experienced with MySQL queries.

Adding, changing or removing data saved every tick is more complicated. To ensure a swift speed when uploading data to the database, the `LOAD DATA LOCAL INFILE` statement is used. This means the data is inserted not through individual SQL statements, but read directly from the buffer file according

to the format specified in the statement. Adding or removing new attributes to one of the existing tables is as simple as updating the structure of the buffer file and changing the `LOAD DATA LOCAL INFILE` statement accordingly. Adding a new table is more complicated; this requires creating a new buffer file for the table and writing an additional `LOAD DATA LOCAL INFILE` statement for it.

3.1.3 List of attributes and variables associated with them

Currently, the following data is saved by using the following DAISI variables:

Table	Attribute	DAISI variable	Notes
Blackbox	file	SDAConfig:: GetBlackBoxFilePath().filename()	
	version	filesystem::last_write_time(SDAConfig::GetBlackBoxFilePath())	
	name	SDAConfig:: GetBlackBoxFilePath().stem()	
Environment	file	tTrack::filename	
	version	tTrack::version	
	name	tTrack::name	
Participant	participant_id	SDAConfig :: GetUserId()	
Trial	trialtime	std::chrono::system_clock::now()	
Settings	mode	SDAConfig::GetInterventionType()	Translated to a string using a switch statement in SQLDatabaseStorage
TimeStep	tick	Mediator::m_tickCount	
AccelDecision	amount	DecisionTuple::GetSteer() or SKIP_DECISION	SKIP_DECISION if no decision of this type was made.
SteerDecision	amount	DecisionTuple::GetBrake() or SKIP_DECISION	SKIP_DECISION if no decision of this type was made.
BrakeDecision	amount	DecisionTuple::GetAccel() or SKIP_DECISION	SKIP_DECISION if no decision of this type was made.
UserInput	steer	tCarCtrl::steer	
	brake	tCarCtrl::brakecmd	
	gas	tCarCtrl::accelCmd	
	clutch	tCarCtrl::clutchCmd	
GameState	x	CarElt::pub.DynGCg.pos.x	
	y	CarElt::pub.DynGCg.pos.y	
	z	CarElt::pub.DynGCg.pos.z	
	direction_x	CarElt::pub.DynGCg.pos.ax	
	direction_y	CarElt::pub.DynGCg.pos.ay	
	direction_z	CarElt::pub.DynGCg.pos.az	
	speed	CarElt::pub.DynGC.mov.vel.x	
	acceleration	CarElt::pub.DynGC.mov.acc.x	
	gear	CarElt::priv.gear	

Table 3.1: Database attributes and their DAISI variables

3.2 Changing Data Compression

In the data compression menu the amount of compression on the data can be easily changed as explained in the user guide. This section focuses on how to change the data compression method for each variable. Table 3.2 contains the original compression methods for each saved variable.

In `src/simulatedDrivingAssistance/backend/FileDataStorage.h` and `src/simulatedDrivingAssistance/backend/FileDataStorage.cpp` the data is saved to the buffer file before being sent to the database. Here the compression takes place and all methods and variables discussed in this section can be found in those files.

Each time step the `FileDataStorage::Save` function is called. Here all car, human, and intervention data is saved in data structures (using the `SaveCarData()`, `SaveHumanData()`, and `SaveInterventionData()` methods). After the requested amount of time steps (which is set in the data compression menu), the data collected in the data structures is compressed and saved to the buffer file (using the `WriteCarData()`, `WriteHumanData()`, and `WriteInterventionData()` methods).

Data Set	**Variable Name**	**Variable Type**	**Variable Range**	**Method**	**Explanation**
Car	Position	Smooth	$(-\infty, \infty)$	Averaging	Smooth variable, so averaging
	Rotation	Smooth	$(-\infty, \infty)$	Averaging	Smooth variable, so averaging
	Speed	Smooth	$(-\infty, \infty)$	Averaging	Smooth variable, so averaging
	Acceleration	Smooth	$(-\infty, \infty)$	Averaging	Smooth variable, so averaging
	Gear	Discrete	$[-1, 8]$	Least Common	A discrete variable, this will use the least common method as gear shifting happens really fast. If we would use median or sampling the gear shift would most likely be missed.
Human	Steer	Non-Smooth	$[-1, 1]$	Median	A non-smooth variable if a participant is reacting to a situation, so median.
	Brake	Non-Smooth	$[0, 1]$	Median	A non-smooth variable if a participant is reacting to a situation, so median.
	Acceleration	Non-Smooth	$[0, 1]$	Median	A non-smooth variable if a participant is reacting to a situation, so median.
	Clutch	Non-Smooth	0 or 1	Median	A non-smooth variable if a participant is reacting to a situation, so median.
Decision	Steer	Non-Smooth	$[-1, 1]$	Median	Depends on the black box, but we will see this as a non-smooth variable, so median.
	Brake	Non-Smooth	$[0, 1]$	Median	Depends on the black box, but we will see this as a non-smooth variable, so median.
	Acceleration	Non-Smooth	$[0, 1]$	Median	Depends on the black box, but we will see this as a non-smooth variable, so median.
	Gear	Discrete	-1, 0 or 1	Least common	A discrete variable, this will use the least common method as gear shifting happens really fast. If we would use median or sampling the gear shift would most likely be missed.

Table 3.2: Data Compression

3.2.1 Changing the compression method

The compression method can be changed for each variable. Currently implemented is compression using averaging, median, and least common. To implement a new form of compression see section 3.2.2.

Averaging

The current averaging method is $O(1)$. It adds the value of this time step to the total collected in the previous time steps for that variable. After the requested amount of time steps for compression (the compression rate) this total is divided by the compression rate and saved to the buffer file. The current total is set to zero for the next compression step.

To change the compression method of an attribute from the car or human data set to averaging the following steps are necessary to be taken. In this example we are going to change the steer value compression method of the human data set from median to averaging:

1. In `FileDataStorage.h`, add a new data structure (recommended: float) which should collect the total and set this value to zero: (e.g. `float m_totalHumanSteer = 0`). Make sure this data structure is also initialized in the `Initialize()` function in `FileDataStorage.cpp`
2. In `FileDataStorage.cpp`, find the method where this attribute is collected: (e.g. `SaveHumanData()`).
3. In the method found in step 2, find the line where the value is saved and change it to `AddForAveraging([the data structure containing the total declared in step 1], [the variable from the current time step])`:
   (e.g. `AddToArray<float>(m_steerValues, ctrl.steer, m_compressionStep)` to `AddForAveraging(m_totalHumanSteer, ctrl.steer)`).
4. In `FileDataStorage.cpp`, find the method where this attribute is written to the buffer file:
   (e.g. `WriteHumanData()`).
5. In the method found in step 4, find the line where the attribute is written and change the current compression method to the new one, namely `GetAverage([the data structure containing the total declared in step 1])`: (e.g. `WRITE_VAR(m_outputStream, GetMedian(m_steerValues))` to `WRITE_VAR(m_outputStream, GetAverage(m_totalHumanSteer))`)
6. Now the values will be compressed with the averaging method.

To change the compression method of an attribute from the decision data set to averaging extra steps are necessary to be taken.

1. A new function should be declared in `FileDataStorage.h` and created in `FileDataStorage.cpp` that gets the average of an array (similar to the `Getmedian` function).
2. In the `WriteInterventionData()` function, change the correct line to call the function described in step 1 instead of `GetMedian` or `GetLeastCommon`.

Median

The current median method uses randomized quick select. [1] This method has an average of $O(n)$. [2] It adds the value of this time step to an array with a maximum size corresponding to the maximum compression rate of 49 to the correct place in the array (for the first time step in the current compression step it goes to place 0, for the second time step in the current compression step to place 1, etc.). After the requested amount of time steps for compression the median is found using randomized quick select.

To change the compression method of an attribute to median the following steps are necessary to be taken. In this example we are going to change the gear value compression method of the car data set from least common to median:

1. In `FileDataStorage.h`, check whether there already exists an array for the attribute you want to compress using the median. If yes, which is the case for the gear values, go to step 4.

[1] Randomized quick select: https://www.geeksforgeeks.org/median-of-an-unsorted-array-in-liner-time-on/
[2] Proof average of $O(n)$: https://eugene-eeo.github.io/blog/randomized-quickselect.html

2. In `FileDataStorage.h`, add a new datastructure (recommended: array of size `COMPRESSION_LIMIT`) to collect the values of the time steps in the current compression step:
(e.g. `int m_gearValues[COMPRESSION_LIMIT]`).

3. In `FileDataStorage.cpp`, find the method where the attribute is collected (e.g. `SaveCarData()`).

4. In the method found in step 2, find the line where the value is saved and change it to `AddToArray<[type]>([the data structure containing the total declared in step 1], [the variable from the current time step], m_compressionStep)`: (for gear values this is already correct, as there already exists an array for the attribute).

5. In `FileDataStorage.cpp`, find the method where the attribute is written to the buffer file: (e.g. `WriteCarData()`).

6. In the method found in step 5, find the line where the attribute is written and change the current compression method to the new one, namely `GetMedian([the array containing all values])`:
(e.g. `WRITE_VAR(m_outputStream, GetLeastCommon(m_gearValues))` to
`WRITE_VAR(m_outputStream, GetMedian(m_gearValues))`.

7. Now the values will be compressed with the median method.

Least Common

The current least common method is $O(n)$. It adds the value of the attribute of the time step to an array with a maximum size corresponding to the maximum compression rate of 49 to the correct place in the array (for the first time step in the current compression step it goes to place 0, for the second time step in the current compression step to place 1, etc.). After the requested amount of time steps for compression, the least common value is found by counting the frequencies of all values in an array and returning the value with the lowest frequency.

To change the compression method of an attribute to the least common the following steps are necessary to be taken. In this example we are going to change the x position attribute compression method of the car data set from average to least common:

1. In `FileDataStorage.h`, check whether there already exists an array for the attribute you want to compress using the median. If yes, go to step 4.

2. In `FileDataStorage.h`, add a new datastructure (recommended: private array of size `COMPRESSION_LIMIT`) to collect the values of the time steps in the current compression step:
(e.g. `int m_posXValues[COMPRESSION_LIMIT]`).

3. In `FileDataStorage.cpp`, find the method where the attribute is collected (e.g. `SaveCarData()`).

4. In the method found in step 2, find the line where the value is saved and change it to `AddToArray<int>([the data structure containing the total declared in step 1], [the variable from the current time step], m_compressionStep)`:
(e.g. `AddToArray<int>(m_posXValues, pos.x, m_compressionStep)`).

5. In `FileDataStorage.cpp`, find the method where the attribute is written to the buffer file: (e.g. `WriteCarData()`).

6. In the method found in step 5, find the line where the attribute is written and change the current compression method to the new one, namely `GetLeastCommon([the array containing all values])`: (e.g. `WRITE_VAR(m_outputStream, GetAverage(m_totalPosX))` to
`WRITE_VAR(m_outputStream, GetLeastCommon(m_posXValues))`.

7. Now the values will be compressed with the least common method.

(Note: the least common method is only for attributes with integer values)

3.2.2 Adding a compression method

It is possible to use a compression method different from the already implemented averaging, median, and least common. To add a new compression method it is necessary to perform the following steps:

1. Find a compression method you want to use with the according data structure to save all values in the current compression step.
2. In `FileDataStorage.h`, add the following methods/attributes:
   (a) An attribute containing the data structure to save all values in
   (b) A void function which saves the attribute values for each time step with as input at least the previously defined attribute and the value of the attribute of the current time step.
   (c) A function which calculates the compressed value using your compression method and returns the correct value with as input at least the previously defined attribute (Note: Make sure that after each compression step, the current values in the previously defined attribute are reset or overwritten in the next compression step).
3. In `FileDataStorage.cpp`, implement the functions from step 2 according to your compression method.
4. In `FileDataStorage.cpp`, find the method where the values of the attribute you want to compress with your new compression method is collected (in the same manner as described in section 3.2.1.
5. In the method found in step 4, find the line where the value is saved and change the method to your method defined in step 2b with the attribute defined in step 2a.
6. In `FileDataStorage.cpp`, find the method where the compressed value is written to the buffer file (in the same manner as described in section 3.2.1.
7. In the method found in step 6, find the line where the attribute is written and change the current compression method to your method defined in step 2c with the attribute defined in step 2a.
8. Now the value will be compressed with your new compression method.

3.2.3 Failing Tests

Changing the compression method might lead to failing tests in `src/simulatedDrivingAssistance/backendUnitTests/FileDataStorageTests.cpp`. To solve this issue find the `TestDataStorageSaveCompressionRates` function. Here, add the data structure you defined, change the saving function for the variable you changed the compression method from, and change the compression function.

3.3 Saving Internal Meta Of AI

When you want to save the internal meta data of your AI, you are required to add the desired functionality to your own code as the internal meta data is dependent on your AI. In this section we explain how to add this functionality to a C++ and Python decision maker. You are required to have SDALib installed and your decision maker should be located there as well. This is because all the required MySQL libraries have been installed here. If you prefer to work outside of SDALib, make sure MySQL is installed.

Before adding this functionality you are required to know basic MySQL query knowledge, what data you want to save and where this data can be found in your AI, what your table will look like, and the credentials of the database you want to save it to.

Saving the internal metadata happens in two stages. The first stage is the real-time saving of the metadata. This is done by writing the data to a buffer file. The second stage is the inserting of data from this buffer file, into a database. This happens at the end of a run. We cannot save the metadata in real-time directly to a database, because SQL queries are simply too slow.

3.3.1 Saving to the Buffer File

In the first stage, you have to write the data you want to save to a buffer file. You can write the data to any file you want. However, we have set up a system within DAISI that might be useful for you. By writing to `..\AppData\Local\Temp\blackbox_internal_metadata_buffer.bin`, you can link the AI's metadata to the trial saved by the DAISI software. In short: when you decide to save the data in DAISI, the new trial ID will be written to the aforementioned file. DAISI will overwrite the first four bytes of the file with the trial ID.

First we will explain how to create this buffer file in C++, and write the first four bytes to this file. Then we will explain how to write the data you want to save to this buffer file in C++ as well as in Python.

Opening the Buffer File

Start by including `<experimental/filesystem>`. This can be used to get the path to the temporary folder on both Windows and Linux with the `temp_directory_path()` function. Using `std::ofstream` an output file can be opened. This file can be opened with a variety of modes. The `std::ios::binary` mode is needed for us, since we will be writing an integer value as bytes. The code snippet below shows how this could be done.

```
#include <fstream>
#include <experimental/filesystem>

// create filesystem namespace for easier-to-read code
namespace filesystem = std::experimental::filesystem;

void InitialiseBufferFile()
{
    filesystem::path bufferPath = filesystem::temp_directory_path();

    std::ofstream bufferFile(bufferPath, std::ios::binary);
}
```

DAISI will overwrite the first four bytes with the trial ID. In our use-cases we wanted to be able to check if DAISI has written to the file or not. This would be an indication to save the data. Therefore we first write `INT_MAX` as bytes to the file, as shown in the code snippet below. This will allow us to later check if the first four bytes are still `INT_MAX`, in which case we do not want to save the data, as the data can then not be linked to the run experiment.

```
void InitialiseBufferFile()
{
    // ..after opening the buffer file

    // Convert INT_MAX to array of characters
    bufferFile.write(reinterpret_cast<const char*>(INT_MAX), sizeof(INT_MAX));

    file.flush();
    file.close();
}
```

The `InitialiseBufferFile()` function can be called before calling the `Driver`'s `Run()` function, or in the `InitAI()` function.

Saving Data in C++

The previously explained steps can be used to open the buffer file. Include the `std::ios::app` mode when opening the file, as this will ensure the file content is retained. The `std::ofstream` can be a private member variable of your `Driver`. This allows you to write to the file in every `UpdateAI()`. Opening the buffer file can be done in `InitAI()`. In the code snippet below, an example is shown on how to save the internal meta data to the buffer file.

```
void InitAI()
{
    filesystem::path bufferPath = filesystem::temp_directory_path();

    // Append buffer file name to temp directory path
    bufferPath.append("blackbox_internal_metadata_buffer.bin");

    std::ofstream bufferFile(bufferPath, std::ios::binary | std::ios::app);
}

void UpdateAI(SDAData& p_data)
{
    // After doing your calculations

    bufferFile << <data point of variable 1>
               << <data point of variable 2>
               ...
               << <data point of variable n>

    file.flush();
    file.close();
}
```

Saving Data in Python

To save the internal meta data of a Python decision maker, you are required to have followed the previous steps in the guide to ensure that the buffer file is created and saved to the database correctly. After that is implemented and works correctly, you only have to ensure that the data is saved to the buffer file, which will be explained in this section.

In the Python file that contains the data you want to save, you must open the buffer file and write your data. The data should be saved to the buffer file in the order you want the buffer file to be read when saving the data after the experiment is finished. This means, the first data that is written to the file will be the first to be processed in the MySQL queries.

To open the buffer file import `os` and add the following lines to the function containing the data you want to save:

```
f = open(os.path.join(os.getenv('APPDATA'),
         'Local\\Temp\\blackbox_internal_metadata_buffer.bin'), 'w')
```

After opening the file, you can write all data points to the buffer file. Make sure to write it in the same order as that the buffer file is read out after completing the run (so in the same order as stated in your database tables) and use separate statements for each data point, like so:

```
f.write(<data point variable 1>)
f.write(<data point variable 2>)
...
f.write(<data point variable n>)
```

After writing the data in your function, make sure you close the file by adding:

```
f.close()
```

At the end of each run, SDALib will then read this buffer file and translate the information to MySQL queries to save it to the database.

3.3.2 Saving to a Database

This section explains how the internal meta data from the buffer file is saved to the database. To save the data from the buffer file to a database, the MySQL Connector/C++ library can be used. This library is

pre-installed on SDALib. For this stage, you only have to work in C++, no matter what language your AI has been written in.

When writing data to the buffer file is finished, you can start saving it to the database. First, your `Driver` class needs to include the library's header. The following code can be used on Windows, as well as on Linux:

```
#ifdef WIN32
    #include "mysql/jdbc.h"
#else
    #include "mysql_connection.h"
    #include "mysql_driver.h"
    #include "mysql_error.h"
    #include "cppconn/build_config.h"
    #include "cppconn/config.h"
    #include "cppconn/connection.h"
    #include "cppconn/datatype.h"
    #include "cppconn/driver.h"
    #include "cppconn/exception.h"
    #include "cppconn/metadata.h"
    #include "cppconn/parameter_metadata.h"
    #include "cppconn/prepared_statement.h"
    #include "cppconn/resultset.h"
    #include "cppconn/resultset_metadata.h"
    #include "cppconn/statement.h"
    #include "cppconn/sqlstring.h"
    #include "cppconn/warning.h"
    #include "cppconn/version_info.h"
    #include "cppconn/variant.h"
#endif
```

Then, add an `sql::Driver`, an `sql::Connection` and an `sql::Statement` as private member variables to your `Driver`. Their usage will be explained later. Furthermore, add the public function `SaveToDatabase()` and the private functions `ConnectToDatabase()`, `InsertData()`, and `CloseDatabase()` (see the code snippet below). We will implement these one by one.

```
// Inside the ExampleDriver.h file
#include "mysql/jdbc.h"

class ExampleDriver : SDADriver
{
public:
    void SaveToDatabase();

private:
    sql::Driver* driver;
    sql::Connection* connection;
    sql::Statement* statement;

    void ConnectToDatabase();
    void InsertData();
    void CloseDatabase();
}
```

We will start by implementing the `ConnectToDatabase()` function. To connect to a database, the `sql:Driver` needs to be set up using `sql::mysql::get_mysql_driver_instance()`. This will tell the program that we are connecting to a MySQL database. An `sql::ConnectOptionsMap` can be used to set connection properties. These include the host, username and password, but also other options like the SSL key. The `sql::Connection` can be setup using the `connect()` function. This will allow the

communication between our program and the database. In the example below, we connect to a local database.

```
void ExampleDriver::ConnectToDatabase()
{
    // Initialise SQL driver
    driver = sql::mysql::get_mysql_driver_instance();

    // Set connection options, and connect to the database
    sql::ConnectOptionsMap connection_properties;
    connection_properties["hostName"] = "tcp://localhost";
    connection_properties["userName"] = "root";
    connection_properties["password"] = "PASSWORD";
    connection_properties["port"] = 3306;

    connection = driver->connect(connection_properties);
}
```

A schema needs to be created as well. We can do this inside the `ConnectToDatabase()` function as well. The `sql::Statement` can be used to execute SQL-statements. An `sql::Statement` needs to be created with the `createStatement()` function. This statement needs to be closed and deleted after usage, as it cannot be used within the created schema, and will otherwise create memory leaks. Our `sql:Connection` can be set to use this newly created schema. In the example below we create an `exampleSchema` and connect to it.

```
void ExampleDriver::ConnectToDatabase()
{
    // After connecting to the database

    // Create the database schema if this is a new schema. This has to be done
    // before setting the schema on the connection.
    statement = connection->createStatement();
    statement->execute("CREATE DATABASE IF NOT EXISTS exampleSchema")
    statement->close();
    delete statement;

    // Set the correct database schema
    connection->setSchema("exampleSchema");
}
```

Now, a database table needs to be created. This table will look different, based on what data you want to save. We will create an example table with only two attributes. Here, we create an `sql:Statement` that we can use in our schema. We will not close and delete this `sql:Statement`, because we can reuse it in the `InsertData()` function.

```
void ExampleDriver::ConnectToDatabase()
{
    // After creating and connecting to the schema

    // Create a (reusable) statement
    statement = connection->createStatement();

    // Create an example table
    statement->execute("CREATE TABLE IF NOT EXISTS exampleTable ("
                       "example_id    INT   NOT NULL AUTO_INCREMENT,"
                       "example_value FLOAT NOT NULL,"
                       "CONSTRAINT primary_key PRIMARY KEY(example_id)"
                       ")")
}
```

Next, we can implement the `InsertData()` function. This function will read from the buffer file, and insert the data into the table(s). This can be done using the statement that has been created in the `ConnectToDatabase()` function. First, we will open the buffer file from the same location as explained earlier, see the code snippet below.

```
#include <experimental/filesystem>
namespace filesystem = std::experimental::filesystem;

void ExampleDriver::InsertData()
{
   // Open the buffer file in Temp folder
   filesystem::path bufferPath = filesystem::temp_directory_path();
   bufferPath.append("blackbox_internal_metadata_buffer.bin");

   std::ifstream bufferFile;
   bufferFile.open(bufferFilePath, std::ios::binary);
}
```

Now that the buffer file is opened, we can start reading its data. To check if we want to save the data, we will first read the first four bytes as an integer. If this integer is equal to INT_MAX, we will not save the data. This is shown in the example below.

```
void ExampleDriver::InsertData()
{
   // After opening the buffer file

   // Read trial ID from file
   int trialId = INT_MAX;
   bufferFile.read(reinterpret_cast<char*>(&trialId), sizeof(trialId));

   // We have written INT_MAX to the buffer file meaning,
   // if speed-dreams did not give us the trialID, this will be true,
   // and we don't want to save the data.
   if (trialId == INT_MAX) return;
}
```

If we do want to save the data, we will enter a while-loop, where we will be reading from the buffer file until we reach the end (see the code snippet below). We can use the `sql::Statement` to insert the data into the table.

```
void ExampleDriver::InsertData()
{
   // After checking for INT_MAX

   while (!bufferFile.eof())
   {
      // Read data from buffer file in same order as you wrote data
      bufferFile >> <data point of variable 1>
                 >> <data point of variable 2>
                 ...
                 >> <data point of variable n>

      // Execute insert statement
      statement->execute("INSERT INTO exampleTable (example_value)"
                         "VALUES ('" + std::to_string(<data point of variable 1>) + "')";
   }
}
```

After inserting all the data, we must close the database. Upon closing the database, the `sql::Statement` and `sql::Connection` have to be closed and deleted. If this is not done, it will cause memory leaks.

```
void ExampleDriver::CloseDatabase()
{
    if (statement) {
    statement->close();
    delete statement;
    }

    if (connection) {
    connection->close();
    delete connection;
    }
}
```

Finally, these functions have to be combined. This can be done inside the `SaveToDatabase()` function. This function should be called from within the `main()` function, that is used to run the driver.

```
// The main function
int main()
{
    ExampleDriver exampleDriver;
    exampleDriver.Run();

    // DAISI has disconnected when we get here
    exampleDriver.SaveToDatabase();
}

// Inside ExampleDriver.cpp
void ExampleDriver::SaveToDatabase()
{
    ConnectToDatabase();
    InsertData();
    CloseDatabase();
}
```

This functionality is not set in stone and can be changed to your desire. If you use SELECT-statements and need to read data from the database, `sql::ResultSet` can be used. More documentation about this—and other functionalities—can be found on the MySQL Connector website.

Chapter 4

Configuration

4.1 Configuring Menus

As a developer you may like to change the menus of DAISI. In this section we will explain how to modify the attributes of a menu page, as well as how to add new menu pages. All the images, sound and text of the menus are loaded from XML files.

4.1.1 Hints

Hints are explanations of menu parts which are shown when hovering over the corresponding parts. When hovering over the *Options* button in the start screen a hint can be seen that states "Options for graphics, sound and opponents" (figure 4.1). In this example this hint will be changed to the sentence: "Options for graphics, sound and controls".

Changing hints in menus of DAISI is quite easy. First navigate towards the directory that contains all the menu XML files. The path to this directory is the following: `source-2.2.3/data/data/menu`

Once in this directory, find the `<name of menu>.xml` in the folder and open it with a text editor. In this example the `mainmenu.xml` will be edited in Notepad++. In figure 4.2 the change to `mainmenu.xml` can be seen. The start section contains all the start button setting. The `tip` attribute contains the hint. So for this example it will be changed to: "Options for graphics, sound and controls". This can be seen on the right of figure 4.2. Finally, don't forget to save. Now startup DAISI and the hint should have been saved.

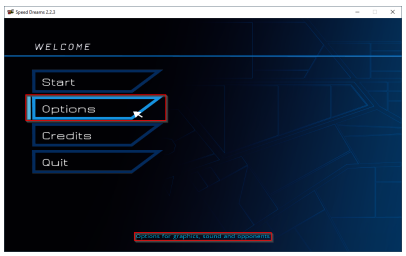


Figure 4.1: Options button hint

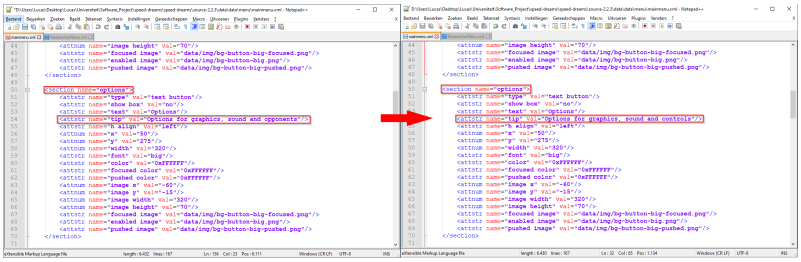


Figure 4.2: Change options hint

4.1.2 Background

To change the backgrounds of DAISI, navigate to `source-2.2.3/data/data/img`. Here, all the backgrounds of DAISI are located (see figure 4.3).

To change the background, simply replace these pictures with the desired image while maintaining the original file name. If renaming the file is desired, make sure to change the XML files accordingly (look for the background image under static controls and then change the image path of the image attribute). For example, when changing the main menu background make sure to update the filename in the `mainmenu.xml` as can be seen in figure 4.4.

Then the final result after changing the hint and the background can be something like figure 4.5.

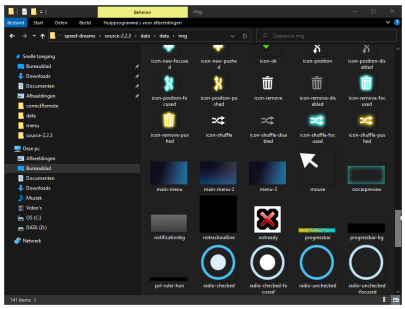


Figure 4.3: Image directory

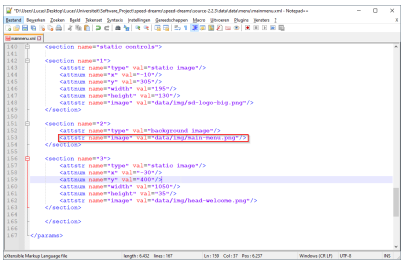


Figure 4.4: Background in XML file

Figure 4.5: Changed menu

4.1.3 Adding Menus

A New File

To add a new menu in DAISI you need to look at the `modules\userinterface\legacymenu` folder. In this folder are three subfolders containing all the .cpp and .h files for the main, config and race menus. Add a new .h and .cpp file for your new menu. Do not forget to add them to the `CMakeList.txt`. The `MenuName.h` file only needs to contain the following:

```
#pragma once

extern void* MenuNameInit(void* p_prevMenu)
```

Add to the top of the `MenuName.cpp` file the following.

```
#include <tgfclient.h>
#include "legacymenu.h"
#include "MenuName.h"
#include "Mediator.h"
#include "guimenu.h"

static void* s_scrHandle  = nullptr;
static void* s_prevHandle = nullptr;
```

Every menu needs an initialization function. This function reads from an XML file how it should look like and what it should contain. The `Init()` function then creates the menu defined there, gives the buttons etc. functionality, and finally returns the menu handle id, which is a unique id that can be used to access the menu.

First the `MenuNameInit()` function checks whether it already exists. If so, it immediately returns the handle id that can be used to access it. If not, it creates a new empty screen, reads the XML file and then starts creating buttons. It also creates keyboard shortcuts. After this, it reads the local saved parameters from another XML file and sets everything internally and on screen to those parameters. Finally it returns the handle id.

```
void* MenuNameInit(void* p_prevMenu)
{
    // screen already created
    if (s_scrHandle)
    {
       return s_scrHandle;
    }

    s_scrHandle = GfuiScreenCreate((float*)nullptr, nullptr, OnActivate, nullptr,
```

```
                                    (tfuiCallback)nullptr, 1);
    s_prevHandle = p_prevMenu;

    void* param = GfuiMenuLoad("MenuName.xml");
    GfuiMenuCreateStaticControls(s_scrHandle, param);
    // add UI button controls here

    GfParmReleaseHandle(param);
    // add keyboard controls here

    // Retrieve the saved local xml file
    char buf[512];
    sprintf(buf, "%s%s", GfLocalDir(), "config/MenuName.xml");
    if (GfFileExists(buf))
    {
        void* param = GfParmReadFile(buf, GFPARM_RMODE_STD);
        // Initialize settings with the retrieved xml file
        LoadConfigSettings(param);
        GfParmReleaseHandle(param);
        return s_scrHandle;
    }
    LoadDefaultSettings();

    return s_scrHandle;
}
```

Every menu also needs an `OnActivate()` function, where you can specify what happens every time you open the menu.

```
static void OnActivate(void*)
{
    // add whatever needs to be done when the menu is opened here
}
```

The XML File

The XML file that the `MenuNameInit()` function reads from needs to be written in the following way. An example for a text button, edit field, radio button list, checkbox and label have been incorporated. The XML files are located in the `data\data\menu` folder

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE params SYSTEM "../tgf/params.dtd">
<params name="MenuName">

   <section name="dynamic controls">

        <!-- Example comment -->
        <section name="ExampleButton">
            <attstr name="type" val="text button"/>
            <attstr name="show box" val="no"/>
            <attstr name="text" val="Example text"/>
            <attstr name="tip" val="Example hint"/>
            <attstr name="h align" val="left"/>
            <attnum name="x" val="100"/>
            <attnum name="y" val="200"/>
            <attnum name="width" val="120"/>
            <attstr name="font" val="medium"/>
            <attstr name="color" val="0xFFFFFF"/>
            <attstr name="focused color" val="0xFFFFFF"/>
```

```
            <attstr name="pushed color" val="0xFFFFFF"/>
            <attnum name="image x" val="-10"/>
            <attnum name="image y" val="4"/>
            <attnum name="image width" val="60"/>
            <attnum name="image height" val="16"/>
            <attstr name="focused image" val="data/img/button-left-focused.png"/>
            <attstr name="enabled image" val="data/img/button-left.png"/>
            <attstr name="pushed image" val="data/img/button-left-pushed.png"/>
        </section>

        <section name="ExampleEditField">
            <attstr name="type" val="edit box"/>
            <attnum name="max len" val="32"/>
            <attnum name="x" val="100"/>
            <attnum name="y" val="100"/>
            <attnum name="width" val="150"/>
            <attstr name="h align" val="left"/>
            <attstr name="font" val="medium"/>
        </section>

        <section name="ExampleCheckbox">
            <attstr name="type" val="check box"/>
            <attstr name="text" val="Example text"/>
            <attstr name="tip" val="Example hint"/>
            <attnum name="x" val="200"/>
            <attnum name="y" val="100"/>
            <attnum name="image width" val="20"/>
            <attnum name="image height" val="20"/>
            <attstr name="color" val="0xFFFFFF"/>
            <attstr name="font" val="small_t"/>
            <attstr name="checked" val="no"/>
        </section>

        <section name="ExampleRadioButtonList">
            <attstr name="type" val="radio button list"/>
            <section name="options">
                <section name="0">
                    <attstr name="text" val="Example text radio button 1"/>
                    <attstr name="tip" val="Example hint radio button 1"/>
                </section>
                <section name="1">
                    <attstr name="text" val="Example text radio button 2"/>
                    <attstr name="tip" val="Example hint radio button 2"/>
                </section>
            </section>
            <attnum name="x" val="200"/>
            <attnum name="y" val="200"/>
            <attnum name="image width" val="20"/>
            <attnum name="image height" val="20"/>
            <attstr name="color" val="0xFFFFFF"/>
            <attstr name="font" val="small_t"/>
            <attnum name="selected" val="0"/>
            <attnum name="distance" val="10"/>
            <attnum name="minimum of one" val="1"/>
        </section>

    </section>
```

```
    <section name="static controls">

        <section name="1">
            <attstr name="type" val="background image"/>
            <attstr name="image" val="data/img/example.png"/>
        </section>

        <section name="2">
                    <attstr name="type" val="static image"/>
                    <attnum name="x" val="10"/>
                    <attnum name="y" val="400"/>
                    <attnum name="width" val="500"/>
                    <attnum name="height" val="50"/>
                    <attstr name="image" val="data/img/example.png"/>
            </section>

        <section name="3">
            <attstr name="type" val="label"/>
            <attnum name="x" val="100"/>
            <attnum name="y" val="300"/>
            <attstr name="text" val="Example text"/>
            <attstr name="h align" val="left"/>
            <attstr name="font" val="medium_t"/>
        </section>

    </section>

</params>
```

Using the XML

The XML file is split into two parts. The dynamic and static controls. The `dynamic controls` section could contain buttons and other controls as listed in the XML above. The `static controls` section could contain background images, static images, and labels, again listed in the XML above. Every dynamic control in the XML needs to be created in `MenuNameInit()`. Every `GfuiMenuCreateControl()` function returns a control id that can be used to access the object. Add a private `int m_exampleControl` variable for them.

```
// UI button controls
GfuiMenuCreateButtonControl(s_scrHandle, param, "ExampleButton", nullptr, OnButtonPush);
m_editFieldControl = GfuiMenuCreateEditControl(s_scrHandle, param, "ExampleEditField",
                     nullptr, nullptr, OnEditFieldSet);
m_checkboxControl = GfuiMenuCreateCheckboxControl(s_scrHandle, param, "ExampleCheckbox",
                    nullptr, OnCheckboxPush);
m_radioButtonListControl = GfuiMenuCreateRadioButtonListControl(s_scrHandle, param,
                           "ExampleRadioButtonList", nullptr, OnRadioButtonSelected);
```

All these buttons, regardless of their type (checkbox, radiobutton, etc.), need some `OnPush()` function. In GfuiMenuCreateButtonControl(), they have been named `OnButtonPush`, `OnCheckboxPush` and `OnRadioButtonSelected`. All of these have the same structure, but use either a different struct with info on the button type or no struct at all. For example `p_info->bChecked` for checkboxes and `p_info->Selected` for radiobutton lists.

```
static void ExampleOnPushFunction(tButtonTypeInfo* p_info)
{
   // add the functionality of the function here
}
```

The default values of the dynamic and static controls in the XML file are loaded and displayed on screen, but the internal variables (`m_checkboxChecked`, `m_radiobuttonSelected` and `m_editFieldContent`) have not yet been set to them. This is where the `LoadDefaultSettings()` function comes in.

```
static void LoadDefaultSettings()
{
    m_checkboxChecked =
        GfuiCheckboxIsChecked(s_scrHandle, m_checkboxControl);

    m_radiobuttonSelected =
        GfuiRadioButtonListGetSelected(s_scrHandle, m_radioButtonListControl);

    m_editFieldContent = GfuiEditboxGetString(s_scrHandle, m_editFieldControl);
}
```

Going back to `MenuNameInit()`. Keyboard controls can be made with `GfuiAddKey()`. GfuiMenuDefaultKeysAdd() adds the possibility to use the up and down arrow keys (as well as pageup and pagedown) to scroll through all the dynamic controls in the menu. This is done in the order they are initialized in the `MenuNameInit()` function. Pressing `Enter` activates the currently selected button.

```
// keyboard controls
GfuiMenuDefaultKeysAdd(s_scrHandle);
GfuiAddKey(s_scrHandle, GFUIK_RETURN, "ExampleActionName", nullptr,
           ExampleOnPushFunction, nullptr);
GfuiAddKey(s_scrHandle, GFUIK_SPACE, "ExampleActionSwitchMenu", s_prevHandle,
           GfuiScreenActivate, nullptr);
```

Saving to Local Directory

The XML contains the default values and `MenuNameInit()` sets the internal values to these defaults. To be able to have the program remember the changed settings, you need to add functions that save and load the settings to a local XML file. In the code snippet below is an example how this can be done.

```
static void SaveSettingsToDisk()
{
    // Copies xml to documents folder and then opens file parameter
    std::string dstStr("config/MenuName");
    char dst[512];
    sprintf(dst, "%s%s", GfLocalDir(), dstStr.c_str());
    void* readParam = GfParmReadFile(dst, GFPARM_RMODE_REREAD | GFPARM_RMODE_CREAT);

    GfParmSetStr(readParam, "ExampleCheckbox", GFMNU_ATTR_CHECKED,
                 GfuiMenuBoolToStr(m_checkboxChecked));

    char val[bufferSize];
    sprintf(val, "%d", m_radiobuttonSelected);
    GfParmSetStr(readParam, "ExampleRadioButtonList", GFMNU_ATTR_SELECTED, val);

    GfParmSetStr(readParam, "ExampleEditField", GFMNU_ATTR_TEXT, m_editFieldContent);

    // Write all the above queued changes to the xml file
    GfParmWriteFile(nullptr, readParam, "MenuName");
    GfParmReleaseHandle(readParam);
}

static void LoadSettingsFromDisk(void* p_param)
{
    // Retrieve all setting variables from the xml file and assigning them to the
    // internal variables
    m_checkboxChecked = GfuiMenuControlGetBoolean(p_param, "ExampleCheckbox,
                        GFMNU_ATTR_CHECKED, false);
```

```
    m_radiobuttonSelected = std::stoi(GfParmGetStr(p_param, "ExampleRadioButtonList",
                            GFMNU_ATTR_SELECTED, nullptr));

    m_editFieldContent = GfParmGetStr(p_param, "ExampleEditField", GFMNU_ATTR_TEXT,
                         nullptr));

    // Match the menu buttons with the initialized values
    SynchronizeControls();
}
```

The `LoadSettingsFromDisk()` function at the end calls the `SynchronizeControls()` function. This function synchronises what is displayed on screen with the internal variables, making sure that whatever is shown on screen corresponds to what is set in the internal parameters. The `OnActivate()` function should also call this function, so add it there.

```
static void SynchronizeControls()
{
    GfuiCheckboxSetChecked(s_scrHandle, m_checkboxControl, m_checkboxChecked);

    GfuiRadioButtonListSetSelected(s_scrHandle, m_radioButtonListControl,
                                   m_radiobuttonSelected);

    char buf[bufferSize];
    sprintf(buf, "%d", m_editFieldContent);
    GfuiEditboxSetString(s_scrHandle, m_editFieldControl, buf);
}
```

Going to this menu can be done by calling the following line of code from where you want to open the menu, e.g. in an `OnPush()` function of a button.

```
GfuiScreenActivate(MenuNameInit(nullptr));
```

Now, everything is set up for a new menu. You only have to customize the XML with whatever you want to add, then add functionality to these elements in the .cpp and of course make sure they get saved locally.

4.2 Configuring Indicators

The configuration for indicators can be found in `speed-dreams/source-2.2.3/data/data/indicators` relative to the git repository, or in `data/data/indicators` relative to the installation location. This folder contains a `Config.xml` file and two folders: `sound` and `texture`. The `Config.xml` defines all of the properties for the indicators, while the actual sounds and textures are in the corresponding folders.

Each possible decision (e.g. steer left) has its own section. This section defines what texture, sound, and text should be shown when that decision occurs. Below is an example of the steer left section:

```
<section name ="steer left">
    <section name="textures">
        <attnum name ="xpos" val="0.75"/>
        <attnum name ="ypos" val="0.04"/>
        <attnum name ="width" val="80"/>
        <attnum name ="height" val="80"/>
        <attstr name ="no-help"           val="Steering_Off.png"/>
        <attstr name ="signals-only"      val="Steering_Warning.png"/>
        <attstr name ="shared-control"    val="Steering_SharedControl.png"/>
        <attstr name ="complete-takeover" val="Steering_CompleteTakeover.png"/>
        <attstr name ="autonomous-ai"     val="Steering_AI.png"/>
    </section>
    <section name="text">
        <attstr name ="content"   val="Steering"/>
```

```
        <attnum name ="xpos"      val="0.745"/>
        <attnum name ="ypos"      val="0"/>
        <attstr name ="font"      val="jura-normal.glf"/>
        <attnum name ="font-size" val="300"/>
    </section>
    <section name="sound">
        <attstr name ="source" val="warning_left_driftingright_tts.wav"/>
        <attstr name="looping" val="yes"/>
        <attnum name="loop-interval" val="3"/>
    </section>
</section>
```

Note that all of the sections are optional. Removing the outermost section (in the example `steer left`) will prevent that decision from having any indicators. Removing an inner section (e.g. `texture`) will prevent that specific feature to not show up.

4.2.1 Icons

To define the icon texture add a section with the name `textures`. Every different intervention type can have unique icons defined. This section has the following attributes:

- `xpos` is the x coordinate where the icon will be shown. The range is [0, 1] with 0 being the left side, and 1 being the right side of the screen.
- `ypos` is the y coordinate where the icon will be shown. The range is [0, 1] with 0 being the bottom, and 1 being the top of the screen.
- `width` is the icon width in pixels, it should be bigger than 0.
- `height` is the icon height in pixels, it should be bigger than 0.
- `no-help` is the path to the image to show when `No Signals` is selected, relative to the `texture` folder. This attribute is optional, when not defined it will not show an image when `No Signals` is selected.
- `signals-only` is the path to the image to show when `Only Signals` is selected, relative to `texture` folder. This attribute is optional, when not defined it will not show an image when `Only Signals` is selected.
- `shared-control` is the path to the image to show when `Shared Controls` is selected, relative to `texture` folder. This attribute is optional, when not defined it will not show an image when `Shared Controls` is selected.
- `complete-takeover` is the path to the image to show when `Complete Takeover` is selected, relative to `texture` folder. This attribute is optional, when not defined it will not show an image when `Complete Takeover` is selected.
- `autonomous-ai` is the path to the image to show when `Autonomous AI` is selected, relative to `texture` folder. This attribute is optional, when not defined it will not show an image when `Autonomous AI` is selected.

An example section:

```
<section name="textures">
    <attnum name ="xpos" val="0.75"/>
    <attnum name ="ypos" val="0.04"/>
    <attnum name ="width" val="80"/>
    <attnum name ="height" val="80"/>
    <attstr name ="no-help"           val="Steering_Off.png"/>
    <attstr name ="signals-only"      val="Steering_Warning.png"/>
    <attstr name ="shared-control"    val="Steering_SharedControl.png"/>
    <attstr name ="complete-takeover" val="Steering_CompleteTakeover.png"/>
    <attstr name ="autonomous-ai"     val="Steering_AI.png"/>
</section>
```

4.2.2 Audio

Adding sounds is done by adding a section with the name `sound` to the decision section. This section has the following options:

- `source` is the path to the sound file relative to the `sound` folder. The file has to be a mono .wav file.
- `looping` is a boolean value that determines whether the sound should loop, or only play once when a decision starts being executed. This value can be `yes` or `no`. This attribute is optional, when not defined it defaults to `no`.
- `loop-interval` is the minimum time in seconds between restarts of the sound. A sound will never be restarted when it's playing. This attribute is optional, when not defined it defaults to `0`.

An example section:

```
<section name="sound">
    <attstr name ="source" val="warning_left_driftingright_tts.wav"/>
    <attstr name ="looping" val="yes"/>
    <attnum name ="loop-interval" val="3"/>
</section>
```

4.2.3 Text

Adding text is done by adding a section with the name `text` to the decision section. This section has the following options:

- `content` is the text that will be shown when the decision is made.
- `xpos` is the x coordinate where the text will be shown. The range is [0, 1] with 0 being the left side, and 1 being the right side of the screen.
- `ypos` is the y coordinate where the text will be shown. The range is [0, 1] with 0 being the bottom, and 1 being the top of the screen.
- `font` is the font to use, this has to be a glf file in the `data/data/fonts` folder.
- `font-size` is the size of the text. This attribute is optional, when not defined it defaults to `10`.

An example section:

```
<section name="text">
    <attstr name ="content"   val="Steering"/>
    <attnum name ="xpos"      val="0.745"/>
    <attnum name ="ypos"      val="0"/>
    <attstr name ="font"      val="jura-normal.glf"/>
    <attnum name ="font-size" val="300"/>
</section>
```

4.3 Creating Environments

This section describes how a new environment is created and how to add the environment to DAISI. To do this there are a couple of programs that need to be installed before a new environment can be added to DAISI.

4.3.1 Required Programs

Every environment in DAISI is first created in the track editor from Torcs. So the first step would be installing Torcs. This can be done via `https://sourceforge.net/projects/torcs/`. When the program has been installed it can be put aside for now. The next program that is needed is Blender. It will be used to add the 3D map we get from the track editor. Blender can be downloaded from Blender.org. Blender can serve two purposes: it can be used to edit the track / adding details (this

could also be done in any other 3D software program), but its most important function will be importing and exporting AC3D files. Since DAISI only works with .ac or .acc files, a converter is needed to open and create these file formats. Blender is the only free 3D software program that can import and export AC3D fill with the help of a plugin. The plugin was created by GitHub user NikolaiVChr, so the plugin can be downloaded from his GitHub page: `https://github.com/NikolaiVChr/Blender-AC3D`. On this page it is also explained how to install the plugin, and what to do if you can not see it in the Blender import/export menu.

4.3.2 Working with the track editor

Workspace

The track editor is a very simple program (see figure 4.6). When creating a new environment this will not be the place where you spent most of your time creating an environment. In the top left there are five different text buttons, namely *file*, *edit*, *view*, *segment*, and *help*. Most of these are not used, because the alternative is much easier to use. The *file* text button is the only button of importance, (see figure 4.7). Here a new file can be created by selecting *new*, an old file can be opened via *open*, and *save* to save the current file. Below that is *import* and *export*. The import function is not used in the process of making a new environment. However, the export function is used, here an .ac file can be exported for later use. Just beneath the *import* and *export* buttons is the *properties* tab.

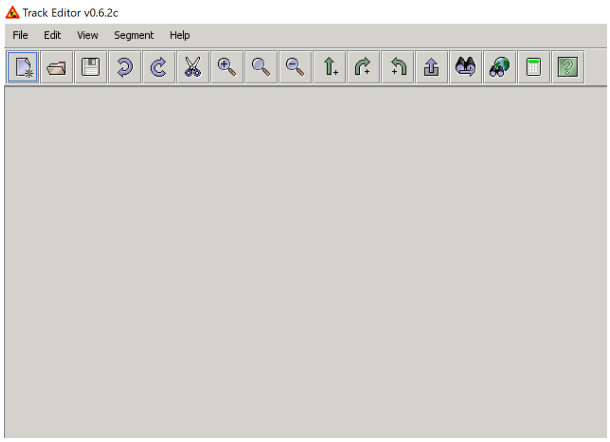


Figure 4.6: Workspace track editor

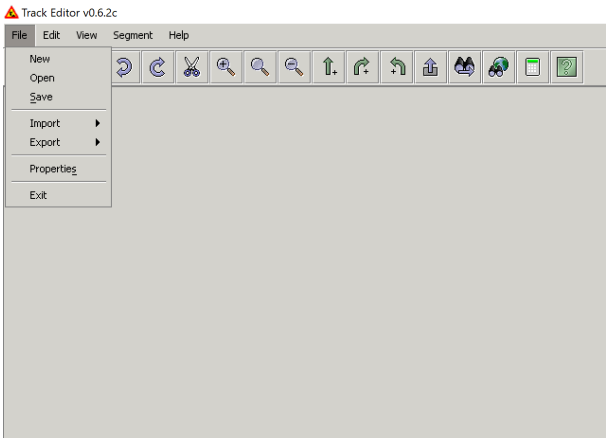


Figure 4.7: File menu options

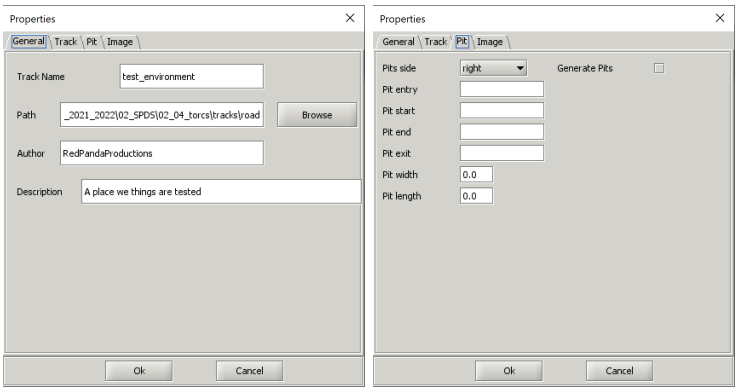
Below the *properties* button in the file menu is *exit*; pressing this will close the track editor and ask if you wish to save your changes. Beneath the text buttons are the icon buttons; these are much easier to use. The icon buttons are explained from left to right, as shown in figure 4.6:

- The first one is a *sheet of paper*. This icon will open the *New Project* window, where a new project can be created.
- Next to that is a *folder*, here you can open an existing track.
- Then the *save* icon, pressing this will save the current document.
- *Left blue arrow*, pressing this will undo the last change you have made.
- *Right blue arrow*, pressing this icon will redo your last action.
- With the *scissor* icon, you can highlight a part of the track and when you press it will cut out that part of the track.
- Pressing the first one will zoom in.
- Pressing the second will center the camera again.
- Then the third, pressing this will zoom out.
- Then there are three different green arrows. Pressing the straight one, will add a straight segment.

- Pressing the right arrow for adding right turns.
- Pressing the left arrow for adding left turns. The next two icons are never used.
- Then the *binocular icon with the globe behind it*, pressing this can toggle a background image on or off.
- Then there is a *calculator*. This function is never used.
- Lastly, there is a *question mark* that will tell you information about the program.

Beneath the toolbar there is the work area. This is where the track segments can be added.

Selecting properties in the file menu the properties window, see figure 4.8. Here are a couple of different things that can be changed. In the *general* tab, the track name, file path, author, and description can be changed. In the *track* tab, the width of the road can be changed. In the *pit* tab a pitstop can be added to the track, though this is not needed for any normal road/highway. In the last tab image, a background image can be added and the scale of the image can be changed. The properties window can be closed by pressing either the *ok* or *cancel* buttons.



(a) General tab (c) Pit tab

(b) Track tab (d) Image tab

Figure 4.8: Properties window

Creating a New File

Before starting with creating an environment, it is recommended to have a picture of the track layout you want to create. See figure 4.9 for an example. The first step to making any environment is working with the track editor from Torcs. To open the track editor, the Windows command prompt is needed. Open the Windows command prompt and set the path directory to the Torcs folder. You can do this by typing `cd` followed by the path directory. (For example, `cd downloads/torcs`.) Once that is done, press the `enter` key so that a new command can be typed. Now type in: *trackeditor*, and hit the `enter` key again. This will open a new program called track editor. If a new window does not appear, it is possible that Java is not installed on your machine yet. But if the window does appear, you can start

working on the new environment. First create a new file. This can be done by clicking on the *sheet of paper icon* (the most left icon in the toolbar) or by first selecting *file*, and then *new*. This action will open a new window called *New Project*. This window consists of four open text boxes that need to be filled in. Important to know is that all boxes must be filled in, if this is not done it will create problems later. One example of such a problem is that you might not be able to export the .ac file. In the top bar the name of the environment can be added. Just below that is the path, here you can select where you want the file to be saved. It is recommended to save the file in `torcs\tracks\road` as this will make future steps much easier. In the two boxes below that, the author and a description of the track can be added. Note that these can be easily modified later, but these boxes must contain something for now. When everything has been filled in, press the *ok* button in the bottom of the window. This will close the New Project window and open the track editor again. Here you will now see an oval in the work area (see figure 4.10), this is an automatically created road that can be removed later.

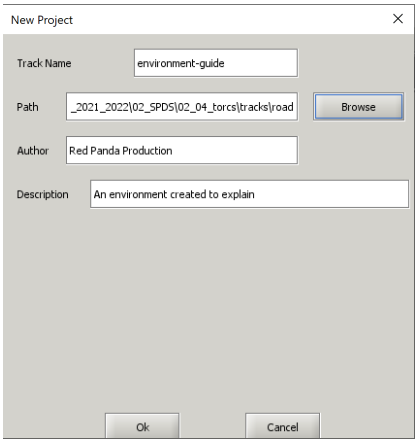


Figure 4.9: New project

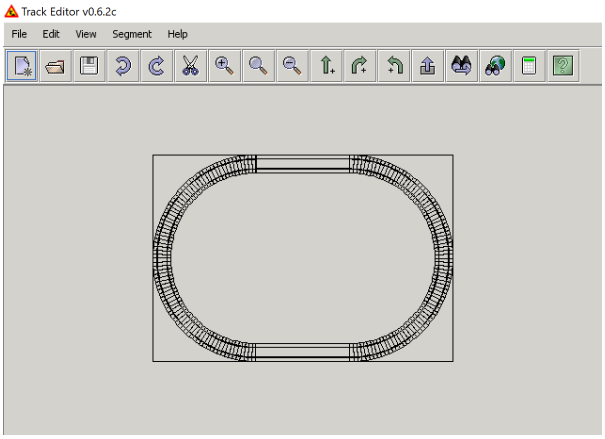


Figure 4.10: Starting oval

Adding a Background Image

In the track editor you can use a background image, (see figure 4.11). This can be done by first going to *file*, then *properties*. Here are four different tabs, the last one is called *image*. Here an image can be selected by selecting *browse* and then selecting the background image you wish to use. Once an image path has been selected the *properties* window can be closed by pressing the *ok* button. If the image is not visible it can be turned on by going to the *binocular icon with the globe behind it* in the toolbar, and pressing it once. This will toggle the background image on or off. The image may be too small or too large, the scale of the image can be changed in the properties window in the *image* tab. Note that the image must be refreshed in the work area to see the change. This can be done by turning the image on and off. The position of the image in the work area can be changed by `right-clicking` and dragging it to the desired location.

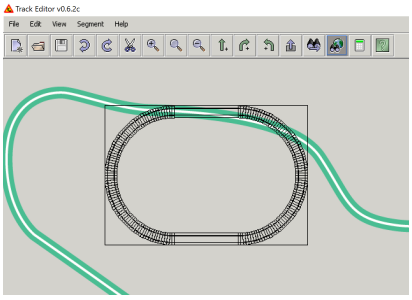


Figure 4.11: Background in track editor

Making the Road

Now that all the preparations are done, the starting oval can be modified. The easiest way to start is by removing everything but one straight segment. Every segment has its own properties; they can be accessed by selecting the segment. This will open a new window called *segment editor*. This window has three tabs, named *left*, (see figure 4.12), *center*, (see figure 4.14), and *right*, (see figure 4.13). These refer to the segment that is selected. The *center* tab is the only one that changes depending on which type of segment is selected. When a straight segment is selected the sliders for *arc*, *radius start*, and *radius end* are disabled, and only the *length*, *height start*, and *height end* are editable, (see figure 4.14). When adding a right or left turn these sliders are editable, see figure 4.15 In the *left* and *right* tabs different

things are controlled. *Barrier height* and *width*, *side height* and *width*, and *border height* and *width*. The surface type can also be changed here. The surface type refers to the material the car will be driving on. For the purpose of DAISI, the best surface type to use is any one of them with **asphalt** in the name. The properties that segments have on the left and right are best changed in the .xml file, because this is much faster than changing the properties one by one.

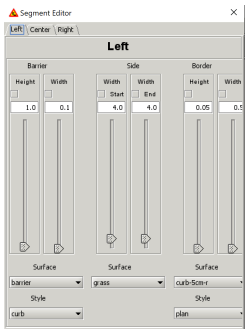


Figure 4.12: Segment editor left

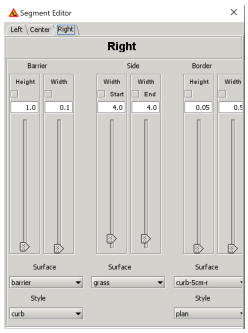


Figure 4.13: Segment editor right

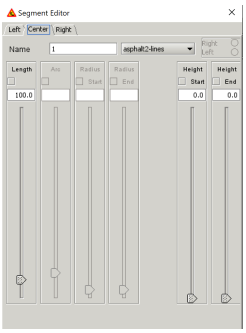


Figure 4.14: Segment editor straight

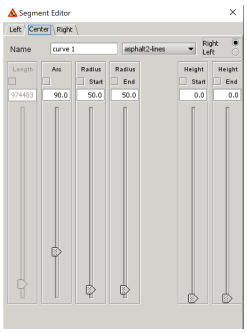


Figure 4.15: Center left turn segment

When building your track, segments will always be added in front of the last one. Thus, it is not possible to build out a track from the back. However, it is possible to add segments between other segments. Adding a segment is done by selecting one of the *green arrows* (see figure 4.16) in the toolbar and selecting where you would like to place it in the work area. As the track must always be a circle, the last segment that is placed must be attached to the first segment. If you want the car to drive in the right most lane, it is important that you take into account that the AI will always try to drive in the middle of the road. So to make it visually apparent that the car is driving in the right lane texture can be used to manipulate the appearance of the road. (For example: when you want a two lane road going one direction, make it a three lane road but change the third lane into a shoulder/breakdown lane.) Doing so will create a road where the car will always drive in the right-most lane. If you want the car to be on the right side of the road, it is important that you take into account that the car will try to stay in the middle of the track. When the track is completed, it can be exported. Be sure to save the track before exporting. Exporting the track can be done by going to, *file* → *export* → *AC3*. When the export is done it will display 'track finished'. Figure 4.17 displays what a successful export might look like.

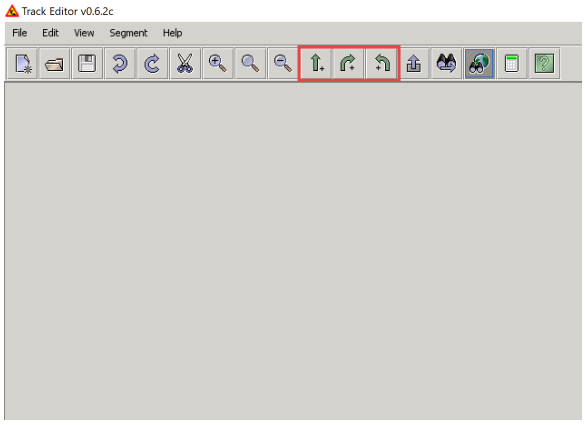


Figure 4.16: Segment arrows

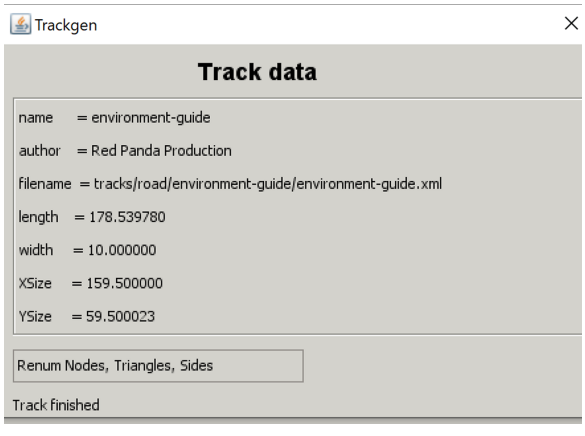


Figure 4.17: Export window

Changing the .XML

Now that the track has been exported and the .xml has been updated by the latest save, changes can be made to the .xml file. The .xml file and the .ac file can be found in the Torcs track folder where the track was originally created in the track editor. The file path for your track will look something like this: `torcs\tracks\road\...`. Here should be a folder with the name that was given at the start of working

in the track editor. When opening a tracks .xml file, a lot of information is shown at once, like the track segment, graphics, and category. In the first section (see figure 4.18), the name, category, version, author, and description are specified. In the next section (see figure 4.18), graphics are specified. Here the 3d description is the most important one, and this is where a background sky image can be added. In the next section (see figure 4.18) the properties of the main track are specified. Then the next section (see figure 4.19) is track segments. In this section, most of the changes are made. To make the road look more like a real one, the following changes are recommended:

- Changing the left and right side start and end width to `2.0` instead of `4.0`.
- Changing the left and right border width from `0.5` to `0.0`.
- Changing the left and right border width from `0.5` to `0.0`.
- Changing the left and right barrier height from `1.0` to `0.7`.

When all changes have been made, be sure to save the file. Note that in figure 4.19 the old values are used.

Figure 4.18: Track description

Figure 4.19: Segment description

Making a New .AC File

Now that you have made changes to the .xml you will need to create a new 3d description of the .xml. To do this open the Windows command prompt and set the path to the Torcs folder, so that you can open the *track editor* again. In the track editor go to *file*, *open*, then locate the folder with the .xml and the .ac file. In this folder open the .prj.xml file. This will load the track from before instead it will have the changes that were made in the .xml file. Once the file has been loaded, save the project. This may seem unnecessary since it looks like nothing has changed, but for a successful export this step is required. Once the file has been saved, a new export can be made. If you want to check the result for yourself, you will need to add a new folder in `...\source-2.2.3\data\tracks` called **road**. Put the environment folder from Torcs in the new road folder. Start up DAISI and select the environment in the environment selection screen. The changes to the .xml file should look something like this figure 4.20.

(a) Track before .xml changes

(b) Track after .xml changes

Figure 4.20: Result .xml changes

4.3.3 Editing the Environment

Loading The Environment Into 3D Software

At this point the environment folder with all the files can be removed from the track editor, and can be moved to any location on your machine. To be able to load the .ac file that was created by the track editor, the Blender plugin is used. Open Blender and create a new general project. Next, go to `file` → `import` and select `AC3D (.ac)`. If this option is not present, the plugin was not installed correctly. (A solution is provided on NikolaiVChr GitHub page `https://github.com/NikolaiVChr/Blender-AC3D/tree/2.80`.) From here, go to where the .ac file is located and import it into Blender. Note that importing any .ac file may take some time depending on the size of the file and your machine. When the .ac file has finished loading, two things can be done:

1. First is to start working in Blender and start by giving the track its textures.
2. Second: if Blender is not the preferred 3D software, it is possible to export the .ac file as a different file type. For example, a .fbx and import that into the preferred 3D software, and also start here with giving textures.

Note that you can move the road, but this will not mean the bounding box will also move.

Making Changes to the Environment

After giving the track its base textures, it is time to add details to the environment. Think about objects like: road signs and street lights. This step may take the most time since an environment could contain a lot of details. However, it is important to know that the more details are added, the less DAISI will be able to perform well. That is why it is recommended to keep the details to a minimum. When adding very large objects like a mountain, make sure that the polygon count is not too high. A recommended number would be round 600. Any objects that you add, the 3D software will not have a bounding box. So during a race you will be able to drive right through them. When all the changes have been made, it is time to export a new .ac file. For this you will also need to use Blender, and export the AC3D file here. Before doing this, make sure that all the polygons are triangulated. Then you can export the .ac file; you can save it anywhere on your machine. But be sure that you add **tkmn** before the name of your track, as this is required for the next step.

AC to ACC

To be able to see the new environment in DAISI the .ac file needs to turn into an .acc file. Luckily, this can easily be done with a script. To run the script, two files are needed: **sd2-accc.exe** (Which can be found in `...\source-2.2.3\out\build\x86-Debug`), and **stripe.exe** (which can be found in the \torcs folder). It is recommended to put both these files into a new folder on the desktop. Before running the script add the .ac file and all the textures in the same folder as sd2-accc and stripe. (See figure 4.21.) For the next step, the Windows command prompt is needed. Open the Windows command prompt

and set the path to the folder on the desktop. In the Windows command prompt, the following command should be run: `sd2-accc +o tkmntrackname.ac trackname.acc`. (Trackname should be changed with the name of your track.) When pressing `enter`, the script will start to run and create the .acc file. It is possible that the script suddenly stops. If this happens most of the time, it can be fixed by just running the script again. It could happen multiple times, but the best fix is to run the script again. When the script stops, it could give an error message. The error is most of the time caused by a mistake in the 3D model. If everything goes well and the script has successfully finished, this will be indicated by the word **end** in the Windows command prompt.

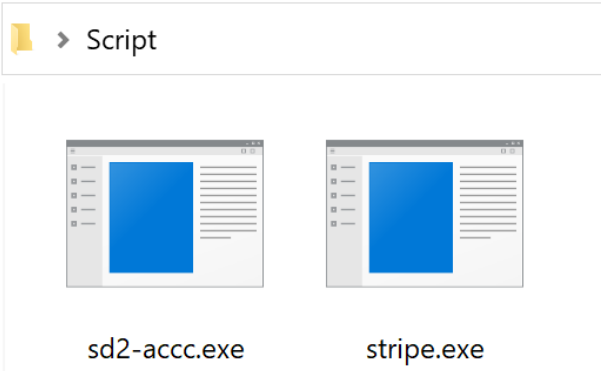


Figure 4.21: Script files

4.3.4 Environments in DAISI

Adding a Environment

Now that an .acc file has been created. The 3D description must be changed in the .xml file. This is done under the *graphic section* → *3d description* and changing the extension here from .ac to .acc. Note that the .acc file and all the texture must be in the same folder as the .xml file, and that the names are the same. Now you can add the folder with all the files to DAISI. This is done by going to: `...\source-2.2.3\data\tracks`, and here choosing a folder that best fits your environment and adding your environment folder there. These folders refer to the category of your track. For DAISI to be able to find the new environment, you must change the category value in the .xml file. It must be changed to the name of the category folder. See figure 4.22 for a visual representation.

```
<params name="test" type="param" mode="rw">
  <section name="Surfaces">&default-surfaces;</section>
  <section name="Header">
    <attstr name="name" val="environment-guide" />
    <attstr name="category" val="road" />
    <attnum name="version" val="4" />
    <attstr name="author" val="Red Panda Production" />
    <attstr name="description" val="An environment created to explain" />
  </section>
  <section name="Graphic">
    <attstr name="3d description" val="environment-guide.ac" />
    <section name="Terrain Generation">
      <attnum name="track step" unit="m" val="20" />
      <attnum name="border margin" unit="m" val="50" />
      <attnum name="border step" unit="m" val="30" />
      <attnum name="border height" unit="m" val="15" />
      <attstr name="orientation" val="clockwise" />
    </section>
  </section>
```

Figure 4.22: Environment changes xml

Adding Speed limits

DAISI can control the speed of the car, but to do so it needs information. This information needs to be added into the .xml file of every new environment, for every segment. It needs to be added in the first part of the segment description below the line that sets the type value. Here add the following line code: `<attnumname="speedLimit" val ="80" />` for every segment. In this example the value is 80, so the maximum speed on that segment will be 80 km/h. You can change the value to what you like.

Adding a Category

It is possible that your new environment does not fit within any of the current categories. So that is why it is possible to add a new one. This process is very simple. First make a new folder in the `data\tracks` folder, give this new folder the name of the new category. Then simply put the folder of the environment into this new category folder. There are now only two steps left. First, open the .xml file of the environment and change the category value. For example, from **road** to **highway**. Next, go to the `data\data \` tracks folder and here copy one of the existing .xml files. Then change the name to that of the new category. Lastly, open the .xml file and also change the naming here.

Changing the Skybox

Giving any environment a new skybox can be done in the .xml of that environment. So you will first need to open the .xml file and go to the section graphic. Here add the lines of code shown in figure 4.23 below the 3D description. Now an image can be added to the environment folder. Make sure that the image is 2048 by 512 pixels, and named **background-sky.png**.

```
<section name="background">
 <attstr name="background image"   val="background-sky.png"/>

<attnum name="background color R" val="0.28125"/>
<attnum name="background color G" val="0.4375"/>
<attnum name="background color B" val="0.75"/>

<attnum name="ambient color R" val="0.1"/>
<attnum name="ambient color G" val="0.1"/>
<attnum name="ambient color B" val="0.05"/>

<attnum name="diffuse color R" val="1"/>
<attnum name="diffuse color G" val="1"/>
<attnum name="diffuse color B" val="1"/>

<attnum name="specular color R" val="0.1"/>
<attnum name="specular color G" val="0.1"/>
<attnum name="specular color B" val="0.1"/>


<attnum name="light position x" val="-0"/>
<attnum name="light position y" val="1000"/>
<attnum name="light position z" val="3000"/>
</section>
```

Figure 4.23: Background-sky

Changing Names

Changing the name of the environment at a later point is possible. But when not all the names are the same, the environment will no longer be able to load. So you must ensure that all the files have the same name. The first thing to change is the environment folder. Next, change the .xml, and .acc or .ac file. Once these names are changed, be sure to also update the names in the .xml file.

4.4 Configuring Default Threshold Values

When the black box sends a decision, it sends a value. This value tells DAISI how much an action from the black box is needed to intervene with the driver's actions. As an example, we have a black box that tells the car to keep a speed above 80 km/h. The black box is set to only be in control of acceleration. When the car reaches 79 km/h it will send a signal of 0.05 and when the cars speed drops to 70 km/h, it will send a signal of 1 (the value send by the black box cannot be higher than 1).
The threshold values decide if the decision send by the black box gets executed. If the value send from the black box is equal or higher than the threshold, it will execute the decision, otherwise it will ignore the decision. So for the accelerate example from before: If you have set the threshold value of 0.1 for accelerate, then the black box is unable to accelerate the car if it is driving 79 km/h, but it is able to accelerate if the car is driving 70 km/h.
You might want to change the threshold values to test how they influence the black box effect on DAISI. However, to ensure that you have always the ability to set a default correct value (for each intervention type), there is a *Reset to defaults* button, both in the *Researcher Menu* and *Developer Menu* (see Figures 4.24 and 4.25).
You need to keep in mind that you probably do not want the same threshold values for different intervention types. You might want to be more lenient (or strict) with the threshold values when you choose *Only Signals* than when you choose *Complete Takeover*.

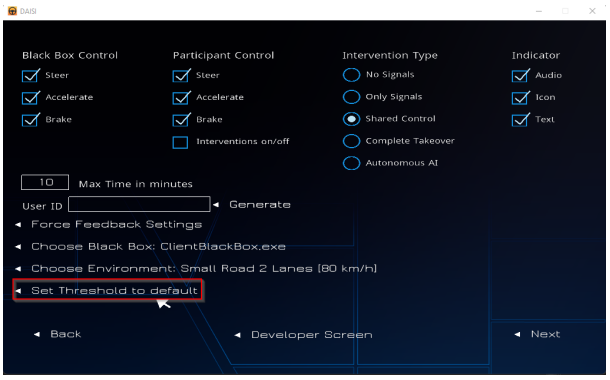


Figure 4.24: "Reset to defaults" button in the *Researcher Menu*

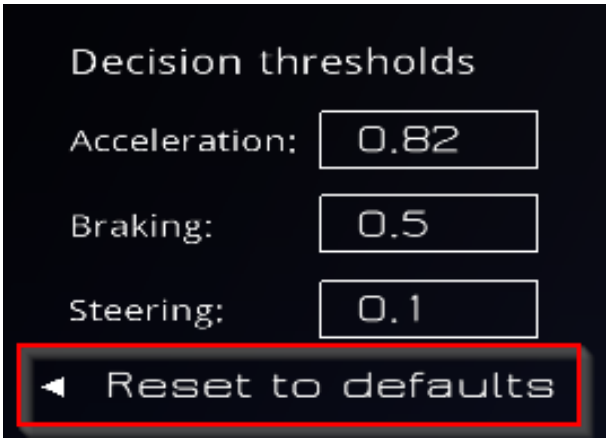


Figure 4.25: "Reset to defaults" button in the *Developer Menu*

4.4.1 Finding Default Threshold Values

Figuring out default threshold values is not a thing that can be done easily. Every black-box algorithm has their own way to calculate their values which they send to DAISI. This means that one black box might, for example, send a signal of 0.2 for a certain intervention, while another black box might send 0.05 for the same intervention for the exact same situation. There are only two ways to finding the default threshold values:

1. Trial-and-error: not the most ideal way, but probably the easiest way to decide the threshold values. Starting an experiment by pressing *Start* and going to the *Researcher Menu*. From there, going to the *Developers Menu* and setting some threshold values and pressing apply. After that you can press apply in the *Research Menu* and apply in the data selection screen to start the experiment. The experiment will start and you can see if the threshold values will give you the desired effect. If not, stop the race and start this process over by selecting *Start*.

2. Calculate the correct thresholds: If you are creating your own black box, you should know what kind of values the algorithm will send to DAISI. You might then be able to estimate the default values that will be required for the intervention types.

4.4.2 Setting Default Thresholds

Once you have found the desired threshold values—for either one or all intervention types—you can take the following steps to let the button *Reset to defaults* set the correct values:

1. Open windows explorer.
2. Open `DAISI/source-2.2.3/data/data/menu`
3. Open *DevelopMenu.xml*
4. Here you can change the default values for each of the intervention types (excluding no-signals) in the DefaultThresholdValues section, by changing the *val="..."* (see also figure 4.26).

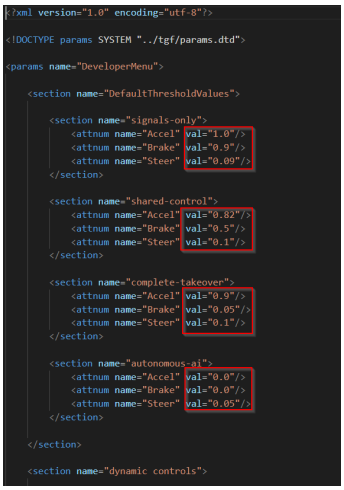


Figure 4.26: Values you need to change to set new default threshold values

Chapter 5

Integration Tests

5.1 Record Runs

The integration tests are pre-recorded runs of the simulation, where the user input, black box decisions, and all other required important information are stored. To record a run in the simulation, the recorder should be turned on in the *Developer Menu* as shown in figure 5.1. The *Developer Menu* can be accessed from the *Researcher Menu* as shown in figure 5.2.

After turning on the recorder, a run can be started as normal. The recorder will record all data necessary to replay the recording later. On windows the recordings can be found in `%Appdata%\sda\user_recordings`. On Linux they can be found in the `sda` folder in your chosen temp directory (default `/tmp/sda`). The recordings are automatically saved to a folder with the following format: `userRecordingYYYYMMDD-HHMMSS` at the end of a run. This folder consists of five files relevant to the recording:

- `car.xml` saves the properties for the car used during the recording.
- `decisions.bin` is a binary file containing all decisions made by the chosen black box.
- `recording.bin` is a binary file containing all user input necessary for replaying.
- `settings.xml` contains all research settings that were chosen for the recorded run, as well as the recorder version, and the decision thresholds.
- `simulation_data.bin` is a binary file containing data from the simulation which is verified during the replay.

Figure 5.1: The *Developer Menu* with the recorder enabled as shown in (1) and the black box in synchronous mode as shown in (2)

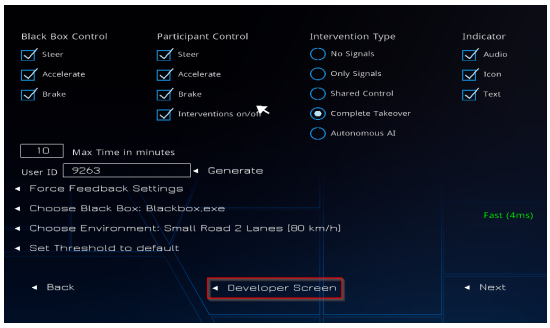


Figure 5.2: The *Researcher Menu* with the button to go to the *Developer Menu* highlighted in the red box.

To add a recording to the integration test suite, simply copy a recording folder (for example the userRecording folder) containing all recording files (`car.xml`, `decisions.bin`, `recording.bin`, `settings.xml`, and `simulation_data.bin`) to `speed-dreams/source-2.2.3/data/integrationTests/Recordings`. Finally, reconfigure CMake to regenerate the integration tests with the newly added recording.

5.2 Run Integration Tests

To run the integration tests the CMake option `OPTION_INTEGRATION_TESTS` has to be set to `ON`. After this running `ctest` will run the integration tests. To filter for just the integration tests the following command can be used: `ctest -R TestReplayRecordings/IntegrationTests.*`. More information can be found in the ctest documentation.

Chapter 6

Appendix A: UML Diagrams

6.1 Front-End Module UML Diagram

The large UML diagram of the front-end module is shown in Figure 6.1. This figure shows the complete architecture of the front-end module. The front-end module handles all the user input, user interfaces, and sends information to the back end

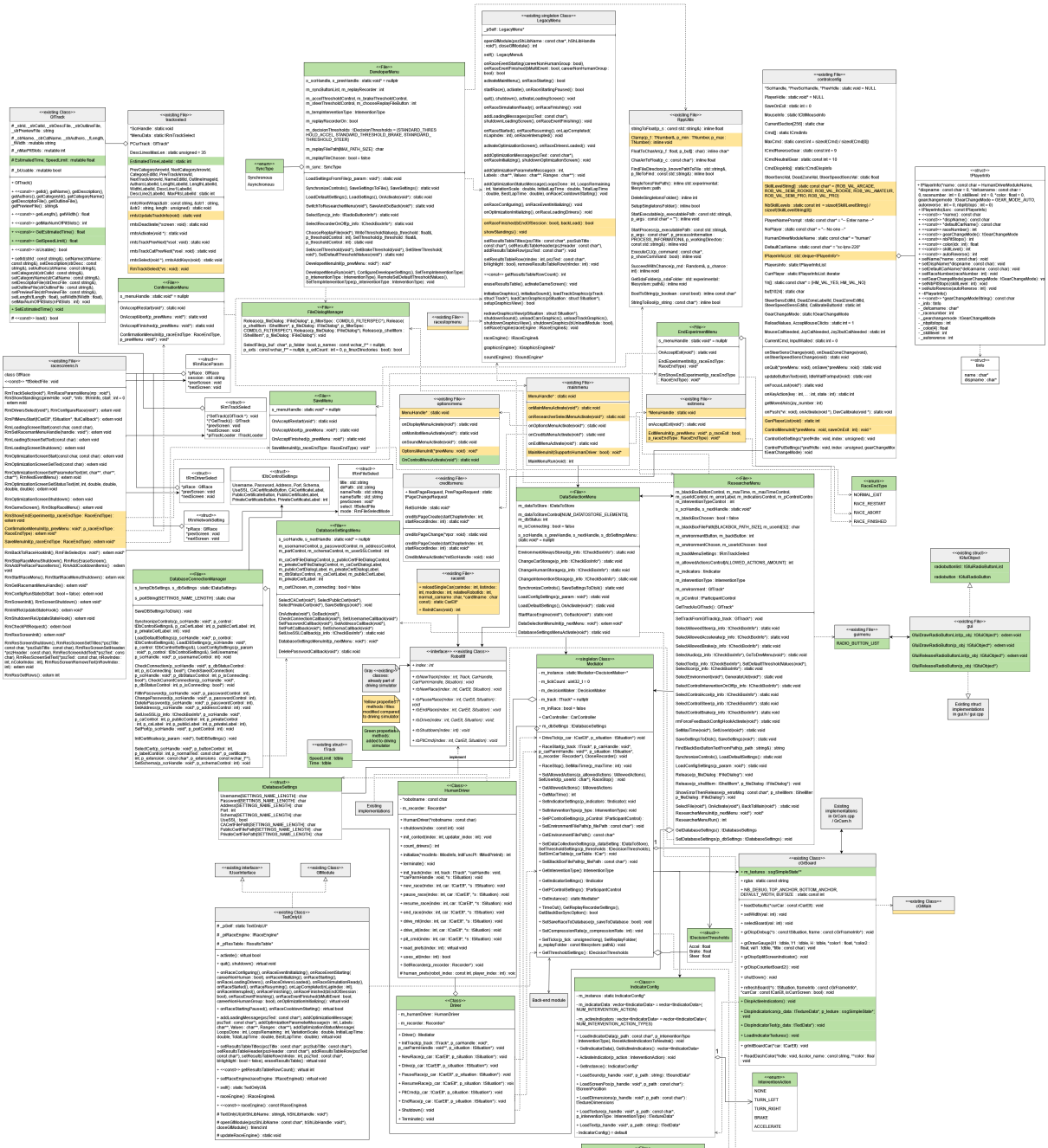


Figure 6.1: Large UML diagram of the front-end module.

6.2 Back-End Module UML Diagram

The large UML diagram of the back-end module is shown in Figure 6.2. This figure is shows the complete architecture of the back-end module. The back-end module handles the decision making process and connections to IPCLib and SDALib.

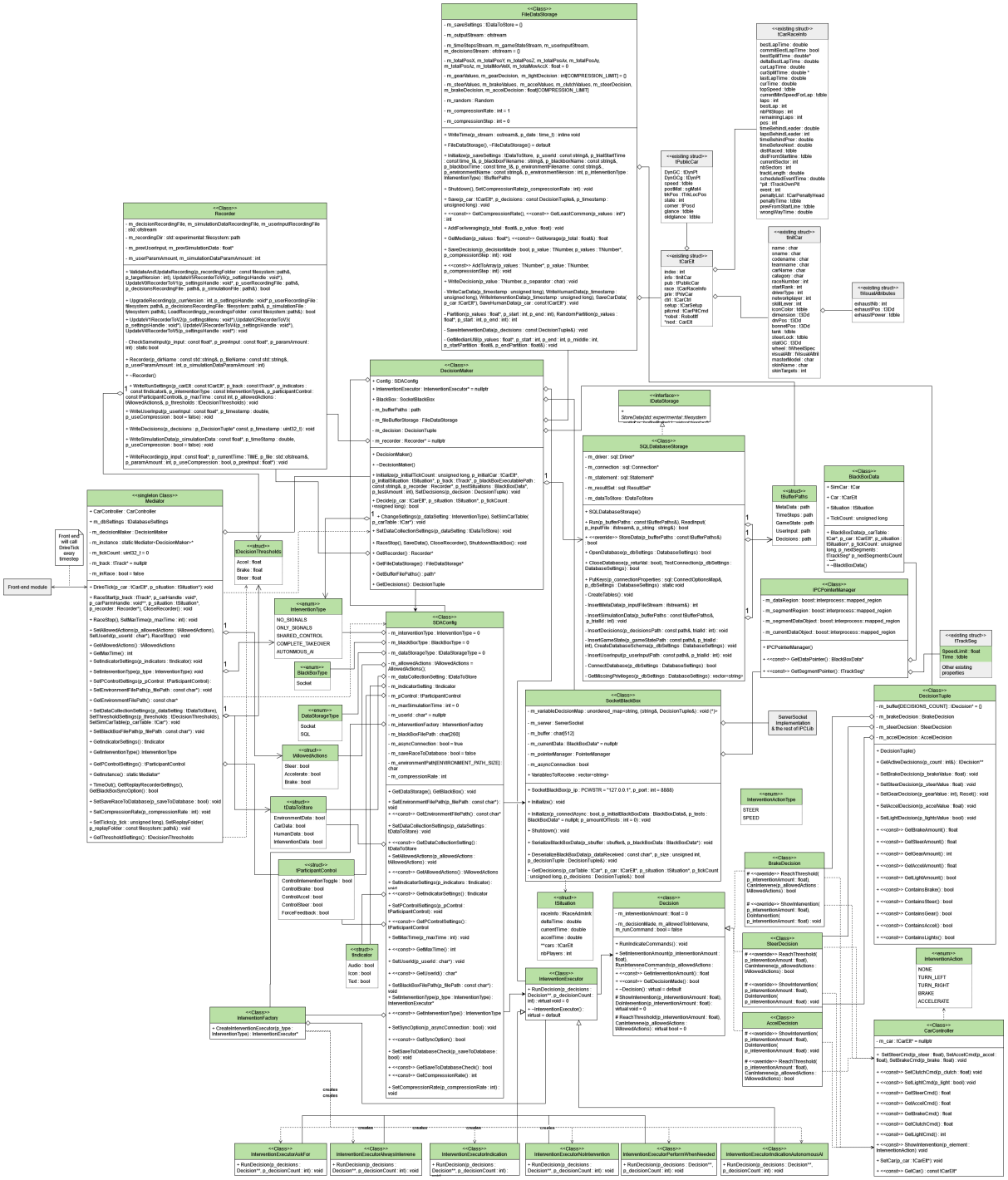


Figure 6.2: Large UML diagram of the back-end module.

6.3 SDALib

Within SDALib, we have implemented support for both a C++ black box algorithm and a Python black box algorithm. Figure 6.3 displays the full implementation of the SDALib library which is a combination of the support for C++ algorithms and Python algorithms as the team has implemented it.

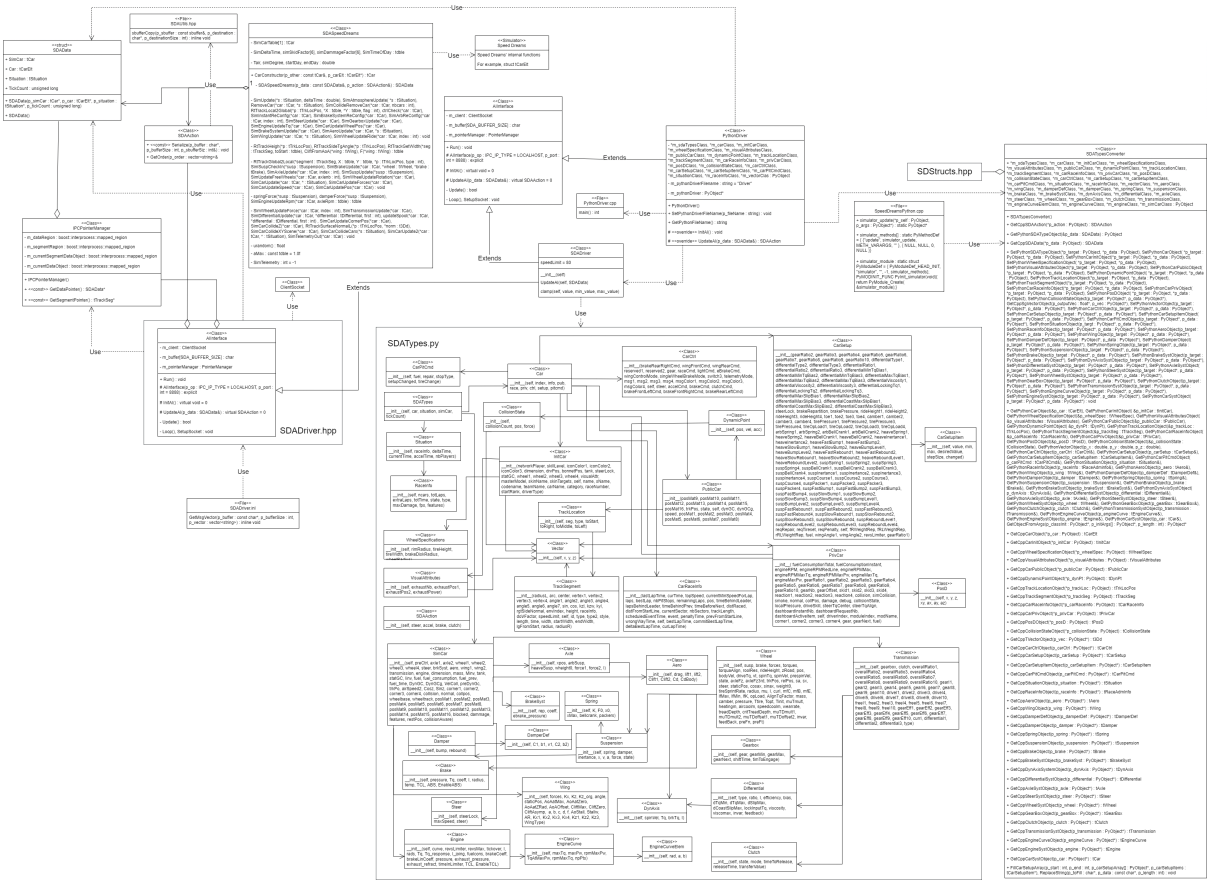


Figure 6.3: UML diagram of both the C++ black box and the Python black box implementation within the library SDALib.

6.4 IPCLib

For both libraries, IPCLib and SDALib, we had the goal to keep them as simple as possible. Therefore, to minimise the size of IPCLib, we minimise the size of the **ClientSocket** and **ServerSocket** classes, they inherit from the parent class **Socket**. This way, the **ReceivingThread** is also only implemented once with the help of the **Socket** class. See Figure 6.4 for the IPCLib UML diagram.

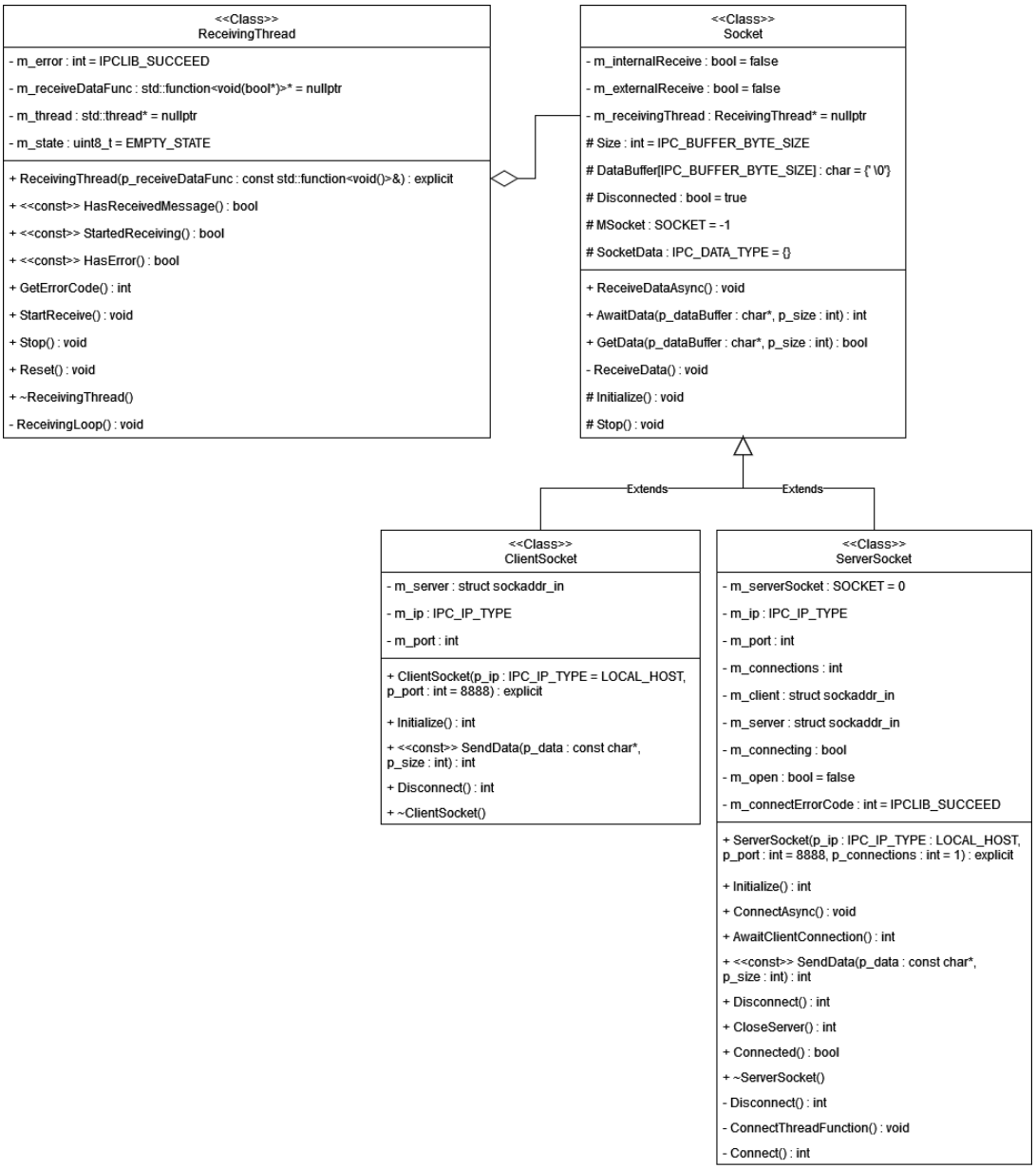


Figure 6.4: UML diagram of the library IPCLib.