QC4OpenX

Quality Checker for ASAM OpenX Standards

This documentation covers the generic part of the framework. You can probably find more CheckerBundles in the `bin` directory, which can be used.

Architecture

High-level Requirements

The following requirement list was the foundation for the design of the framework:

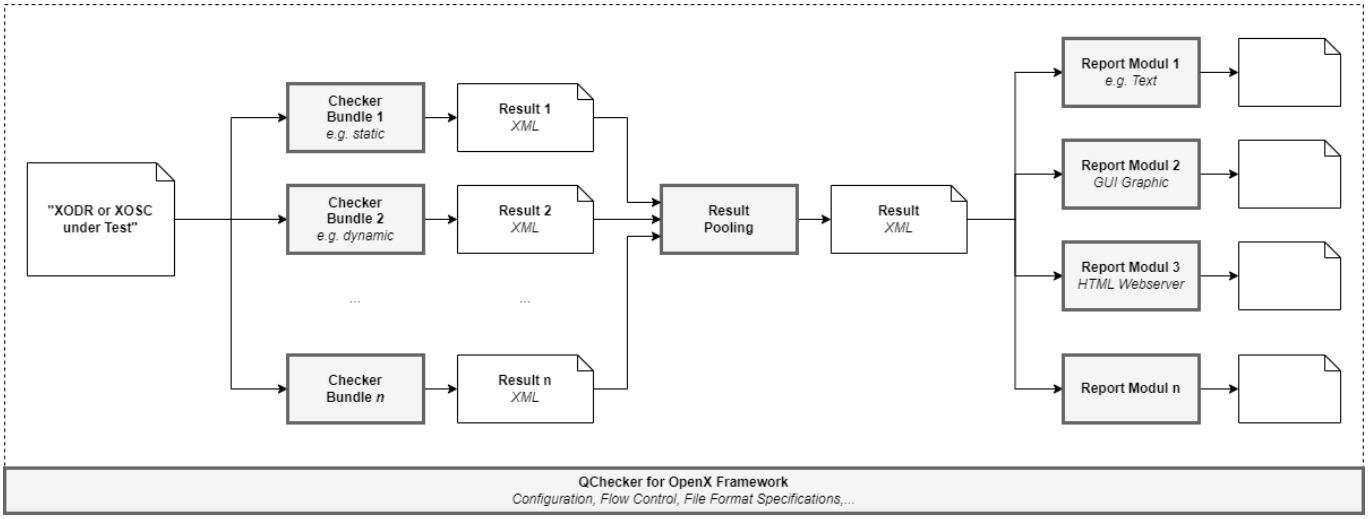
- Possibility to include a common ASAM rule set, which validates the rules from the specification document
- Write own rule sets, implemented in any programming language
- Extract meta information (e.g. if the file contain specific objects and the location of them)
- Configuration sets and parametrization of rules
- Possibility to include results based on the analysis of simulation runs
- Human and machine readable results
- Interactive usage or run in an automated workflow
- Locate the results in the XML source code and in a visualization of the map

Terms / Definitions

- **Route Model:** Static part of a possible concrete scenario ("Track under Test").
    - OpenDRIVE (XODR): Logical route description, exactly **one** file with extension .xodr
    - OpenSceneGraph Binary (OSGB): 3D model for usage in sensor frameworks (e. g. Visualization or raw data generation), exactly **one** file with extension .osgb or .ive (binary OSGB format)
    - Region-of-Interest (ROI): One or more routes in the route model that will be driven later in the simulation. These are "driven" in dynamic tests.
    - OpenSCENARIO (XOSC): Scenario definition, includes route definition (one file per route with extension .xosc)
- **Checker:** A software module or routine that checks exactly one rule or creates statistical information from the route model. This can be a static or a dynamic test. For a list of possible tests, see parent page.
- **Checker Bundle:** A program or framework that includes one or more checkers. CheckerBundles allow checks to be better structured and divided into logical groups. A CheckerBundle also contains the paths to OpenSCENARIO and OpenDRIVE. This makes it possible to implement a module that checks both formats. In short they:
    - are a summary of several checkers
        - with similar functions
        - from one supplier
        - developed in one distinct development environment/programming language
    - allow for files only be read/prepared once via parser.
    - can be started via external scripting/automation.
- **Report or Result File** CheckerBundles write their reports in an .xqar file. See Base Library and File Formats for details and examples.
- **Report Module:** Output / Transmission / Visualization Results

- **QC4OpenX** Framework, that:
    - feeds the route model to the individual Checker Bundles
    - selects and configures the checker
    - summarizes the results (Result Pooling)
    - calls the report modules so that the user can view or further process the results
    - provides standard checkers and standard report modules

Workflow QC4OpenX



Properties / Requirements

- QC4OpenX
    - Runs on Windows & Linux, Local & Server
    - Configuration via XML file
        - Selection of Checker Bundles
        - Selection of individual checkers in the bundles and their optional parameters → also defines the order in the overall report
        - Configuration which checkers output what level (Info/Issue) → leads to overall result "red" or "yellow"
        - Selection of report modules
- Checker
    - Properties
        - Optionally tunable (e.g. Limit values or chosen width for ROIs)
        - Id (short text)
        - Description (Detailed description)
- Checker Bundle
    - Runs if necessary only on certain platforms, because a third party product is not available for all platforms
    - Must be able to print out to the command line which checkers are included
- Result Pooling
    - Summarizes all results (overview and detailed view possible)
    - Gives each incident a unique Id → assignment results in different report modules
    - If necessary sorts and lists all information first and appends warnings and errors. Within these results, information is sorted by a central configuration. There is also the possibility for

summarizing and filtering.

- Report Module
    - Runs maybe only on certain platforms (e.g. Web server or local graphical application)

Parameterization and Sequence Control

A CheckerBundle can be parameterized. A distinction must be made between

- Globale Parameter
    - XodrFile --> OpenDRIVE file to be checked as database
    - XoscFile --> OpenSCENARIO file to be checked as database
- Parameter for the whole CheckerBundle --> "CheckerBundle Parameter"
    - Parameter that is locally relevant for this one Checker Bundle
    - If a graphical data base is checked, the .osgb/.ive file may also be used.
- Parameter for single Checkers --> "Checker Parameter"
    - Parameters that are locally relevant for one specific checker

Base Library

Overview

The Base library is a library for C++ that allows you to create your own modules in the QC4OpenX framework. With it, it's very easy to create your own CheckerBundles or ReportModules. The library consists of two parts, which are described in the following sections. The source code and an exemplary CheckerBundle can be found in folder `examples/src`.

The Configuration Format (`config_format`)

Configurations contain all necessary input parameters and parameterization for a CheckerBundle. Configurations can be created by the user and stored persistently as an XML file. A CheckerBundle should accept as single parameter an XML file containing the corresponding configuration (see User defined modules). The configuration XML file follows the schema file `doc/schema/config_format.xsd`.

To comfortably read out configurations, corresponding classes have been defined in the Base library. Reading out a configuration is simple, as shown in the following snippet.

```
cConfiguration configuration;
cConfiguration::ParseFromXML(&pMyConfiguration, strFilepath))
```

A configuration file can contain the parameters of multiple CheckerBundles and ReportModules. Therefore you should extract those parameters which correspond to your own bundle/module. For this we query the corresponding part from the configuration by the method GetCheckerBundleByName. Parameterization can thus be easily transferred from one parameter container (cParameterContainer) to another. Parameterization are nothing more than mappings from a keyword to a value and therefore allow easy overwriting and reuse.

```
cConfigurationCheckerBundle* checkerBundleConfig =
configuration.GetCheckerBundleByName("myChecker");
if (nullptr != checkerBundleConfig)
{
   inputParams.Overwrite(checkerBundleConfig->GetParams());
}
```

The Report Format (`report_format`)

The results that a Checker or CheckerBundle defines as output are stored in a report file. Reports contain information about the defects found, which are called issues. At least a description and an identifier is assigned to an issue. Additional meta information can be added to an issue: file location (cFileLocation), XML file location (cXMLLocation) or road information (cRoadLocation). This information is relevant for the ReportModule, for example, to point out meaningful errors in a GUI. The report XML file has to follow the schema file doc/schema/xqar_report_format.xsd.

```
myChecker->AddIssue(new cIssue("errorDescription", ERROR_LVL))
myChecker->AddIssue(new cIssue("take a closer look", INFO_LVL,
(cExtendedInformation*) new cFileLocation(3, 0, "This is row 3, column 0.")));
```

Issues are always assigned to exactly one checker. A log level is assigned to an issue: if it represents an error: ERROR_LVL, if it's just an information: INFO_LVL.

All CheckerBundles must contain at least one Checker. Checkers can be created or queried using a factory pattern:

```
auto pXSDCheckerBundle = new cCheckerBundle("myCheckerBundle");
cChecker* pOSCVersionChecker = pXSDCheckerBundle-
>CreateChecker("xoscVersionChecker", "Checks the validity of an xosc version.");
pOSCVersionChecker->AddIssue(new cIssue("Version not supported.", ERROR_LVL));
```

In a ResultContainer, all CheckerBundles are added. The ResultContainer is responsible for reading, writing and managing the data.

```
pResultContainer->AddCheckerBundle(pXSDCheckerBundle);
pResultContainer->WriteResults("resultFile"));
```

All objects in a ResultContainer are managed automatically. A release of the objects is not necessary and can be done with a `cResultContainer::Clear` or in the destructor.

Mapping `XMLLocation` to `FileLocation`

- **XmlLocation**: Addressing in an XML file based on an XPath expression
- **FileLocation**: A reference to a file with row and column

Some issues include a XML location to indicate where the issue occurred. During result pooling, a file location is determined from the XML location and saved as well.

The definition of the XML location is based on a XPath expression. Almost the complete XPath 2. 0 Standard is supported.

The XML processing is based on Qt 5. Therefore there are minor restrictions to the standard: Qt 5 Restrictions.

File Formats

Configuration File (*.xml)

The runtime configuration settings are stored in a configuration file. This file defines which CheckerBundles and what checkers are used, how they are parameterized and whether the issues are warnings or errors. If a CheckerBundle outputs errors that are not configured in this file, the result pooling removes them later on. This way only the relevant results are included in the overall result. Furthermore, it can be configured which ReportModules are started after all CheckerBundles finish execution.

```
<Config>
<Param name="XodrFile" value=e"myTrack.xodr">
<CheckerBundle application="SemanticChecker">
    <Param name="GlobalAngleTolerance" value="0.1"/>
    <Checker checkerId="elevationChecker" minLevel="1" maxLevel="3">
        <Param name="LocalAngleTolerance" value="0.2"/>
    </Checker>
</CheckerBundle>
<ReportModule application="TextReport">
</Config>
```

The runtime parses this file, then executes the configured CheckerBundles, and hands them the configuration file. The parameter "application" to the XML tags `<CheckerBundle/>` and `<ReportModule/>` specifies the name of the executable to be used.

Notes for the paths:

- "application" should contain the path of your executable relative to the path of the ConfigGUI binary - it is recommended to place all executables in the `./bin` directory.
- Results will be stored in the directory from which the call is made.

Result File (*.xqar)

The results of a checker are stored persistently as XML in the form of a result file (*.xqar). The Base library can be used to write a report. A report consists of problems (called issues) which contain syntactic or semantic flaws.

- An issue consists of the following parts:
    - Identifier: Unique number for an issue
    - Level: Indicates whether it is an information, a warning, or an error.
    - Description: Description in the form of a text
    - Optional description of where the problem lies, if applicable several grouped locations inside a **Locations** tag:
        - **FileLocation** *(typically generated automatically by the ResultPooling)*
            - A reference to a file with row and column

        - Example: `<FileLocation column="0" fileType="1" row="223124"/>`, with `fileType="1"` for XODR, `fileType="2"` for XOSC
    - **XmlLocation**
        - Addressing in a XML file with help of a XPath expression
        - Example: `<XMLLocation xpath="/OpenDRIVE/road[@id=&quot;1&quot;]"/>`
    - **RoadLocation**
        - Position in road coordinates, the angles are calculated based on the road orientation
        - Example: `<RoadLocation roadId="502066" s="0.5" t="0.0"/>`
    - **InertialLocation**
        - Position in inertial coordinates
        - Example: `<InertialLocation h="0.0" p="0.0" r="0.3" x="120.998703" y="0.0" z="-0.29552"/>`
- Optional external files (e. g. Images of generated graphs such as speed over distance). Currently not supported.

The following example shows the results obtained by running two CheckerBundles, one called SyntaxChecker and one SemanticChecker.

```
<CheckerResults version="1.0.0">
    <CheckerBundle name="SyntaxChecker" description="Syntax checker bundle"
summary="Found 1 incident" build_date="23.05.2019" version="1.0">
        <Param name="XodrFile" value="myTrack.xodr"/>
        <Param name="globalParameter" value="0.1"/>
        <Checker checkerId="xsdSchemaChecker" description="Checks the xsd
validity"  summary="Found 1 issue">
            <Issue description="Row:8693 Column:12 invalid schema" issueId="1"
level="1">
                <Locations description="empty content is not valid for content
model '(lane+,userData*,include*)'">
                    <FileLocation column="12" fileType="1" row="8693"/>
                </Locations>
            </Issue>
        </Checker>
    </CheckerBundle>
    <CheckerBundle name="SemanticChecker" description="Semantic checker bundle"
summary="Found some incidents" build_date="23.05.2019" version="1.0">
        <Param name="XodrFile" value="myTrack.xodr"/>
        <Checker checkerId="roadIdChecker" description="Checks validity of the
roadIds"  summary="Found 1 issue">
            <Issue description="Road with id=5 invalid" issueId="2" level="1">
                <Locations description="Road id is defined multiple times">
                    <XMLLocation xpath="/OpenDRIVE/road[@id='5']"/>
                    <FileLocation column="25" fileType="1" row="505"/>
                    <RoadLocation roadId="5" s="0.0" t="0.0"/>
                </Locations>
            </Issue>
        </Checker>
    </CheckerBundle>
</CheckerResults>
```

Regarding parametrization here an example of the section of the .xqar file where the default parameters are shown:

```
<CheckerResults version="1.0.0">
    <CheckerBundle description="Checks the..." name="StaticXodrChecker"
summary="173 issues...."> <!-- attribute file deleted! -->
        <Param name="GlobalAngleTolerance" value="0.2"/>
        <Param name="ElevationTolerance" value="0.1"/>
        (... more Params ...)
        <Checker checkerId="elevationChecker" description="Verify ..." summary="68
issues found...">
            <Issue description="#34: delta ...." issueId="0" level="2">
                <Locations description="Init section of scenario">
                    <XMLLocation ... />
                    (... more explicit location types ...)
                </Locations>
                (... more Locations ...)
            </Issue>
            (... more Issues...)
        </Checker>
        (... more Checkers...)
    </CheckerBundle>
    (... more CheckerBundles...)
</CheckerResults>
```

SchemaChecker

This bundles checks if the XML is valid against the schema file.

The following schemas are supported for OpenDRIVE:

- OpenDRIVE 1.1
- OpenDRIVE 1.2
- OpenDRIVE 1.3D
- OpenDRIVE 1.4H
- OpenDRIVE 1.5M
- OpenDRIVE 1.6.1
- OpenDRIVE 1.7.0

... and for OpenSCENARIO:

- OpenSCENARIO 0.9.1
- OpenSCENARIO 1.0.0
- OpenSCENARIO 1.1.0

The schema files are located under `bin/xsd`.

The bundle consists of two separate executables: `bin/XodrSchemaChecker(.exe)` and `bin/XoscSchemaChecker(.exe)`

User Defined Modules

If an own module is called without any command line arguments or options, then it should print a short help how to use it.

CheckerBundles

CheckerBundle Requirements

- You should provide an executable
- Your own module should adhere to the following rules.
- The executable needs execution rights and its location should be the bin folder (see File Formats for configuration details)

Note: The Base Library helps to read and write the file formats.

Run One Single CheckerBundle Based on a Configuration

As mentioned in the Base Library section your CheckerBundle should accept a xml file containing its configuration parameters. All **modules which follow this convention are a part of the framework** and **can be executed from the runtime**.

```
> checker_bundle_exe configuration.xml
```

Determine Checkers and Parameters

If the CheckerBundle executable is called with the command line argument `--defaultConfig`, it should write a report file (.xqar, see Base library for details) containing all checkers with their parameters and no issues. Each parameter should contain default values. The name of the file has to corresponds to the executable name (myCheckerBundle.exe should write myCheckerBundle.xqar).

```
> checker_bundle_exe --defaultConfig
```

Check a Single File With One Single CheckerBundle

A CheckerBundle can be called with one command line parameter and optional parameter settings. The first parameter defines the file which should be checked. The additional options are specific for the CheckerBundle and not defined here. As in the previous cases the CheckerBundle should write in the current directory an (.xqar) file with the same name as the executable.

This approach is used when testing only one CheckerBundle.

```
> checker_bundle_exe track.xodr [-o1 -o2 ...]
```

ReportModules

ReportModule Requirements

- You have to provide an executable
- Your own module shall visualize the results based on the result format
- Your module shall take a module configuration, based on the configuration format.
- Your module shall provide a default configuration by the parameter {{--defaultConfig}}
- The executable needs execution rights and its location should be the bin folder (see File Formats for configuration details)

Determine Configurations for ReportModules

If a ReportModule executable is called with the command line argument `--defaultConfig`, it should write a configuration file (.xml) containing all the parameters. Each parameter should contain default values. The name of the file has to corresponds to the executable name (myReportModule.exe should write myReportModule.xml).

```
> report_module_exe --defaultConfig
```

Using the Checker Framework

Configuration

Introduction

The following documentation refers to the Windows installation. The QC4OpenX core is available for other platforms as well. See the Linux paragraph for usage instructions under Linux.

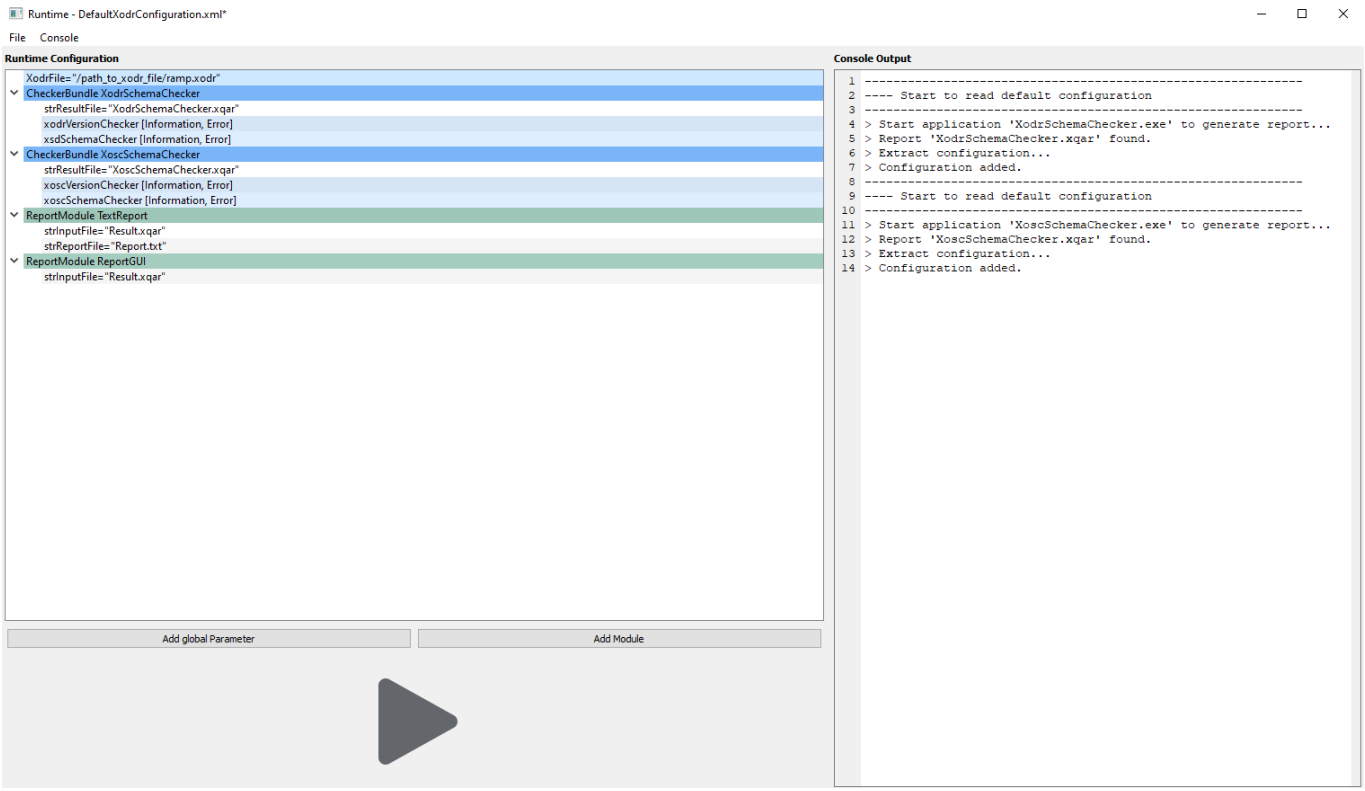
For starting the configuration GUI and the framework just start

- `ConfigGUI.exe`

in the `bin` directory of your installation. Per default, `DefaultXodrConfiguration.xml` is loaded by the application. You are free to replace this configuration by a different one. Depending on your use case, you will find the following pre defined configurations:

- `DefaultXodrConfiguration.xml` which can be used to check OpenDRIVE
- `DefaultXoscConfiguration.xml` which can be used to check OpenSCENARIO

You can open a configuration, by simply dropping the XML on the executable.



From there the CheckerBundle and ReportModuls can be selected and their parameter can be set. Special parameters are the global parameters

- `XodrFile`
- `XoscFile`

on the top of the configuration GUI. These parameters define the OpenDRIVE file and/or the OpenSCENARIO file which should be checked. All global parameters are passed to all CheckerBundles. When defining a XoscFile the XodrFile will be automatically retrieved, if an OpenDRIVE file reference is defined in the scenario. Both parameters are stored as absolute file paths currently.

Loading/Executing Configurations

The loaded configuration is shown on the left side of the GUI. The right side shows the output of each executable in sequence.

To open an existing configuration, use the menu **File→Open** or start the ConfigGUI.exe with the configuration as a command line argument.

The configuration can be started with the big arrow button. The first step in the execution is a **CleanUp**, where the old temporary files are deleted. After that, all configured CheckerBundles are executed as configured. The shown order in the configuration is also the order of it's execution. As a final step before calling the report modules, the result pooling is called automatically to store the checker results in one result file.

Changing/Saving Configurations

The configuration can be modified in the GUI with the context menu. Just press the right mouse button to add, remove or change the order of the modules. It's also possible to modify the parameters of the CheckerBundles. Within the menu **File → Save** or **File → SaveAs** can be used to write the configuration to a file. This is necessary before executing the CheckerBundles for the first time.

If you want to create a configuration from scratch use the menu **File →New**.

Own CheckerBundles and the ReportModules have to follow the specification in CheckerBundle/ReportModule Meta Model. If they use this mechanisms then the framework is able to include their output in the toolchain.

Export Log Files

To export the log from the right side of the GUI, you can use the menu entry **Console->Save** to store the output in one file.

Available Checker Bundles

- SchemaChecker

Reporting

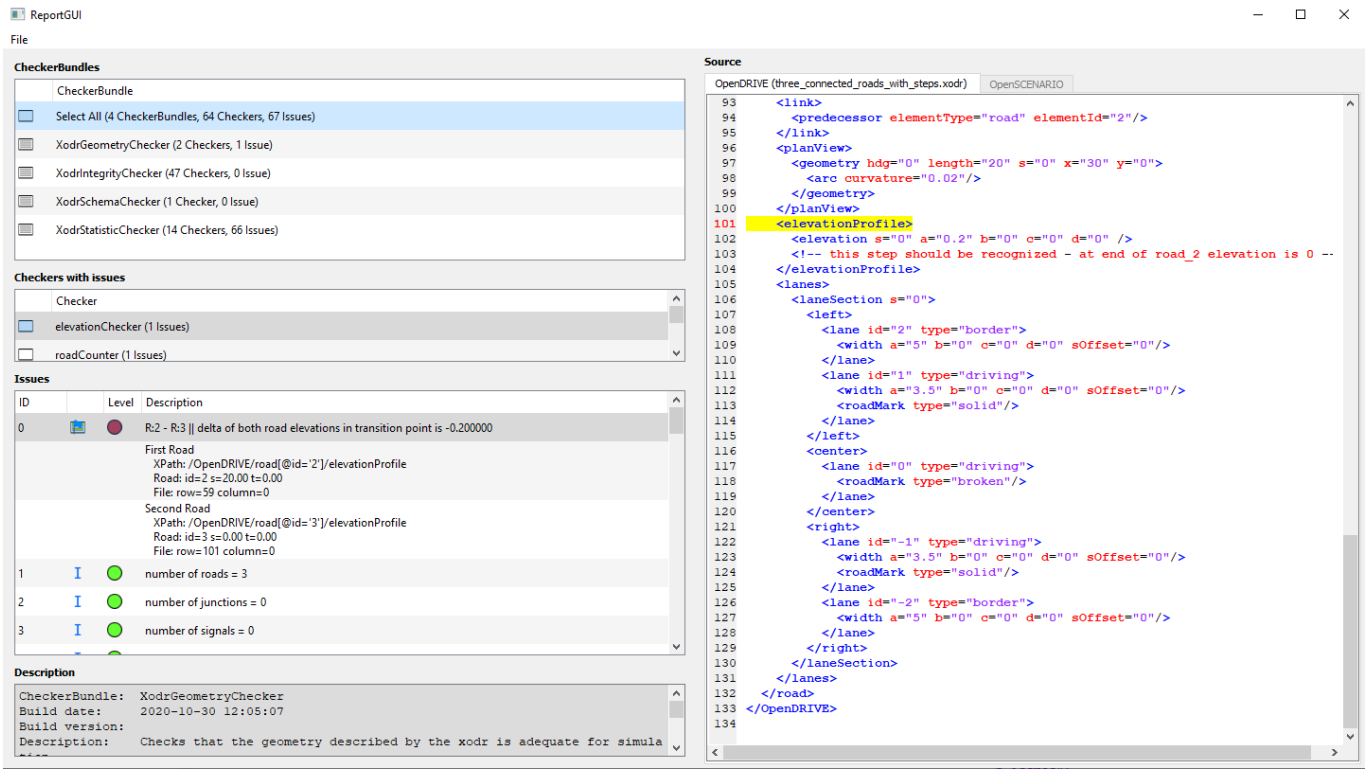
The reporting of issues can be done in a text file, in a GUI and/or graphically in the 3D model of the road network. The foundation for this is a file in the XML base Checker Result Format (XQAR) containing all issues. The file format specification can be found on the page File formats.

The QC4OpenX framework contains the following report modules:

- TextReport - generating a human readable text file with all issues
- ReportGUI (Windows only)

Using the ReportGUI

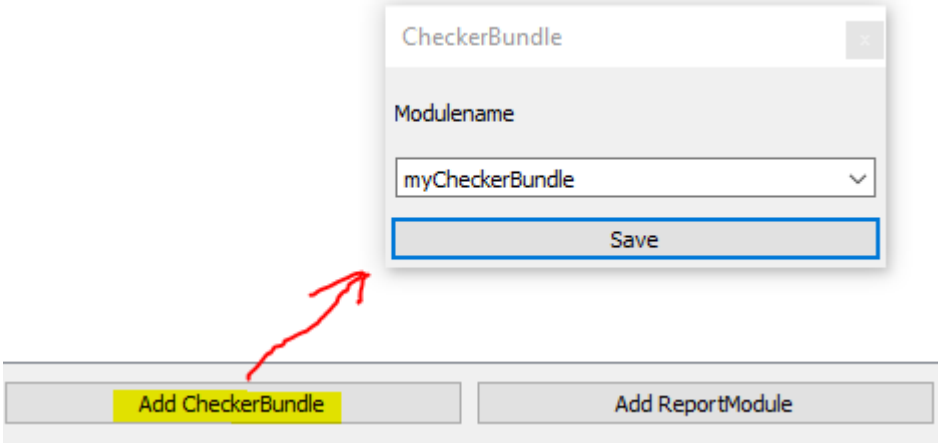
This GUI can be used to filter reported issues. Just click on a CheckerBundle entry to show all issues from this CheckerBundle. The list of Checkers can be used to filter issues from one Checker in one CheckerBundle. The window on the bottom left contains all the information about the selected issue. The source code on the right side moves to the location in the file, if the issue contains a valid FileLocation.



Additionally it is possible to link issues that correspond to a 3D error with a viewer application. A 3D error is an error in the 3D model of the road network - it gets marked with a little icon in the left part of the ReportGUI window. This viewer is not part of the Open Source distributution of the QC4OpenX framework, but we provide a binding mechanism which let's you provide your own implementation. Please refer to the documentation for details.

Add Self Implemented CheckerBundles and ReportModules

You can add your own CheckerBundles and ReportModules to the configuration. Just add the executables with the graphical user interface. There are some templates which are provided with the current version. You are free to choose another executable, just specify the name without the file extension.



Requirements for your own CheckerBundle can be found in the User defined modules documentation.

Automation and Deployment

There are several ways for automating the checking process

Open user interface and load the `DefaultConfiguration.xml`

```
> ConfigGUI.exe
```

Open user interface and load a specified configuration

```
> ConfigGUI.exe myConfiguration.xml
```

Open user interface, load the specified configuration and run the process without user input

```
> ConfigGUI.exe myConfiguration.xml -autostart
```

Open the application with `myConfiguration.xml` and a given XODR, which is under test. Autostart enables the automatic mode

```
> ConfigGUI.exe -config myConfiguration.xml -xodr myTrack.xodr [-autostart]
```

Open the application with `myConfiguration.xml` and a given XOSC, which is under test. Autostart enables the automatic mode

```
> ConfigGUI.exe -config myConfiguration.xml -xosc myScenario.xosc -autostart
```

The easiest way is to adapt and use the two delivered batch scripts `CheckXodr.bat` and `CheckXosc.bat`. They automate the complete process and execute all delivered CheckerBundles, create a text based report and open the ReportGUI for filtering the results.

Linux

You can use the shell scripts `CheckXodr.sh` and `CheckXosc.sh` in combination with a self-written configuration file for execution and generation of a textual report.

Viewer Interface

Link with the Interface

As described in the section "Using the ReportGUI" of Using the Checker Framework, it is possible to show issues in a 3D Viewer, if they have a valid position link. In general QC4OpenX doesn't provide a 3D Viewer but a C-interface to connect your own 3D Viewer to the ReportGUI. The ReportGUI loads viewer plugins in the form of shared libraries from the folder `bin/plugin`. A plugin needs to implement the interface `include/viewer/iConnector.h` and provide it via common C-API `__declspec(dllexport)` mechanism. The ReportGUI will load the library during startup and use your viewer to show issues graphically.

The C-interface declares the following functions, which are called by the ReportGUI:

```
bool StartViewer();
bool Initialize(const char* xoscPath, const char* xodrPath);
bool AddIssue(void * issueToAdd);
bool ShowIssue(void * itemToShow, void* locationToShow);
const char* GetName();
bool CloseViewer();
const char* GetLastErrorMessage();
```

This functions should be implemented inside your 3D viewer library to make your 3D viewer compatible with the Report GUI. If the shared library is not loadable or one of the functions isn't defined, the ReportGUI will show an error and the viewer won't be available.

It is not possible to implement and start multiple viewers. Starting a second viewer automatically closes the currently active one. A viewer is started via the context menu **File -> Start Viewer**.

If you start a viewer the following functions are called in this order:

1. StartViewer
2. Initialize
3. AddIssue (will be called for each issue found)

If an error occurs during the startup process, the ReportGUI will call GetLastErrorMessage to print out the error in the ReportGUI itself.

ShowIssue is triggered if you click on a RoadLocation issue in the ReportGUI. It will send the clicked issue and its location to the viewer.

If the ReportGUI is closed a currently active Viewer receives the closeViewer call.

Viewer Example

A demo viewer is provided as source code under `examples/viewer_example`. When you open the GUI, it looks like in the image above and you can start the viewer from the File menu. Click on issues that represent a

3D error and find the function calls inside the console window. Your own viewer interface implementation can use this information to show the error in 3D.