Project Number 101017258

D7.2 Tools for Generation of Runtime EDDIs

Version 1.0
30 June 2022
Final

Public Distribution

Fraunhofer IESE and University of Hull

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

PROJECT PARTNER CONTACT INFORMATION

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

Document Control

Version	Status	Date
0.1	Initial outline	10 May 2022
0.2	Second version; sections 3.1-3.6 added	1 June 2022
0.3	Input from Hull for sections 3.3 and 3.5	14 June 2022
0.4	Sections 1, 2, 4, 5 added	20 June 2022
0.5	Formal check (tables, figures, abbreviations, references, …)	22 June 2022
0.6	Final version for QA in project	23 June 2022
0.7	Review from Aero41 received	27 June 2022
0.8	Review from TTS received	29 June 2022
1.0	Review changes integrated, ready for submission	30 June 2022

Table of Contents

TABLE OF FIGURES

EXECUTIVE SUMMARY

Executable Digital Dependability Identities (EDDIs) are meant to be deployed across significantly diverse applications and Multi-Robot System (MRS) architectures. In order to engineer EDDIs that supervise the dependability of MRS at runtime and which are assured themselves at design-time, a holistic methodological approach is required.

In this deliverable, the toolchain for the generation of runtime EDDIs is explained. The toolchain assumes a design-time EDDI has been engineered with design-time dependability engineering tools. Based on this model, platform-independent and platform-dependent software components are semi-automatically generated, which contain the functionality to dynamically supervise dependability properties of a multi-robot system. Specifically, in this first iteration of the SESAME toolchain, the Robot Operating System (ROS) has been selected as an exemplary target platform for robotic applications. For each of the generators, detailed information is given on the generator functionality, used technologies, important design decisions. In addition, tutorials are provided for setting up the required tool environment and walking through the generation of exemplary runtime EDDI subcomponents and the final MRS-level runtime EDDI, which can be deployed into existing ROS applications.

Since tightly related to this deliverable, further information on the technical design-time specification of EDDIs can be found in deliverables D4.2. In addition, the conceptual background of the techniques realized technically in this document can be found in deliverable D7.1.

List of Abbreviations

EDDI	Executable Digital Dependability Identity	ODE	Open Dependability Exchange
ODD	Operational Design Domain	ROS	Robot Operating System
ConSerts	Conditional Safety Certificates	XML	Extensible Markup Language
YAML	Yaml Ain't Markup Language	EGL	Epsilon Generation Language
SESAME	Secure and Safe Multi-Robot Systems	MRS	Multi Robot System
JAR	Java Archive	API	Application Programming Interface
XDSL	XML-based Bayesian Network format of the GeNIe toolset	GeNie	Commercial tool for Bayesian network modelling and inference
IDE	Integrated Development Environment	DT	Design Time
PCA	Principle Component Analysis	Hz	Hertzs
EBNF	Extended Backus–Naur Form	RtE	Runtime Evidence
MROS	Model Based ROS		

1. INTRODUCTION

This deliverable reports on the developed tool set for the (semi-)automatic generation of the runtime parts of Executable Digital Dependability Identities (EDDI). As such, it describes the realization and usage of the technical components, which enable the transformation of EDDIs in the shape of design-time models to executable pieces of software that can be deployed to multi robot systems (MRS) in order to perform dynamic risk management.

The conceptual background of the constituents of an EDDI as envisioned by SESAME are described in deliverable D7.1 (see Figure 1). Concretely, these constituents collaboratively performing dynamic risk management are 1. Dynamic Safety Capability Assessment with Conditional Safety Certificates (*ConSerts*), 2. Dynamic Risk Assessment with Bayesian networks, 3. Dynamic Reliability Assessment based on the *SafeDrones* tool, 4. Perception Uncertainty Monitoring with the *SafeML* tool and 5. conditional event monitoring to realize a typed communication interface between the *runtime* EDDI and the nominal functionality of the MRS. Regarding the scope of the tools described in this deliverable, the models consumed by these components are assumed to be present in the shape of a *design-time* EDDI, which is a .xml file conforming to the technical Open Dependability Exchange (ODE) metamodel specification described in D4.2. Thus, the toolset in the present document realizes the pipeline to make the engineered runtime models executable (i.e. generate components that can infer the runtime models based on runtime-available information) and deploy them into common robotic platform architectures.

In the first development iteration, the Robot Operating System (ROS) has been selected as the exemplary target runtime environment, to which the EDDI shall be deployed in. The reason for this decision is the usage of ROS in several SESAME use cases and the general spread of ROS in the robotics domain. Thus, by having support for deploying runtime EDDIs to ROS applications, the transfer of EDDIs to industrial applications is facilitated. By splitting the runtime EDDI generator pipeline in platform-independent and platform-dependent parts, the extension towards other runtime environments is conceptually and technically simplified.

The developed tools provide users with control over the transformation, generation and deployment processes. The tools described in this deliverable can be found in the SESAME public GitHub repository at: https://github.com/sesame-project/

The rest of the deliverable is structured as follows. In Section 2, the big picture of the runtime EDDI generation toolchain is described in the context of its inputs and outputs. In Section 3, we provide an explanation of the technical details of each generator for the different runtime EDDI components. This includes the description of the generator functionality, technical formats, used technologies and important design decisions. In addition, the end-user perspective is considered by providing a step-by-step tutorial for setting up the tool environment as well as using the tool to get the runtime EDDIs generated. While Section 3 focusses on the individual components of the runtime EDDI, in Section 4 we discuss, based on an example, how the overall toolchain is used to generate the runtime EDDI as a whole. We conclude in Section 5 summarizing the deliverable’s main points and outlining next steps.

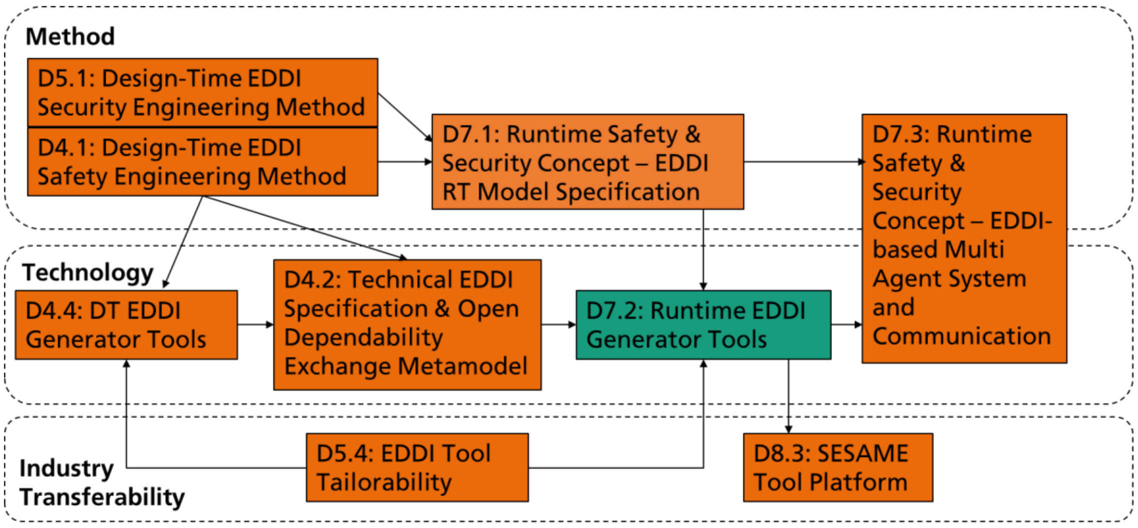

Method
D5.1: Design-Time EDDI Security Engineering Method
D4.1: Design-Time EDDI Safety Engineering Method
D7.1: Runtime Safety & Security Concept – EDDI RT Model Specification
D7.3: Runtime Safety & Security Concept – EDDI-based Multi Agent System and Communication
Technology
D4.4: DT EDDI Generator Tools
D4.2: Technical EDDI Specification & Open Dependability Exchange Metamodel
D7.2: Runtime EDDI Generator Tools
Industry Transferability
D5.4: EDDI Tool Tailorability
D8.3: SESAME Tool Platform

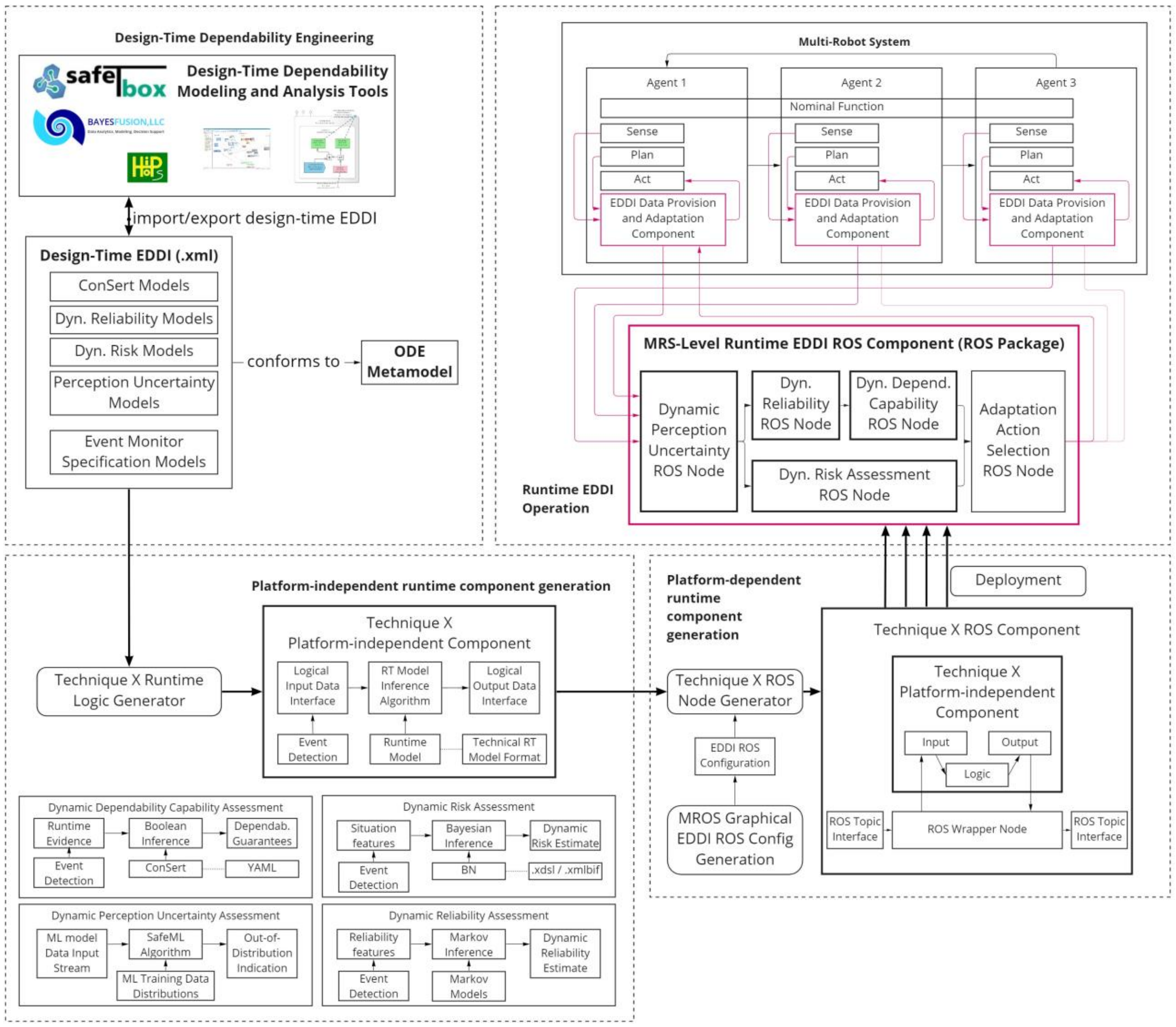
Figure 1 Relationship of D7.2 to other deliverables

2. RUNTIME EDDI GENERATION TOOLCHAIN ARCHITECTURE

The generation of runtime EDDIs, i.e. executable software that performs dynamic dependability management, happens still at design-time and assumes the existence of a design-time EDDI (Figure 2). The design-time EDDI is an .xml based file, which conforms to the Open Dependability Exchange (ODE) metamodel, whose technical specification is described in deliverable D4.2. In addition to the runtime models depicted in Figure 2, the design-time EDDI contains as well models, which have been used as intermediate artefacts to systematically derive the runtime models and thereby provide a basis for their dependability assurance. Technically, this means, a user, who wants to use the runtime EDDI generation toolchain, has available one or more runtime models in the shape of design-time EDDI packages.

The next step towards a runtime EDDI is the generation of a platform-independent runtime component for each of the SESAME dynamic dependability management techniques. Specifically, these are dynamic dependability capability assessment with Conditional Safety Certificate models (ConSerts), situation-aware dynamic risk assessment with Bayesian network models, dynamic reliability assessment based on Markov models (SafeDrones framework) and dynamic perception uncertainty assessment based on the SafeML framework). Apart from a machine-processable representation of the runtime model to be inferred during runtime, the platform-independent component for each technique needs to contain logic for performing the inference task and explicit formats for required input data and provided output data. In the cases of Bayesian networks, ConSerts and Markov models, the input data is already on a higher abstraction level than concretely perceivable variables. Therefore, an event detection logic is required that evaluates higher-level events from low-level perceived variables.

After the generation of a platform-independent component, the functionality for inference on particular runtime models is given. However, involved formats, algorithms and used technologies usually differ with respect to the techniques. In addition, the execution frequency and data interface types differ and not each technique is required for applications. Since this is not a new problem for runtime EDDI execution, middleware exists that is handling the dynamic integration of components, e.g. the Robot Operating System (ROS). There exist various different architectural middleware taking care of orchestrating component integration and inter-component communication. For the first iteration of the SESAME toolchain, we selected ROS as an example runtime execution middleware. Since the platform-independent components are functionally working after generation step 1, it is deemed desirable to have a second generation step, which wraps the platform-independent components into platform-dependent components and therefore make them ready to be integrated in a particular target platform. For this purpose, a set of scripts has been developed to automatically provide a ROS wrapping node around platform-independent components.



Design-Time Dependability Engineering
Design-Time Dependability Modeling and Analysis Tools
safetbox
BAYESFUSION,LLC
import/export design-time EDDI
Design-Time EDDI (.xml)
ConSert Models
Dyn. Reliability Models
Dyn. Risk Models
Perception Uncertainty Models
Event Monitor Specification Models
conforms to
ODE Metamodel
Multi-Robot System
Agent 1
Agent 2
Agent 3
Nominal Function
Sense
Plan
Act
EDDI Data Provision and Adaptation Component
MRS-Level Runtime EDDI ROS Component (ROS Package)
Dynamic Perception Uncertainty ROS Node
Dyn. Reliability ROS Node
Dyn. Depend. Capability ROS Node
Dyn. Risk Assessment ROS Node
Adaptation Action Selection ROS Node
Runtime EDDI Operation
Platform-independent runtime component generation
Technique X Runtime Logic Generator
Technique X Platform-independent Component
Logical Input Data Interface
RT Model Inference Algorithm
Logical Output Data Interface
Event Detection
Runtime Model
Technical RT Model Format
Platform-dependent runtime component generation
Deployment
Technique X ROS Node Generator
EDDI ROS Configuration
MROS Graphical EDDI ROS Config Generation
Technique X ROS Component
Technique X Platform-independent Component
Input
Output
Logic
ROS Topic Interface
ROS Wrapper Node
ROS Topic Interface
Dynamic Dependability Capability Assessment
Runtime Evidence
Boolean Inference
Dependab. Guarantees
Event Detection
ConSert
YAML
Dynamic Risk Assessment
Situation features
Bayesian Inference
Dynamic Risk Estimate
Event Detection
BN
.xdsl / .xmlbif
Dynamic Perception Uncertainty Assessment
ML model Data Input Stream
SafeML Algorithm
Out-of-Distribution Indication
ML Training Data Distributions
Dynamic Reliability Assessment
Reliability features
Markov Inference
Dynamic Reliability Estimate
Event Detection
Markov Models

Figure 2 SESAME Runtime EDDI Generator Tool Architecture

This ROS generator takes the EDDI ROS configuration as input, which specifies information about ROS input/output interfaces that fit with the ROS topic interfaces of the MRS application the runtime EDDI shall be integrated in. In order to simplify the integration for the engineer, a static source code analyser for ROS applications[1] in combination with an Eclipse-based graphical editor has been connected to the runtime toolchain to graphically perform the integration between existing ROS application and runtime EDDI, especially with respect to data interfaces.

After the platform-dependent ROS component generation, a ROS package exists, which contains executable ROS nodes that:

- represent a mechanism for dynamic dependability management for a concrete MRS that has been engineered
- is compatible with existing input and output ROS messages provided by the multi-robot system agents
- are fully and transparently traceable back to the dependability engineering artefacts used to engineer and assure the runtime EDDIs
- were generated to a large extent based on automated tools and therefore the process is efficient and less error-prone than with manual development of dependability mechanisms

The generator components on the platform-independent level are described in sections 3.1-3.5, the generator component on the platform-dependent level is described in section 3.6.

3. Runtime EDDI Generation Components

3.1 ConSert Component Generator

The ConSert Component Generator consists of three major parts. The toolchain takes the EDDI model in XML as an input. From the design-time EDDI XML file, the ConSerts are derived and described in YAML files. Those YAML files are then used as inputs to create the ConSert Monitor as a Python WHL binary via consert-rs and Maturin and the corresponding Python representation as an EDDI Monitor. An overview of this toolchain is shown in Figure 3 (blue boxes represent artefacts and green boxes represent transformation scripts) and the corresponding steps are described in detail in the following sections.

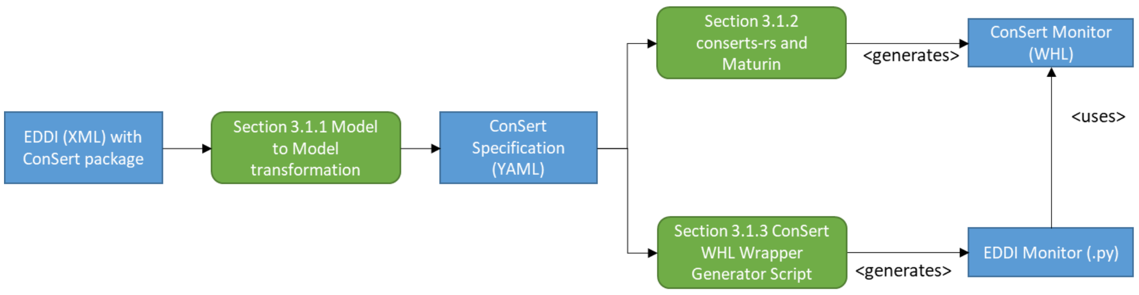
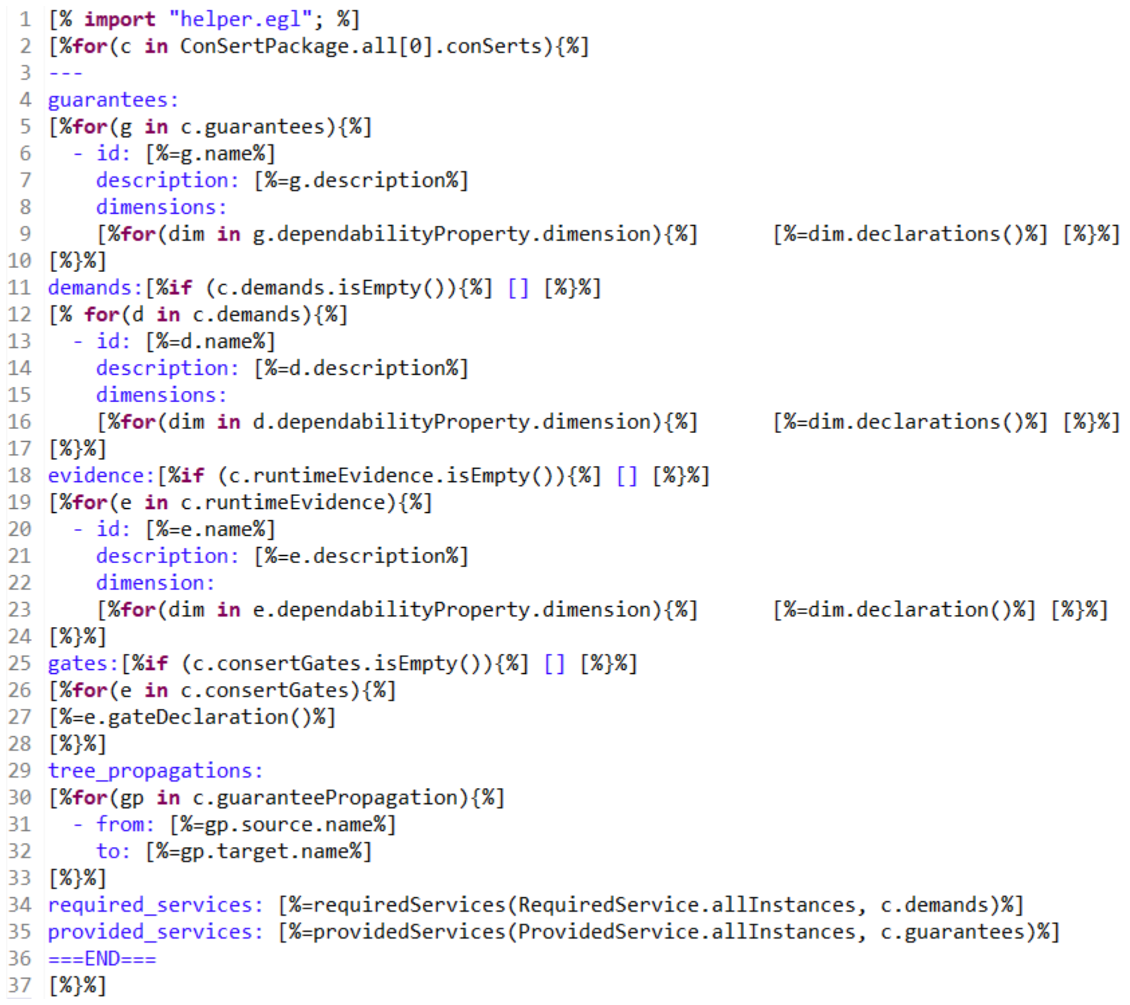

EDDI (XML) with ConSert package
Section 3.1.1 Model to Model transformation
ConSert Specification (YAML)
Section 3.1.2 conserts-rs and Maturin
<generates>
ConSert Monitor (WHL)
<uses>
Section 3.1.3 ConSert WHL Wrapper Generator Script
<generates>
EDDI Monitor (.py)

Figure 3 EDDI ConSert monitor toolchain overview

[1] https://github.com/ipa320/ros-model/

3.1.1 ConSert Model Generation

First, the EDDI XML model is transformed into YAML files that describe the specified ConSerts. These YAML files are in a representation compatible with that of the conserts-rs[2] library, which includes the evaluation of ConSert models at runtime. To this end, we use Eclipse Epsilon[3], in which several Java-based scripting languages exist for automating common model-based software engineering tasks, as code generation, model-to-model transformation and model validation. More precisely, in this step the Epsilon Generation Language[4] (EGL) is used to extract the model-specific information from the ConSert models contained in the EDDI XML model required for generating the corresponding YAML files. Figure 4 shows an excerpt of the EGL program code, which invokes helper functions in an additional EGL program (i.e. "helper.egl") to realize this transformation.



```
[% import "helper.egl"; %]
[%for(c in ConSertPackage.all[0].conSerts){%]
---
guarantees:
[%for(g in c.guarantees){%]
  - id: [%=g.name%]
    description: [%=g.description%]
    dimensions:
    [%for(dim in g.dependabilityProperty.dimension){%]      [%=dim.declarations()%] [%}%]
[%}%]
demands:[%if (c.demands.isEmpty()){%] [] [%}%]
[% for(d in c.demands){%]
  - id: [%=d.name%]
    description: [%=d.description%]
    dimensions:
    [%for(dim in d.dependabilityProperty.dimension){%]      [%=dim.declarations()%] [%}%]
[%}%]
evidence:[%if (c.runtimeEvidence.isEmpty()){%] [] [%}%]
[%for(e in c.runtimeEvidence){%]
  - id: [%=e.name%]
    description: [%=e.description%]
    dimension:
    [%for(dim in e.dependabilityProperty.dimension){%]      [%=dim.declaration()%] [%}%]
[%}%]
gates:[%if (c.consertGates.isEmpty()){%] [] [%}%]
[%for(e in c.consertGates){%]
[%=e.gateDeclaration()%]
[%}%]
tree_propagations:
[%for(gp in c.guaranteePropagation){%]
  - from: [%=gp.source.name%]
    to: [%=gp.target.name%]
[%}%]
required_services: [%=requiredServices(RequiredService.allInstances, c.demands)%]
provided_services: [%=providedServices(ProvidedService.allInstances, c.guarantees)%]
===END===
[%}%]
```

Figure 4 Excerpt of the EGL program code used to transform an EDDI XML model to YAML files

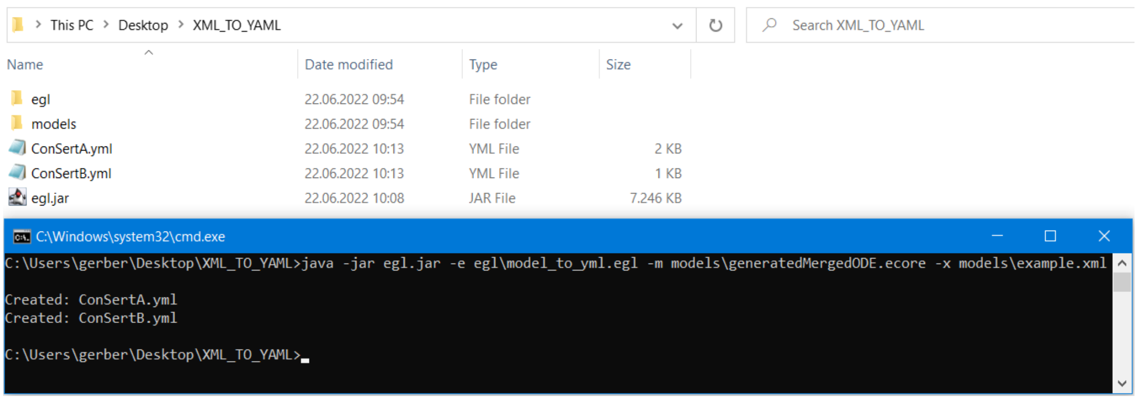
To enable a convenient transformation, in the absence of an existing Eclipse Epsilon development environment, a Java Archive (JAR) has been compiled to execute Epsilon

[2] https://real-time-conserts.feuerberg.iese.fraunhofer.de/opus/conserts-rs.html
[3] https://www.eclipse.org/epsilon/
[4] https://www.eclipse.org/epsilon/doc/egl/

functionality standalone based on an Application Programming Interface (API). It can be used with the following command:

```
java -jar egl.jar -e <egl_script> -m <ecore model_file> -x <xml_file>
```

Executing this command will create the YAML files for the ConSerts defined in the EDDI XML model, as depicted in Figure 5. In this example, the directory "egl" contains the required egl files (i.e. "model_to_yaml.egl" and "helper.egl") and "models" contains the ODE metamodel (i.e. "generatedMergedODE.ecore") as well as a model (i.e. "example.xml") containing two example ConSerts (i.e. "ConSertA", "ConSertB").



This PC > Desktop > XML_TO_YAML

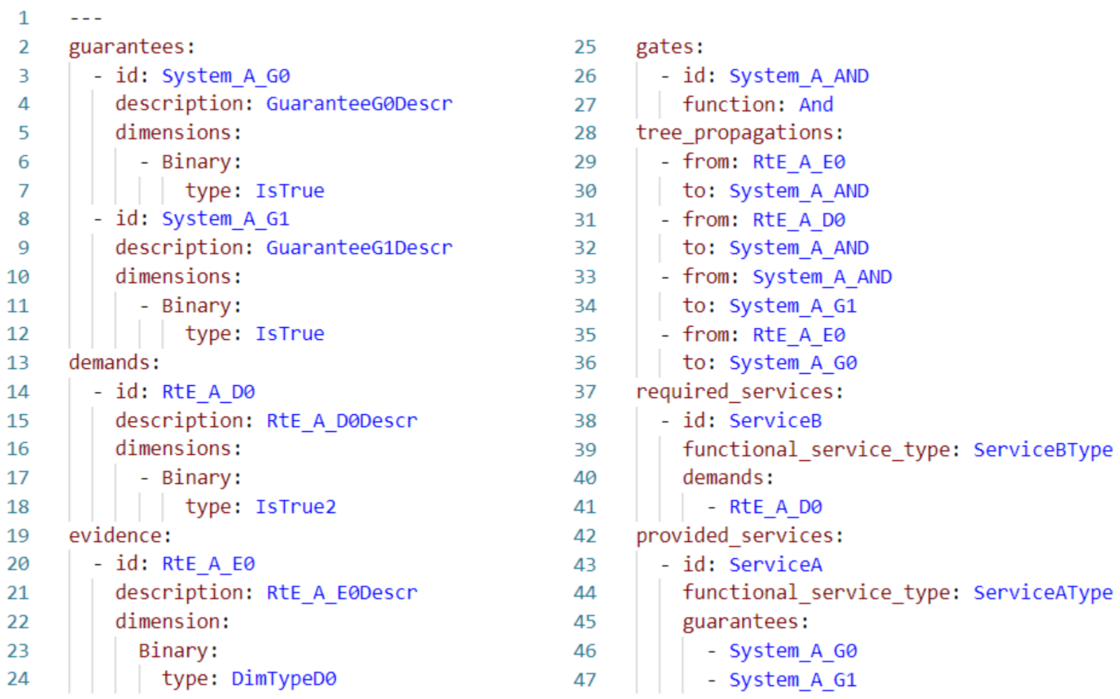
Name	Date modified	Type	Size
egl	22.06.2022 09:54	File folder	
models	22.06.2022 09:54	File folder	
ConSertA.yml	22.06.2022 10:13	YML File	2 KB
ConSertB.yml	22.06.2022 10:13	YML File	1 KB
egl.jar	22.06.2022 10:08	JAR File	7.246 KB

C:\Windows\system32\cmd.exe

```
C:\Users\gerber\Desktop\XML_TO_YAML>java -jar egl.jar -e egl\model_to_yml.egl -m models\generatedMergedODE.ecore -x models\example.xml

Created: ConSertA.yml
Created: ConSertB.yml

C:\Users\gerber\Desktop\XML_TO_YAML>
```

Figure 5 Creation of YAML files for ConSerts defined in an EDDI XML model

An example of the content of such a YAML file can be seen in Figure 6. Note that not all ConSert-related information that is part of the design-time EDDI is also part of generated YAML file. Only the minimal information required to infer ConSerts based on runtime information is contained. For instance, design-time evidences which enable the traceability between the runtime ConSerts and the corresponding design-time safety engineering artefacts (e.g. safety analysis or safety concept models) is not available anymore.



```yaml
---
guarantees:
  - id: System_A_G0
    description: GuaranteeG0Descr
    dimensions:
      - Binary:
          type: IsTrue
  - id: System_A_G1
    description: GuaranteeG1Descr
    dimensions:
      - Binary:
          type: IsTrue
demands:
  - id: RtE_A_D0
    description: RtE_A_D0Descr
    dimensions:
      - Binary:
          type: IsTrue2
evidence:
  - id: RtE_A_E0
    description: RtE_A_E0Descr
    dimension:
      Binary:
        type: DimTypeD0
gates:
  - id: System_A_AND
    function: And
tree_propagations:
  - from: RtE_A_E0
    to: System_A_AND
  - from: RtE_A_D0
    to: System_A_AND
  - from: System_A_AND
    to: System_A_G1
  - from: RtE_A_E0
    to: System_A_G0
required_services:
  - id: ServiceB
    functional_service_type: ServiceBType
    demands:
      - RtE_A_D0
provided_services:
  - id: ServiceA
    functional_service_type: ServiceAType
    guarantees:
      - System_A_G0
      - System_A_G1
```

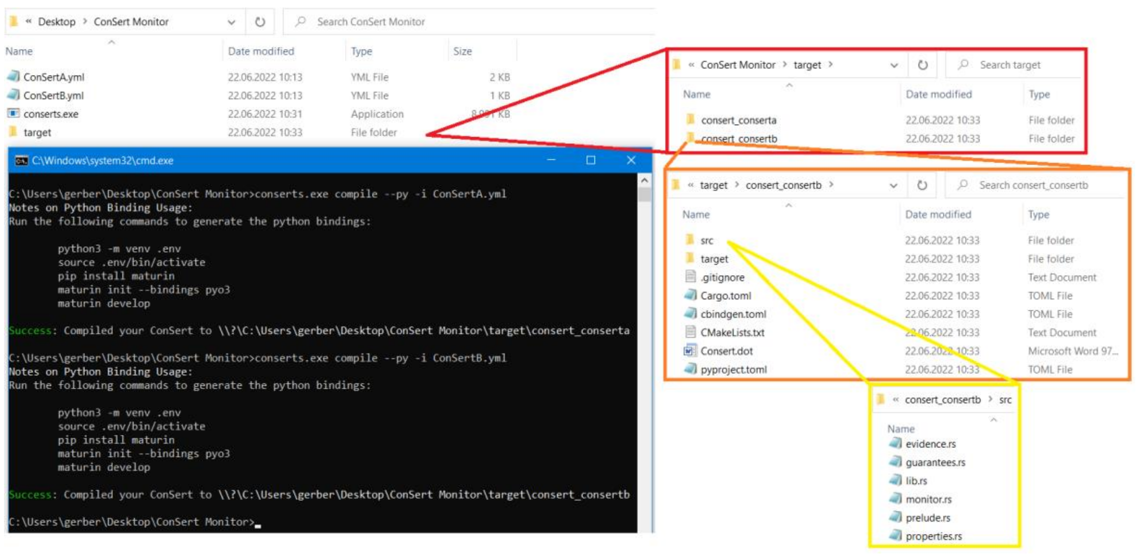
Figure 6 Minimal example of a ConSert described in a YAML file

3.1.2 ConSert Monitor Generation

In order to enable the evaluation of ConSert models at runtime, ConSert monitors are generated in Python for each of the YAML files. For this, we use conserts-rs, which can process a YAML file and create annotated Rust[5] code in a directory named "consert_<consert_name>", located in "target" (see Figure 7). The Rust programming language provides tools and guidance to build fast, efficient and correct code. Generating the Rust code with the Python bindings is done executing the following command from command line:

```
conserts.exe compile --py -i <consert_name.yml>
```

[5] https://www.rust-lang.org/



« Desktop › ConSert Monitor

Name	Date modified	Type	Size
ConSertA.yml	22.06.2022 10:13	YML File	2 KB
ConSertB.yml	22.06.2022 10:13	YML File	1 KB
conserts.exe	22.06.2022 10:31	Application	[illegible] KB
target	22.06.2022 10:33	File folder	

C:\Windows\system32\cmd.exe

```
C:\Users\gerber\Desktop\ConSert Monitor>conserts.exe compile --py -i ConSertA.yml
Notes on Python Binding Usage:
Run the following commands to generate the python bindings:

        python3 -m venv .env
        source .env/bin/activate
        pip install maturin
        maturin init --bindings pyo3
        maturin develop

Success: Compiled your ConSert to \\?\C:\Users\gerber\Desktop\ConSert Monitor\target\consert_conserta

C:\Users\gerber\Desktop\ConSert Monitor>conserts.exe compile --py -i ConSertB.yml
Notes on Python Binding Usage:
Run the following commands to generate the python bindings:

        python3 -m venv .env
        source .env/bin/activate
        pip install maturin
        maturin init --bindings pyo3
        maturin develop

Success: Compiled your ConSert to \\?\C:\Users\gerber\Desktop\ConSert Monitor\target\consert_consertb

C:\Users\gerber\Desktop\ConSert Monitor>_
```

« ConSert Monitor › target ›

Name	Date modified	Type
consert_conserta	22.06.2022 10:33	File folder
consert_consertb	22.06.2022 10:33	File folder

« target › consert_consertb ›

Name	Date modified	Type
src	22.06.2022 10:33	File folder
target	22.06.2022 10:33	File folder
.gitignore	22.06.2022 10:33	Text Document
Cargo.toml	22.06.2022 10:33	TOML File
cbindgen.toml	22.06.2022 10:33	TOML File
CMakeLists.txt	22.06.2022 10:33	Text Document
Consert.dot	22.06.2022 10:33	Microsoft Word 97...
pyproject.toml	22.06.2022 10:33	TOML File

« consert_consertb › src

Name
evidence.rs
guarantees.rs
lib.rs
monitor.rs
prelude.rs
properties.rs

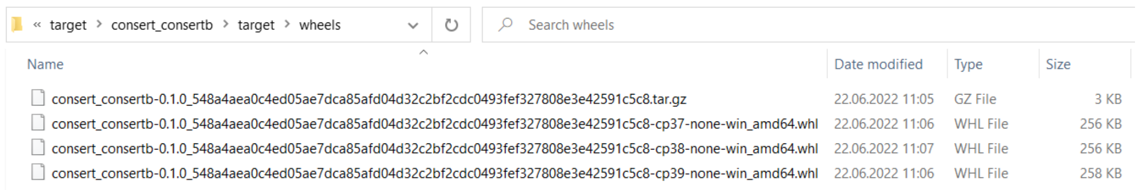
Figure 7 Generated Rust code for ConSertB

During this activity, the “--py” argument results in Rust code that has additional annotations, which are required by PyO3/maturin[6] to generate a native Python module (wheel file) in the next step. To be able to use maturin, it can be installed via pip as follows:

```
pip install maturin
```

For generating a wheel file (.whl), containing the Python-based ConSert monitor module, the following command needs to be executed in the directory “consert_<consert_name>”:

```
maturin build
```



« target › consert_consertb › target › wheels

Name	Date modified	Type	Size
consert_consertb-0.1.0_548a4aea0c4ed05ae7dca85afd04d32c2bf2cdc0493fef327808e3e42591c5c8.tar.gz	22.06.2022 11:05	GZ File	3 KB
consert_consertb-0.1.0_548a4aea0c4ed05ae7dca85afd04d32c2bf2cdc0493fef327808e3e42591c5c8-cp37-none-win_amd64.whl	22.06.2022 11:06	WHL File	256 KB
consert_consertb-0.1.0_548a4aea0c4ed05ae7dca85afd04d32c2bf2cdc0493fef327808e3e42591c5c8-cp38-none-win_amd64.whl	22.06.2022 11:07	WHL File	256 KB
consert_consertb-0.1.0_548a4aea0c4ed05ae7dca85afd04d32c2bf2cdc0493fef327808e3e42591c5c8-cp39-none-win_amd64.whl	22.06.2022 11:06	WHL File	258 KB

Figure 8 Created Python wheel files from annotated Rust code for ConSertB using maturin build

In order to use a ConSert monitor, the resulting .whls files, located in “consert_<consert_name>/target/wheels” (see Figure 8), are then installed with:

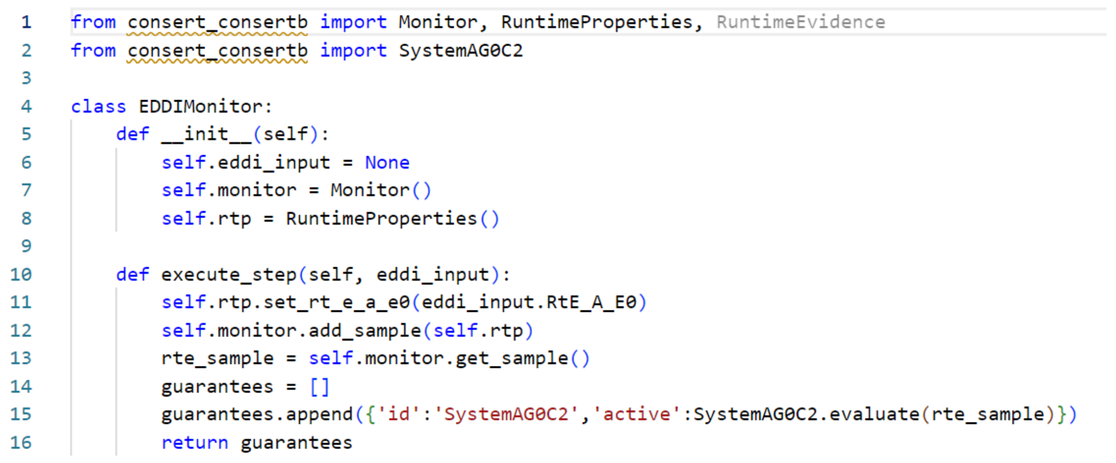
```
pip install consert_<consert_name>.whl
```

3.1.3 EDDI Monitor Generation

The consert_monitor.py script generates an EDDI monitor in Python based on a ConSert YAML file. As input parameters, the ConSert YAML file and the path where

[6] https://github.com/PyO3/maturin

the generated monitor script should be saved. The generated code imports the previously generated bindings to access setter and other functions of the ConSert Monitor.



```
 1  from consert_consertb import Monitor, RuntimeProperties, RuntimeEvidence
 2  from consert_consertb import SystemAG0C2
 3
 4  class EDDIMonitor:
 5      def __init__(self):
 6          self.eddi_input = None
 7          self.monitor = Monitor()
 8          self.rtp = RuntimeProperties()
 9
10      def execute_step(self, eddi_input):
11          self.rtp.set_rt_e_a_e0(eddi_input.RtE_A_E0)
12          self.monitor.add_sample(self.rtp)
13          rte_sample = self.monitor.get_sample()
14          guarantees = []
15          guarantees.append({'id':'SystemAG0C2','active':SystemAG0C2.evaluate(rte_sample)})
16          return guarantees
```

Figure 9 Generated EDDI Monitor example file

In Figure 9, the generated EDDI Monitor is shown. It was generated based on the minimal ConSert described in a YAML which is shown in Figure 6. The ConSert has one runtime evidence (RtE_A_E0) and one guarantee (SystemAG0C2). At the top, the generated ConSert Monitor dependencies are imported. During the “execute_step” method, the runtime evidence and demands are set (line 11). Then, the guarantees are evaluated, gathered in a list and returned. Later in section 3.6 it is explained how the result is further processed within a ROS node.

In order to generate the EDDI Monitor the “consert_monitor.py” script can be executed from command line with the following syntax:

```
python consert_monitor.py -t <target dir> -c <consert.yml>
```

Thereby, the <target dir> argument represents the directory where the final EDDI monitor script should be saved. As a second parameter, the path to the ConSert YAML file must be set.

In order to execute the EDDI monitor, the ConSert monitor wheel files (see above) must be installed on the local machine.

3.2 DYNAMIC RISK ASSESSMENT COMPONENT GENERATOR

The Bayesian network component generator enables to generate Bayesian network Python monitors which can be used for runtime inference of a specific Bayesian network in the context of performing situation-aware dynamic risk assessment. It requires a design-time EDDI Bayesian network model as an input and provides the generated Python component as an output that can perform the runtime inference.

The generation procedure consists of three major steps. The toolchain takes the EDDI model (XML-like format) representing one or more Bayesian networks as an input. From the EDDI file, the runtime representation of the Bayesian network is generated. Therefore, the .xdsl file format (XML-like format) is used. .xdsl is the proprietary

Bayesian network format from the BayesFusion[7] GeNIe Modeler tool capable of modeling and analyzing Bayesian networks. This runtime model representation is then used as input for the subsequent generation of the Python Bayesian network EDDI monitor package. Finally, the monitor must be manually adapted to be tailored to the concrete use case. This includes the discretization of the inputs and the mapping to the evidence. An overview of the process is given in Figure 10 (a). These steps are described in detail in the following sections.

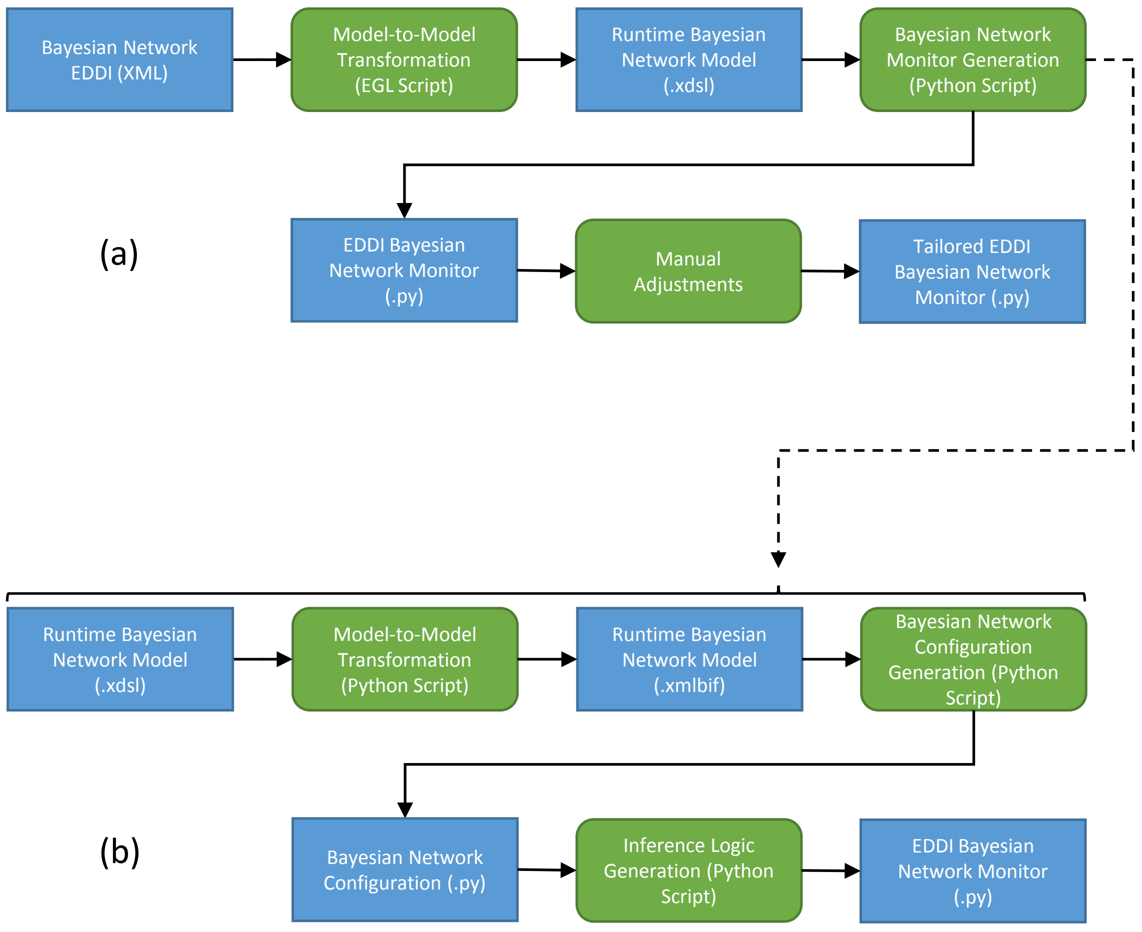

Bayesian Network EDDI (XML)
Model-to-Model Transformation (EGL Script)
Runtime Bayesian Network Model (.xdsl)
Bayesian Network Monitor Generation (Python Script)
(a)
EDDI Bayesian Network Monitor (.py)
Manual Adjustments
Tailored EDDI Bayesian Network Monitor (.py)
Runtime Bayesian Network Model (.xdsl)
Model-to-Model Transformation (Python Script)
Runtime Bayesian Network Model (.xmlbif)
Bayesian Network Configuration Generation (Python Script)
(b)
Bayesian Network Configuration (.py)
Inference Logic Generation (Python Script)
EDDI Bayesian Network Monitor (.py)

Figure 10 Process of generating a Bayesian network monitor given a Bayesian network in form of an EDDI. (a) Shows the whole process, (b) shows the details of the actual Python monitor generation. The artifacts are shown in blue and the processes are shown in green.

3.2.1 EDDI Model to Runtime Model Transformation

First, the EDDI Bayesian network model is transformed into a runtime model, i.e., a .xdsl file that is representing the respective Bayesian network. This format is compatible with GeNIe Modeler[8] and the SMILE[9] C++ library for programmatic manipulation and analysis of Bayesian networks.

[7] https://www.bayesfusion.com/
[8] https://www.bayesfusion.com/genie/
[9] https://www.bayesfusion.com/smile/

Further, this file format is internally transformed to the .xmlbif format supported by the open-source pgmpy[10] Python library using a Python script. The pgmpy library is used by the generated EDDI monitors for the runtime inference. This design decision was taken to make the SESAME DRA technology independent of commercial tools. Thus, the inference is performed with an open-source library that can be fed either via .xmlbif (pgmpy-compatible) or .xdsl (GeNIe-compatible). This file format translation from .xdsl to .xmlbif is hidden from the user and automatically triggered in the EDDI monitor generation explained in Section 3.2.2.

Coming back to the actual task at hand, the transformation from the EDDI model to the runtime format (.xdsl) is implemented facilitating the Eclipse Epsilon Generation Language (EGL)[11]. EGL is building on the Eclipse Modeling Framework (EMF)[12] and enables easy working with the ODE metamodel. So, the EGL script extracts the information from the EDDI Bayesian network file (using the corresponding ODE's .ecore metamodel file) and maps the information to the file content of the .xdsl file. During the .xdsl creation process the order of the nodes and conditional probability values in the Bayesian network are checked and adapted accordingly to assure that the parametrization of the network stays the same.

For running this EGL script, it is possible to either install the Eclipse Integrated Development Environment (IDE)[13] and set it up accordingly to work with Epsilon, or it is possible to use the provided JAR file to be independent of the IDE setup. The second option is the recommended way.

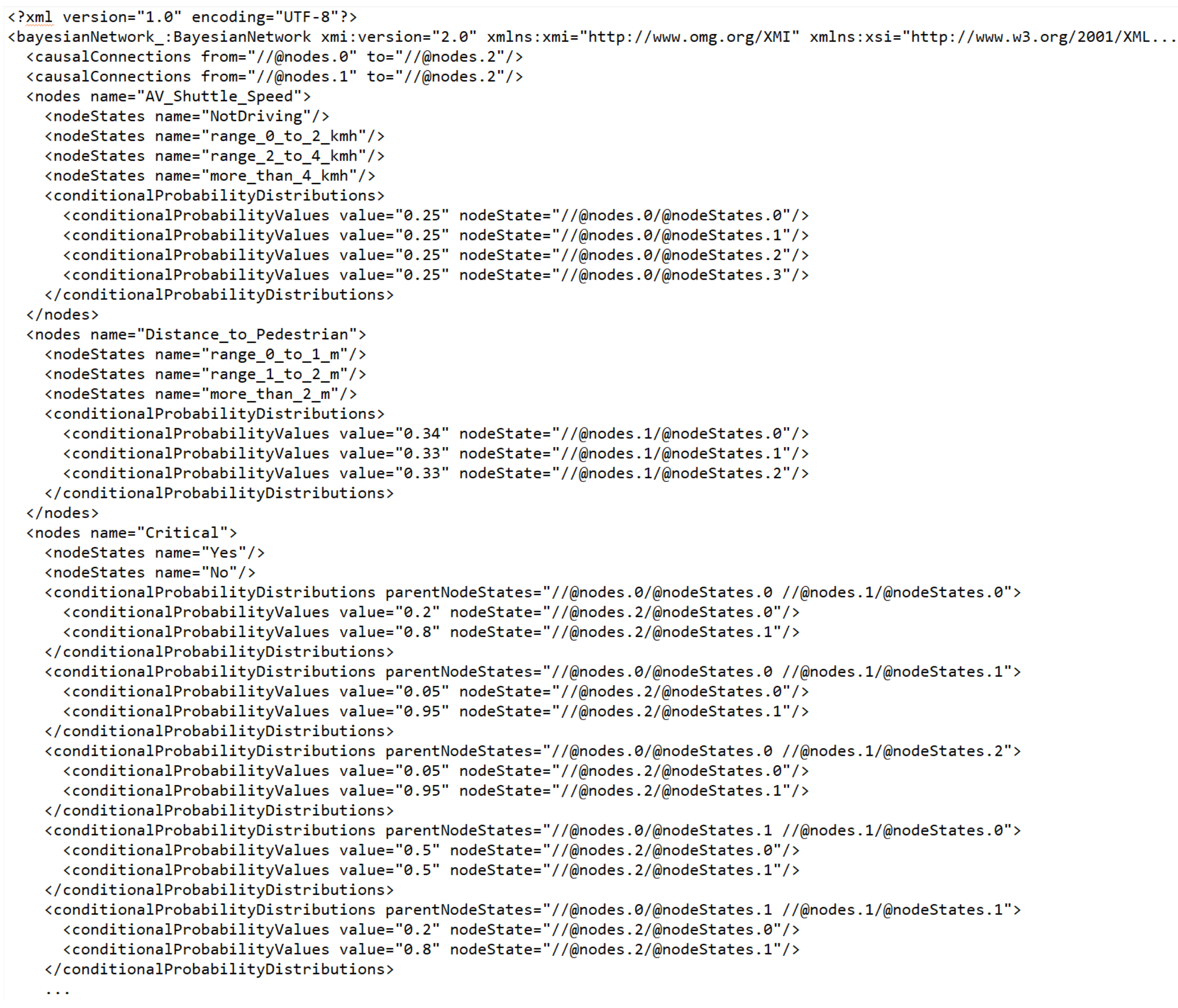
To run the EGL script with the provided JAR file, the following command must be used giving the previously mentioned files as arguments:

```
java -jar egl.jar -e <egl_script> -m <ecore model_file> -x <xml_file>
```

Executing this command will create the .xdsl file for the Bayesian network that is defined in the EDDI model.

To illustrate this transformation with an example, Figure 11 and Figure 12 show excerpts of an actual EDDI model file and the generated .xdsl file. A graphical representation of the corresponding Bayesian network can be found in Section 3.2.2 (see Figure 14) in which this example is picked up again.

[10] https://pgmpy.org/
[11] https://www.eclipse.org/epsilon/doc/egl/
[12] https://www.eclipse.org/modeling/emf/
[13] https://www.eclipse.org/ide/



```
<?xml version="1.0" encoding="UTF-8"?>
<bayesianNetwork_:BayesianNetwork xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XML...
  <causalConnections from="//@nodes.0" to="//@nodes.2"/>
  <causalConnections from="//@nodes.1" to="//@nodes.2"/>
  <nodes name="AV_Shuttle_Speed">
    <nodeStates name="NotDriving"/>
    <nodeStates name="range_0_to_2_kmh"/>
    <nodeStates name="range_2_to_4_kmh"/>
    <nodeStates name="more_than_4_kmh"/>
    <conditionalProbabilityDistributions>
      <conditionalProbabilityValues value="0.25" nodeState="//@nodes.0/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.25" nodeState="//@nodes.0/@nodeStates.1"/>
      <conditionalProbabilityValues value="0.25" nodeState="//@nodes.0/@nodeStates.2"/>
      <conditionalProbabilityValues value="0.25" nodeState="//@nodes.0/@nodeStates.3"/>
    </conditionalProbabilityDistributions>
  </nodes>
  <nodes name="Distance_to_Pedestrian">
    <nodeStates name="range_0_to_1_m"/>
    <nodeStates name="range_1_to_2_m"/>
    <nodeStates name="more_than_2_m"/>
    <conditionalProbabilityDistributions>
      <conditionalProbabilityValues value="0.34" nodeState="//@nodes.1/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.33" nodeState="//@nodes.1/@nodeStates.1"/>
      <conditionalProbabilityValues value="0.33" nodeState="//@nodes.1/@nodeStates.2"/>
    </conditionalProbabilityDistributions>
  </nodes>
  <nodes name="Critical">
    <nodeStates name="Yes"/>
    <nodeStates name="No"/>
    <conditionalProbabilityDistributions parentNodeStates="//@nodes.0/@nodeStates.0 //@nodes.1/@nodeStates.0">
      <conditionalProbabilityValues value="0.2" nodeState="//@nodes.2/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.8" nodeState="//@nodes.2/@nodeStates.1"/>
    </conditionalProbabilityDistributions>
    <conditionalProbabilityDistributions parentNodeStates="//@nodes.0/@nodeStates.0 //@nodes.1/@nodeStates.1">
      <conditionalProbabilityValues value="0.05" nodeState="//@nodes.2/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.95" nodeState="//@nodes.2/@nodeStates.1"/>
    </conditionalProbabilityDistributions>
    <conditionalProbabilityDistributions parentNodeStates="//@nodes.0/@nodeStates.0 //@nodes.1/@nodeStates.2">
      <conditionalProbabilityValues value="0.05" nodeState="//@nodes.2/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.95" nodeState="//@nodes.2/@nodeStates.1"/>
    </conditionalProbabilityDistributions>
    <conditionalProbabilityDistributions parentNodeStates="//@nodes.0/@nodeStates.1 //@nodes.1/@nodeStates.0">
      <conditionalProbabilityValues value="0.5" nodeState="//@nodes.2/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.5" nodeState="//@nodes.2/@nodeStates.1"/>
    </conditionalProbabilityDistributions>
    <conditionalProbabilityDistributions parentNodeStates="//@nodes.0/@nodeStates.1 //@nodes.1/@nodeStates.1">
      <conditionalProbabilityValues value="0.2" nodeState="//@nodes.2/@nodeStates.0"/>
      <conditionalProbabilityValues value="0.8" nodeState="//@nodes.2/@nodeStates.1"/>
    </conditionalProbabilityDistributions>
    ...
```

Figure 11 Example content of an EDDI Bayesian network model

```
<?xml version="1.0" encoding="UTF-8"?>
<smile version="1.0" id="Network1" numsamples="10000" discsamples="10000">
  <nodes>
    <cpt id="AV_Shuttle_Speed">
      <state id="NotDriving" />
      <state id="range_0_to_2_kmh" />
      <state id="range_2_to_4_kmh" />
      <state id="more_than_4_kmh" />
      <probabilities>0.25 0.25 0.25 0.25</probabilities>
    </cpt>
    <cpt id="Distance_to_Pedestrian">
      <state id="range_0_to_1_m" />
      <state id="range_1_to_2_m" />
      <state id="more_than_2_m" />
      <probabilities>0.3399999999999999 0.33 0.33</probabilities>
    </cpt>
    <cpt id="Critical">
      <state id="Yes" />
      <state id="No" />
      <parents>AV_Shuttle_Speed Distance_to_Pedestrian</parents>
      <probabilities>0.2 0.8 0.05 0.95 0.025 0.975 0.5 0.5 0.2 0.8 0.05 0.95 ...
    </cpt>
  </nodes>
  <extensions>
    <genie version="1.0" app="GeNIe 3.0.6518.2 1d74be939ce3f00" name="Network1">
      <node id="AV_Shuttle_Speed">
        <name>AV Shuttle Speed</name>
        <interior color="e5f6f7" />
        ...
```

Figure 12 Example content of the .xdsl Bayesian network model generated from the EDDI shown in Figure 11

3.2.2 Python Monitor Generation

The generation of the actual Bayesian network monitor as a Python package comprises internally several sub-steps which are hidden from the user. First, the runtime Bayesian network model (.xdsl) file from the previous step is transformed into a slightly different format (.xmlbif) which enables to use the open-source pgmpy library for runtime inference. Second, based on this .xmlbif file a Bayesian network configuration is generated which enables to manually define the discretization of the input and the mapping to the evidence / states in the network, as well as, to define the output nodes of the monitor. Finally, the inference logic itself is generated wrapping everything together. An overview of those steps is given in Figure 10 (b).

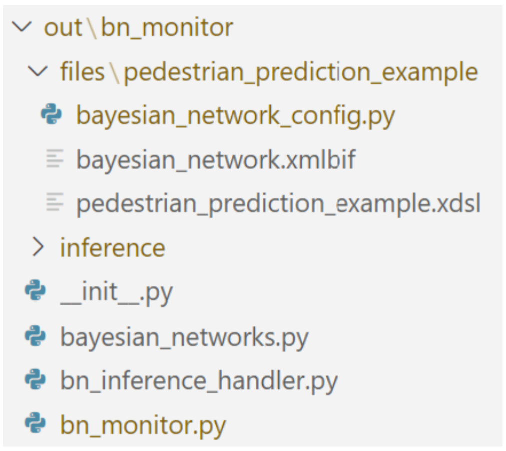
To run the Python-based Bayesian network monitor generation the following steps must be conducted:

1. Place the previously generated runtime Bayesian network model file (.xdsl) in the Bayesian network monitor generator Python package directory.

2. Trigger the generation process:

- `python3 -m BayesianNetwork.bn_monitor_generator -bn <BayesianNetworkName>`
- With "<BayesianNetworkName>" being the name of your .xdsl file.

3. The generated Bayesian network monitor Python package is located in the "./out/bn_monitor/" directory. Figure 13 gives an overview of the generated file structure.



```
out\bn_monitor
  files\pedestrian_prediction_example
    bayesian_network_config.py
    bayesian_network.xmlbif
    pedestrian_prediction_example.xdsl
  inference
  __init__.py
  bayesian_networks.py
  bn_inference_handler.py
  bn_monitor.py
```

Figure 13 File structure of the generated EDDI Python monitor for the example Bayesian network

The Bayesian network configuration Python script is located in the generated Python package in the "./files/<BayesianNetworkName>" sub-directory and is called "bayesian_network_config.py". In there, every node of the Bayesian network has a respective Python method in which the mapping to the evidence can be implemented, as well as, the set-up of the output nodes of the monitor. More on this is given in the following section about manual adjustments (Section 3.2.3). The content of the generated Bayesian network config is illustrated with a simple example Bayesian network that infers the risk of a driving situation for a shuttle and a close pedestrian. The network is shown in Figure 14.

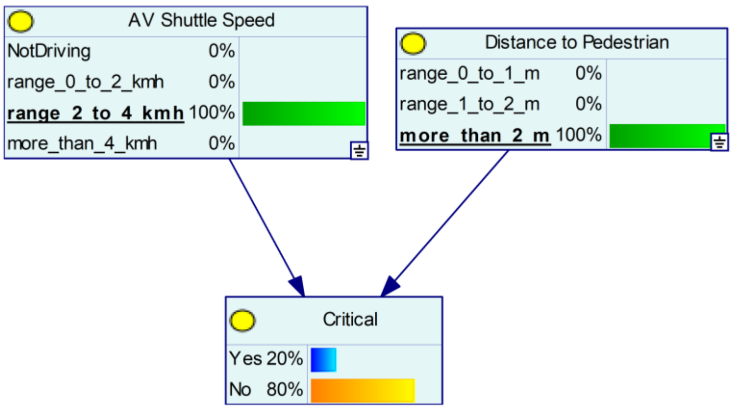
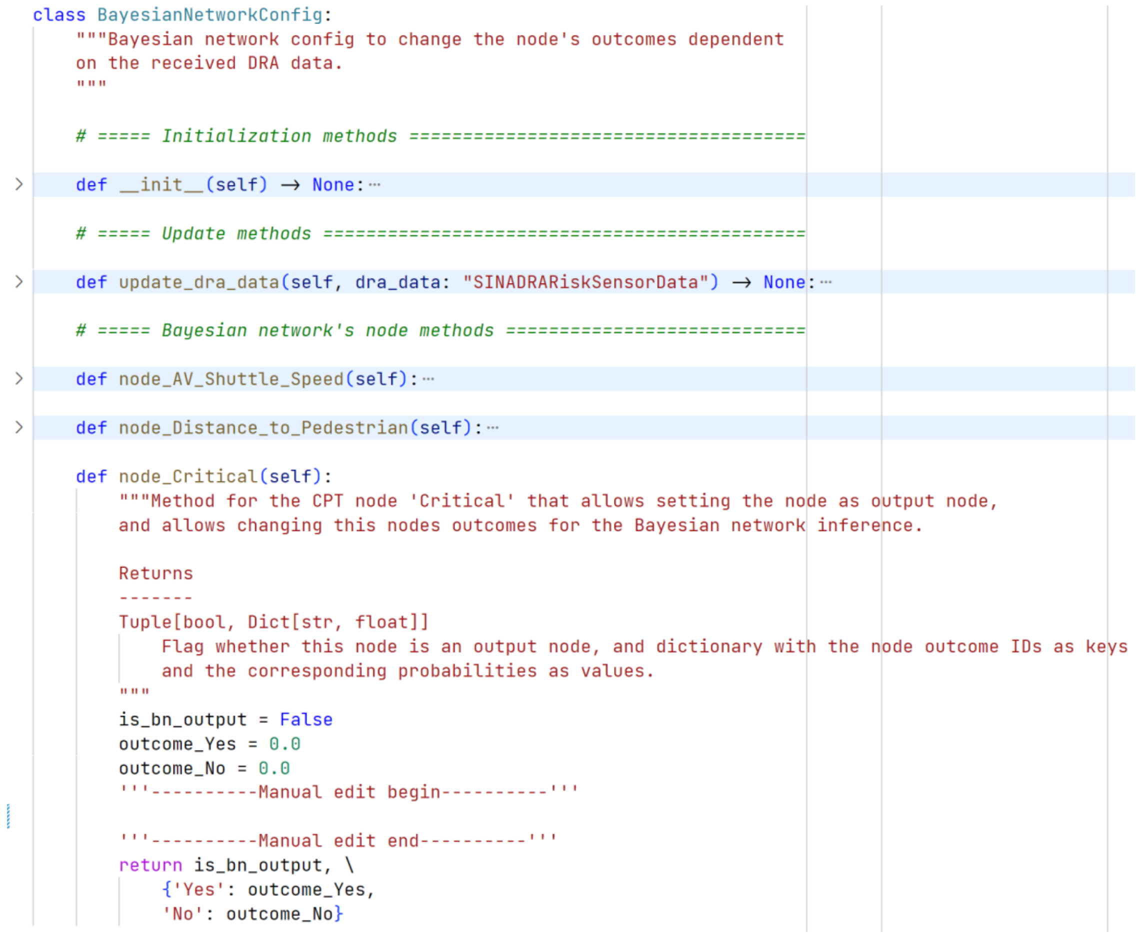


Figure 14 Example Bayesian network to illustrate the Python Bayesian network monitor generation

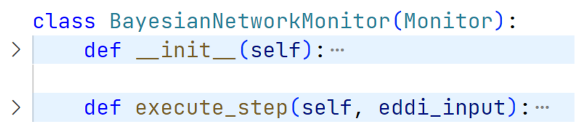
In Figure 15, you can see the actual content of the corresponding configuration Python script. As stated above there is a method for each of the network's nodes (methods with the prefix "node_") including variables for the respective states (= outcomes) and a flag for indicating whether a node is the output of the monitor's inference process. Additionally, there are internal methods for making the data available in the nodes' methods and keeping the data up-to-date. Those methods must not be adapted. The data itself can be accessed by "self.dra_data" inside any of the node methods and is reflecting the same data and data structure as was given to the monitor from outside as an input argument in the "execute_step()" method.



```
class BayesianNetworkConfig:
    """Bayesian network config to change the node's outcomes dependent
    on the received DRA data.
    """

    # ===== Initialization methods ====================================

    def __init__(self) → None:⋯

    # ===== Update methods ============================================

    def update_dra_data(self, dra_data: "SINADRARiskSensorData") → None:⋯

    # ===== Bayesian network's node methods ===========================

    def node_AV_Shuttle_Speed(self):⋯

    def node_Distance_to_Pedestrian(self):⋯

    def node_Critical(self):
        """Method for the CPT node 'Critical' that allows setting the node as output node,
        and allows changing this nodes outcomes for the Bayesian network inference.

        Returns
        -------
        Tuple[bool, Dict[str, float]]
            Flag whether this node is an output node, and dictionary with the node outcome IDs as keys
            and the corresponding probabilities as values.
        """
        is_bn_output = False
        outcome_Yes = 0.0
        outcome_No = 0.0
        '''----------Manual edit begin----------'''

        '''----------Manual edit end----------'''
        return is_bn_output, \
            {'Yes': outcome_Yes,
            'No': outcome_No}
```

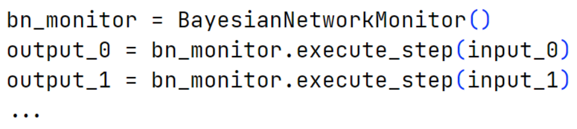
Figure 15 Bayesian network configuration Python script for the example Bayesian network.

The generated monitor can be used and integrated by the "BayesianNetworkMonitor" class which acts as an interface for the monitor (see Figure 16). It is located in the "bn_monitor.py" script. For using the monitor, the class itself must be instantiated which triggers some internal set-up procedure loading the Bayesian network related files. After that, the inference process can be triggered using the "execute_step()" method providing the latest data input as an argument, forwarding the data internally, and triggering the inference process. The input data can be an arbitrary data class. Since the data will be forwarded internally, the same data structure is available inside the Bayesian network configuration Python script that requires manual adjustments (see Section 3.2.3). An example of multiple inferences is shown in Figure 17. The return values are the inferred nodes that are marked in the respective Bayesian network config as an output node. The output data structure is given in Figure 18.



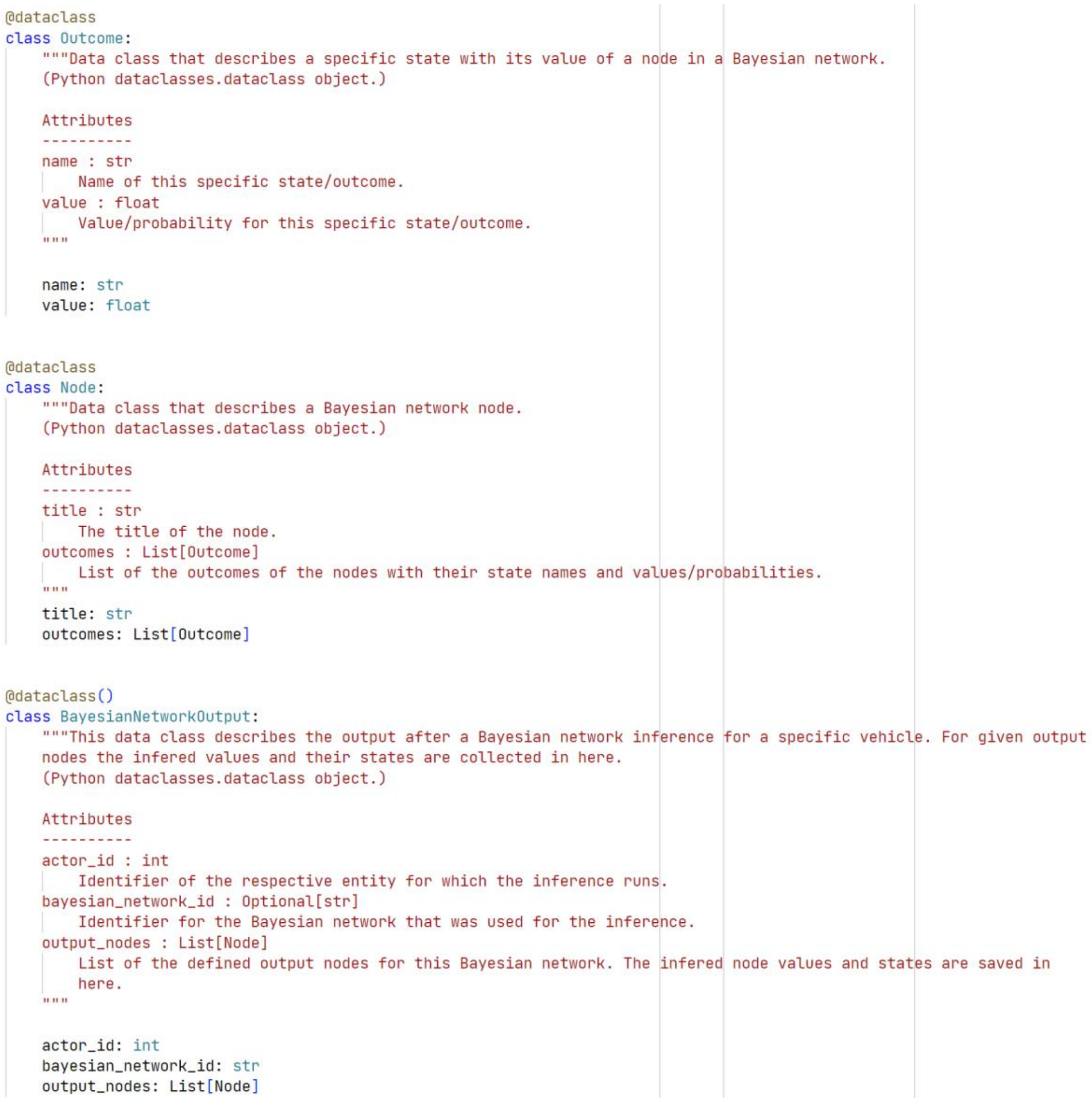
```
class BayesianNetworkMonitor(Monitor):
    def __init__(self):…

    def execute_step(self, eddi_input):…
```

Figure 16 Generated class handling the external interface of the generated Python Bayesian network EDDI monitor



```
bn_monitor = BayesianNetworkMonitor()
output_0 = bn_monitor.execute_step(input_0)
output_1 = bn_monitor.execute_step(input_1)
...
```

Figure 17 Interface of the generated Python Bayesian network EDDI monitor



```
@dataclass
class Outcome:
    """Data class that describes a specific state with its value of a node in a Bayesian network.
    (Python dataclasses.dataclass object.)

    Attributes
    ----------
    name : str
        Name of this specific state/outcome.
    value : float
        Value/probability for this specific state/outcome.
    """

    name: str
    value: float


@dataclass
class Node:
    """Data class that describes a Bayesian network node.
    (Python dataclasses.dataclass object.)

    Attributes
    ----------
    title : str
        The title of the node.
    outcomes : List[Outcome]
        List of the outcomes of the nodes with their state names and values/probabilities.
    """
    title: str
    outcomes: List[Outcome]


@dataclass()
class BayesianNetworkOutput:
    """This data class describes the output after a Bayesian network inference for a specific vehicle. For given output
    nodes the infered values and their states are collected in here.
    (Python dataclasses.dataclass object.)

    Attributes
    ----------
    actor_id : int
        Identifier of the respective entity for which the inference runs.
    bayesian_network_id : Optional[str]
        Identifier for the Bayesian network that was used for the inference.
    output_nodes : List[Node]
        List of the defined output nodes for this Bayesian network. The infered node values and states are saved in
        here.
    """

    actor_id: int
    bayesian_network_id: str
    output_nodes: List[Node]
```

Figure 18 Output data format of the inferred data when running the Python-based Bayesian network EDDI monitor’s “execute_step” method. The “BayesianNetworkOutput” is the return data type which is then using the “Node”, and respectively the “Outcome”, data types.

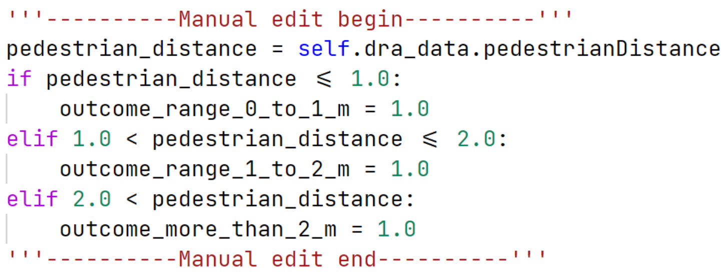
3.2.3 Manual Adjustments

For tailoring the Bayesian network EDDI monitor to the specific Bayesian network some manual adjustments are needed. All the manual changes are located to the Bayesian network configuration Python script that was introduced in Section 3.2.2. In there, the continuous input data must be discretized and mapped to the states of the input data nodes in the Bayesian network. Further, the output nodes of the monitor's inference process are defined in here by setting a specific flag.

The process of adapting the Bayesian network configuration Python script is illustrated using the example Bayesian network and configuration from the previous section (Figure 14, and Figure 15). The example implementation for the discretization of the continuous input is shown for the distance to the pedestrian in Figure 19. Analogously, the required change for selecting a node (in this case the "Critical" node indicating the risk) as an output is shown in Figure 20.

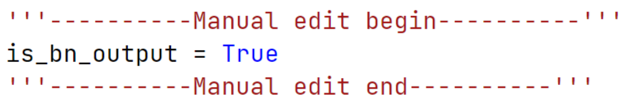
In the config itself, the areas for manual adjustments are explicitly marked in each node's method by comments to not accidentally mess up the internal logic and to support the manual adjustment process.

Notice: For the discretization of an input and the mapping to a node's states it is important that the state probabilities are set properly. To be specific, one of the states must be set to 1.0 and the other states must not be set for a given input. This is a minor limitation of the open-source inference library pgmpy which does not support virtual evidence so far.



```
'''----------Manual edit begin----------'''
pedestrian_distance = self.dra_data.pedestrianDistance
if pedestrian_distance ⩽ 1.0:
    outcome_range_0_to_1_m = 1.0
elif 1.0 < pedestrian_distance ⩽ 2.0:
    outcome_range_1_to_2_m = 1.0
elif 2.0 < pedestrian_distance:
    outcome_more_than_2_m = 1.0
'''----------Manual edit end----------'''
```

Figure 19 Required manual adjustment in the method "node_Distance_to_Pedestrian()" for discretizing a continuous input (here the "pedestrianDistance") and mapping it to the states of the node (concerning the "Distance to Pedestrian" node of the example Bayesian network)



```
'''----------Manual edit begin----------'''
is_bn_output = True
'''----------Manual edit end----------'''
```

Figure 20 Required manual adjustment in the method "node_Critical()" for selecting a node as the output of the monitor (concerning the "Critical" node of the example Bayesian network)

Those manual adjustments may be (partially) automated in a future revision of the Bayesian network EDDI monitor generator to further simplify the process of monitor generation for the user and to facilitate all available data in the EDDIs.

After having done the manual adaptions, the EDDI Bayesian network Python monitor can be executed and the inference process can be triggered. In Figure 21 there is an example of an instantiation of the generated monitor for the used Bayesian network example. Also, the execution of two inferences with example inputs is triggered. Figure 22 shows the respective output that is generated by this example.

```
bn_monitor = BayesianNetworkMonitor()

class InputData:  # Example input dataclass
    def __init__(self, speed, distance):
        self.vehicleSpeed = speed
        self.pedestrianDistance = distance

input0 = InputData(4 / 3.6, 0.4)
output0 = bn_monitor.execute_step(input0)
print(output0)

input1 = InputData(4 / 3.6, 5.3)
output1 = bn_monitor.execute_step(input1)
print(output1)
```

Figure 21 Example instantiation and execution of the generated EDDI Python monitor for the example Bayesian network

```
[BayesianNetworkOutput(actor_id=0, bayesian_network_id='pedestrian_prediction_example',
output_nodes=[Node(title='Critical', outcomes=[Outcome(name='Yes', value=0.7), Outcome(name='No', value=0.3)])])]

[BayesianNetworkOutput(actor_id=0, bayesian_network_id='pedestrian_prediction_example',
output_nodes=[Node(title='Critical', outcomes=[Outcome(name='Yes', value=0.2), Outcome(name='No', value=0.8)])])]
```

Figure 22 Example output for the execution of the generated EDDI Python monitor (see Figure 21)

3.2.4 Python Environment Setup

For setting up the Python environment for the Bayesian network EDDI monitor generation and for the runtime Python environment of the monitor it must be assured that the pgmpy Python library is installed. This can be easily achieved with the following command:

```
python3 -m pip install pgmpy
```

3.2.5 ROS EDDI Bayesian Network Monitor Generation

It is possible to not only generate a plain Python EDDI monitor but rather a ROS EDDI monitor that can directly be integrated into an existing ROS environment. For this, the step previously explained in the “Python Monitor Generation” Section 3.2.2 will be replaced by the procedure explained below in Section 3.6, respectively Section 3.6.1. There details on the ROS EDDI monitor generation are given. The ROS monitor generator uses a specific YAML configuration which must be tailored to the ROS environment in which the monitor shall be used and must be tailored to the respective Bayesian network itself. This YAML configuration is explained in detail in Section 3.6.2 and can be generated as explained in Section 3.6.3. The steps explained in this section stay the same independent of whether the target environment of the generation is a Python package or a ROS package.

3.3 Dynamic Reliability Assessment Component

The dynamic reliability assessment component has been introduced into the project quite recently, as it conceptually integrates well with the ConSerts approach by adding more sophisticated runtime evidences about system and component reliability properties influencing system dependability guarantees. The accompanying technical Python component was evolved and directly applied for drone use cases in the SESAME context (see D7.1 for details) and is at the current stage not fully technically integrated into the Runtime EDDI Tool Generation Framework. In the remainder of the project, it is planned to integrate it technically into the toolchain architecture and add support for semi-automated generation of application-specific ROS monitors that perform dynamic reliability assessment. The current state of the technical implementation of the SafeDrones dynamic reliability assessment component is available at GitHub: https://github.com/koo-ec/SafeDrones

3.4 PERCEPTION UNCERTAINTY MONITOR GENERATOR

The SafeML monitor is applied to monitor scope compliance which supports uncertainty estimation for data-driven perception components. The monitor is designed for runtime application. It performs runtime out-of-distribution detection on a running window of sensor inputs to data-driven models. This way, ODD violations and other anomalies in the execution and the environment can be detected. Statistical distance measures between the empirical distribution of a reference sample and the empirical distribution of the current inputs are computed to detect statistically significant deviations. The reference sample represents an implicit specification of the input from within the scope.

The SafeML monitor receives as input a generic Float32MultiArray message or any other message of which the data can be converted into a numpy floating-point array. The output of the SafeML monitor is a SafeMLOutput message. The SafeMLOutput message contains three fields two of which are Boolean values and one of which is a floating-point number:

- `bool omission`
- `bool odd_violation`
- `float32 significance_level`

The *omission* flag is set in case the running window of inputs is smaller than the desired size. This occurs at the start and if the monitor receives an empty message. The *odd_violation* flag is set if the statistical distance between the reference sample and the current running window is statistically significant. The *significance_level* gives the probability of detecting false positives. A false positive is detected if the agent is operating within the specified ODD but a statistically significant difference between the reference sample and the perceived input is detected. The confidence in the detections of the SafeML monitor is higher the lower the *significance_level* is.

The SafeML monitor builds on the existing SafeML library. The monitor employs the functions for the calculation of the distances between empirical cumulative distribution functions. For the SESAME project this core functionality was extended by implementing statistical distance thresholds for the statistical distance measures such that statistical tests on distances between univariate empirical distributions can be performed. For multivariate data, the monitor applies repeated univariate tests. To consider alpha error accumulation when performing repeated dependent tests, **Bonferroni correction** is applied. For the Bonferroni correction, the significance level is divided by the number of dependent tests to avoid false positive test results. On top of that the monitor applies **Principle Component Analysis** (PCA) to reduce the dimension of high dimensional, multivariate data. This reduces the number of tests needed for high-dimensional data and allows the monitor to be applicable for high-dimensional data without significantly computation overhead.

3.4.1 Environment setup

Computations for the SafeML monitor are performed using the Numpy>=1.16.4 package. To meet the performance demand of computations at runtime, the SafeML distance computations are enabled for GPU support. To avoid compatibility issues both the Tensorflow>=2.4.1 package and the Pytorch>=1.9.1 package are supported. Relevant functions from the SafeML library are attached to the monitor code such that no further dependencies are required. The dependencies are installed using the following command:

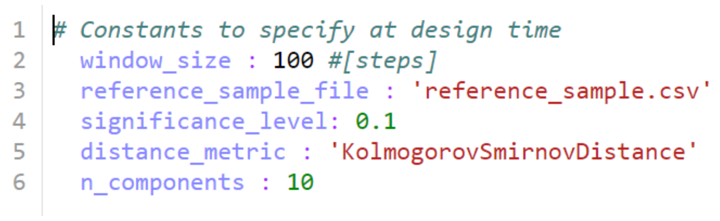
```
python3 -m pip install numpy
```

One of `python3 -m pip install tensorflow==2.4.1` or

```
python3 -m pip install torch==1.9.1
```

3.4.2 SafeML Monitor Configuration

The execution of the SafeML monitor depends on several parameters which are specified in the safeml_config.yml file. The *window_size* determines the amount of current inputs that are compared to the reference sample. It needs to be a positive integer. The larger the *window_size* the more samples are used for the statistical test for homogeneity of the empirical distributions the higher the discriminatory power of the statistical test will become. However, increasing the number of samples also increases the time horizon of the out-of-distribution detection and the computation time. In the example illustrated in Figure 23, the distribution of the last 100 inputs is compared to the distribution of the reference sample.



```
# Constants to specify at design time
  window_size : 100 #[steps]
  reference_sample_file : 'reference_sample.csv'
  significance_level: 0.1
  distance_metric : 'KolmogorovSmirnovDistance'
  n_components : 10
```

Figure 23 Example saveml_config.yml file.

The *reference_sample_file* specifies the path to the *.csv file that contains sufficient samples of inputs during correct system behavior in the specified ODD. The *.csv file should contain a two-dimensional array where each row represents one sample and each column represents one feature of the input to the perception component. This file can be created using the *savetxt* function from the numpy package. To create the reference_sample.csv file the reference data is converted to a two-dimensional numpy array and saved as a csv file.

```
reference_sample = numpy.asarray(reference_data)
```

```
numpy.savetxt(“filename”, reference_sample, delimiter = “,”)
```

The *significance_level* variable specifies the significance level of the statistical test for homogeneity. It is a floating-point number between 0 and 1. It can be interpreted as the likelihood of reporting a false positive result if the distribution of the compared samples is identical.

The *distance_metric* specifies the metric that is used to compute the statistical distance between the empirical cumulative distribution functions of the compared samples. Currently the significance thresholds are implemented for the **‘KolmogorovSmirnovDistance’**, the **‘AndersonDarlingDistance’** and the **‘CramerVanMisesDistance’**. The choice of the distance metric impacts computation time and discriminatory power of the statistical test.

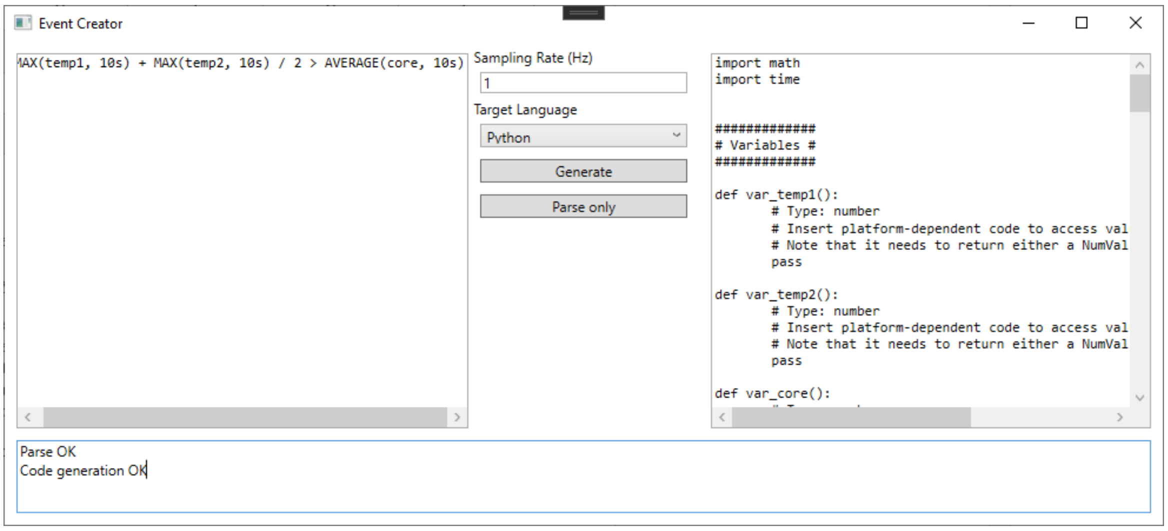
The *n_components* parameter specifies the dimensionality of the input data after the dimension reduction using Principal Component Analysis. A higher number of components results in more statistical tests and longer computation times. It increases the sensitivity of the monitor but it also increases computation time and reduces the robustness to noisy input.

3.5 CONDITIONAL EVENT MONITOR COMPONENT GENERATOR

Having so far described runtime EDDI components, which infer dependability-relevant knowledge about the MRS and its environment, this section supports in defining the atomic pieces of information that are required by the higher level EDDI components – the events. In D4.2, an Extended Backus–Naur Form (EBNF) grammar has been defined to formally express hierarchical event specifications, e.g. conditions in the form of `AVERAGE(temp, 5s) > 100 AND TIME(warning == TRUE, 5s).` However, these conditions are not yet directly executable, let alone in different target programming languages.

To illustrate how events specified in this format can be used to generate executable code suitable for runtime usage, a prototype Event Creator tool has been developed. For now, this is a standalone application for demonstration purposes, although this functionality might eventually be combined into a wider runtime EDDI-generation process. Specifically, the idea is to integrate the expression generation directly into the ConSert, Risk Assessment, Reliability Assessment and SafeML monitor generation, as the realization of input events is at this point still a manual task.

The tool is shown in the Figure 24.



```
MAX(temp1, 10s) + MAX(temp2, 10s) / 2 > AVERAGE(core, 10s)
```

Sampling Rate (Hz): 1

Target Language: Python

```
import math
import time


#############
# Variables #
#############

def var_temp1():
        # Type: number
        # Insert platform-dependent code to access val
        # Note that it needs to return either a NumVal
        pass

def var_temp2():
        # Type: number
        # Insert platform-dependent code to access val
        # Note that it needs to return either a NumVal
        pass

def var_core():
```

```
Parse OK
Code generation OK
```

Figure 24: Event Creator interface

On the left is a text field where event condition expressions can be typed or copied in, according to the grammar presented earlier. In the centre are the parameters: the sampling rate (in Hz) of the monitor, the target language (Python in this case), and two buttons to either generate code or just check the expression syntax.

On the right is the results box, where either the syntax check results or the code appears. The code for the example is too long to include in full, but to provide an overview of the generation algorithm different sections will be described.

The generated code broadly falls into four subsections:

- Variable definitions, which are stub functions where platform-specific code should be inserted to connect the monitor to whichever signals/sensors/messages provide the raw data;
- Function definitions, which define the code and supporting data for any functions used (e.g. circular buffers for timed expressions);
- Classes / data structures for relevant types used (in particular, to support the three-value logic);
- The main monitor itself, responsible for calling the rest of the code, evaluating the expression, and triggering the event as required.

Brief examples of each are discussed below.

```
def var_temp1():
   # Type: number
   # Insert platform-dependent code to access value here
   # Note that it needs to return either a NumVal or a
   # BoolVal based on the type
   pass
```

Here is a brief stub for a function that, when completed with platform-specific code, should interpret any raw data, convert it into the appropriate type/format, and return it for consumption by the rest of the monitor.

```
# Buffer and timer
buffer_MAX_temp1_10s = []
timer_MAX_temp1_10s = time.perf_counter()

# Update function
def update_MAX_temp1_10s():
   if time.perf_counter() - timer_MAX_temp1_10s > 1:
         timer_MAX_temp1_10s = time.perf_counter()
         buffer_MAX_temp1_10s.append(var_temp1());
         if len(buffer_MAX_temp1_10s) > 10:
               buffer_MAX_temp1_10s = buffer_MAX_temp1_10s[1:]

# Actual function
def MAX_temp1_10s():
   if len(buffer_MAX_temp1_10s) > 0:
         return NumVal(max(buffer_MAX_temp1_10s))
   return NumVal(0, True)
```

Here we have the definition of one of the functions used in the example — MAX(temp_1, 10s), meaning "calculate the maximum temperature value from temperature sensor 1 over the past 10 seconds". This entails defining a buffer, setting up a timer and an update function to populate the buffer, and another function that actually evaluates the expression (i.e., calculates the maximum in this case).

There are two things to note here. First is that the buffer update function only executes when enough time has passed, calculated from the time period (10 seconds in this case), the size of the buffer, and the sampling rate (1 Hz). Here, with a sample rate of 1 Hz, we should collect 10 values in 10 seconds; when the buffer exceeds this, we remove the first element, thus creating a circular buffer.

The second note is that the evaluation function uses three-value logic. If the buffer does not contain enough data to perform the calculation, or if it theoretically contained errors, we can return an "UNKNOWN" value (`NumVal(0, True)` means that we don't know what the true value of the number is, so the unknown flag is set to true).

Finally, the main function:

```
def main():
   # Update buffers
   update_MAX_temp1_10s()
   update_MAX_temp2_10s()
   update_AVERAGE_core_10s()

   # Event condition
   if MAX_temp1_10s() + MAX_temp2_10s() / 2 > AVERAGE_core_10s():
         # Insert platform-dependent code to raise this event here
         pass
```

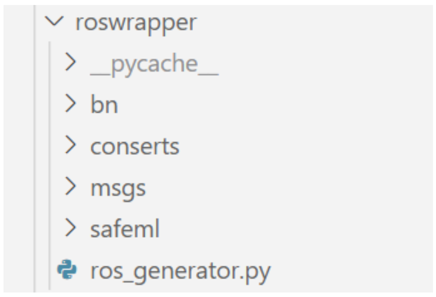
The main function runs at whatever rate the host platform calls it; as shown above, the buffer update functions each have their own timers to ensure that the appropriate readings take place (although if too long passes between each call, there may be unavoidable gaps in the data). It also evaluates the overall condition and has a stub for platform-specific code to be inserted to trigger the event itself, e.g. by emitting a message in a ROS topic or similar.

The intention is to expand the Event Creator to support multiple languages, showing how it can be used to tailor the design-time EDDI information to different scenarios.

3.6 Generic EDDI ROS Wrapper Generator

3.6.1 ROS Wrapper Generation

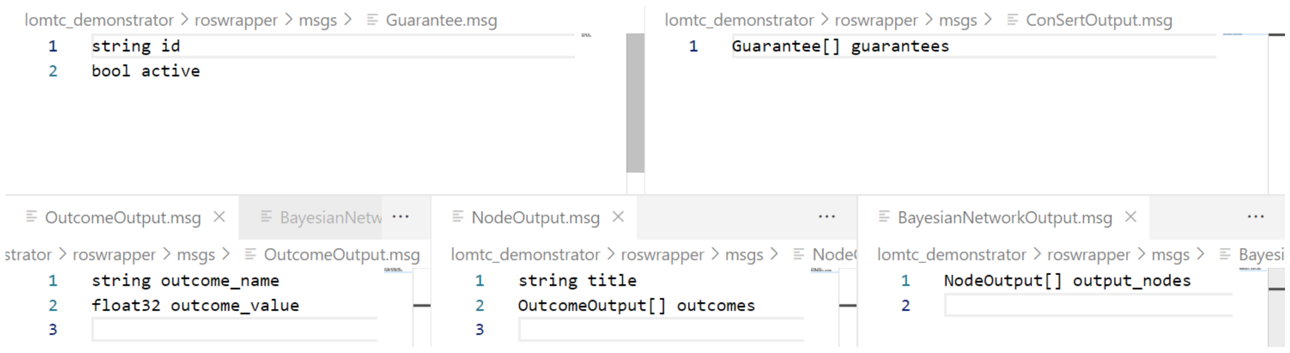
The structure of the ROS wrapper generator consists of the "ros_generator.py" script, a "msgs" folder and a folder for each supported EDDI model, as shown in Figure 25.



```
roswrapper
  __pycache__
  bn
  conserts
  msgs
  safeml
  ros_generator.py
```

Figure 25 ROS Wrapper Generator file structure

Each EDDI model derives from the abstract **RosGenerator** class inside the "ros_generator.py". Further, all custom ROS messages are located as a ".msg" file within the "msgs" folder as shown in Figure 26. These files are copied during the instantiation of the ROS package. Usually those messages correspond to the result of the EDDI monitor, which is published in the ROS network.



lomtc_demonstrator > roswrapper > msgs > Guarantee.msg

```
string id
bool active
```

lomtc_demonstrator > roswrapper > msgs > ConSertOutput.msg

```
Guarantee[] guarantees
```

OutcomeOutput.msg

```
string outcome_name
float32 outcome_value
```

NodeOutput.msg

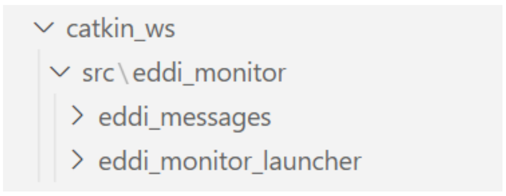
```
string title
OutcomeOutput[] outcomes
```

BayesianNetworkOutput.msg

```
NodeOutput[] output_nodes
```

Figure 26 EDDI ROS Message files

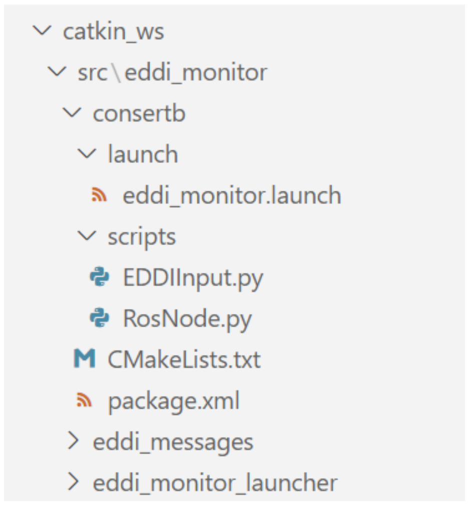
The ROS generator requires the ROS configuration file and the path to the catkin workspace. That means, the catkin_ws must already exist beforehand. Otherwise, a new catkin_ws must be created. The workspace must be in the standard format, where a src folder is located within the catkin workspace root.



```
catkin_ws
  src\eddi_monitor
    > eddi_messages
    > eddi_monitor_launcher
```

Figure 27 Top level ROS package structure without EDDI ROS nodes

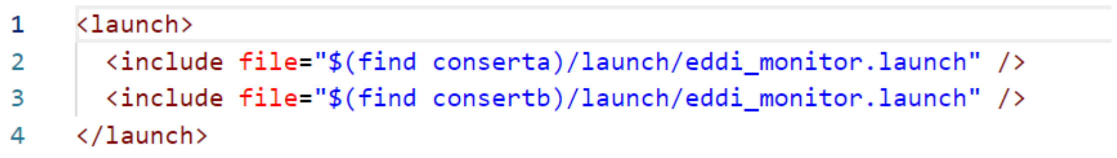
The ROS wrapper generator generates a meta ROS package named "eddi_monitor" within the catkins src folder as shown in Figure 28.



```
catkin_ws
  src\eddi_monitor
    consertb
      launch
        eddi_monitor.launch
      scripts
        EDDIInput.py
        RosNode.py
      CMakeLists.txt
      package.xml
    > eddi_messages
    > eddi_monitor_launcher
```

Figure 28 EDDI ROS node package structure

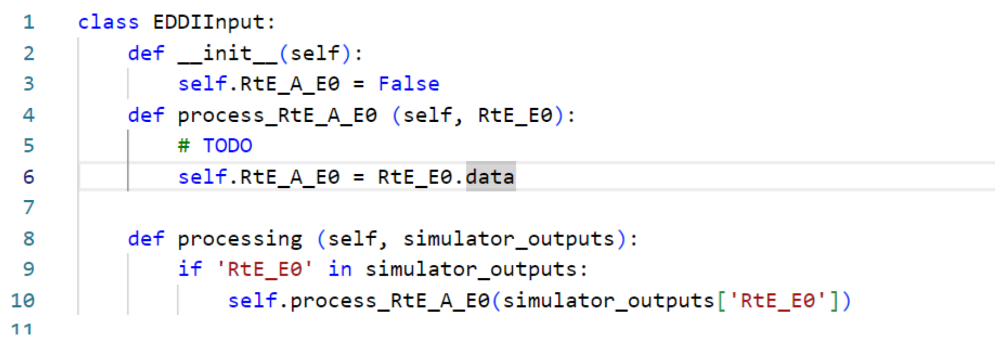
Within this meta package two general ROS packages (eddi_monitor_launcher and eddi_messages) are generated. The "eddi_messages" package contains all custom ROS messages. Several EDDI monitors can exists which all will use the same custom ROS messages. Therefore, one central package was created that contains all required custom messages and no redundant message definition is introduced.



```
<launch>
  <include file="$(find conserta)/launch/eddi_monitor.launch" />
  <include file="$(find consertb)/launch/eddi_monitor.launch" />
</launch>
```

Figure 29 Exemplary launch file of the eddi_monitor_launcher package

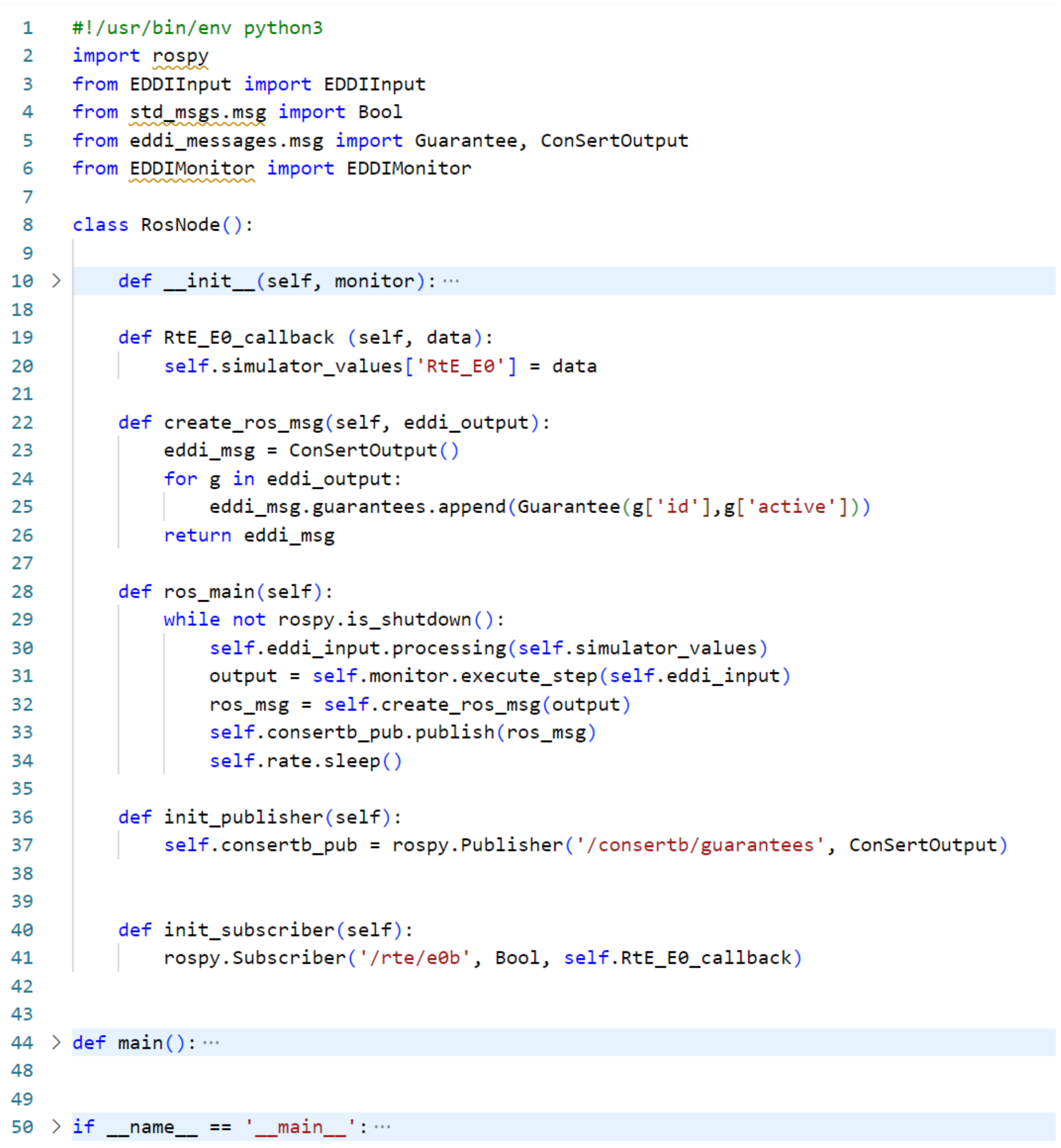
The "eddi_monitor_launcher" package contains only a launch file that starts the launch files of all EDDI ROS nodes within the eddi_monitor package as shown in Figure 29. The aforementioned ROS package structure is generated by the ros_generator script. It requires as arguments the catkin workspace path and the path to the ros generation config file. In case no "eddi-monitor" package exists within the catkin workspace; the script will generate first the ROS meta packages as described before. Afterwards, a ROS node according to the configuration file is generated as a ROS package within the "eddi_monitor" package. Beside the ROS specific package files such as the package.xml, CMakLists.txt, a launch file as well as the EDDIInput.py and the RosNode.py scripts are generated.



```python
class EDDIInput:
    def __init__(self):
        self.RtE_A_E0 = False
    def process_RtE_A_E0 (self, RtE_E0):
        # TODO
        self.RtE_A_E0 = RtE_E0.data

    def processing (self, simulator_outputs):
        if 'RtE_E0' in simulator_outputs:
            self.process_RtE_A_E0(simulator_outputs['RtE_E0'])
```

Figure 30 EDDI Input file for an example configuration file

The EDDIInput.py describes the input parameters of the EDDI monitor. These parameters are defined within the configuration file (see next section). There, the name of the input parameters, the type and unit of measurement (optional) are described. Also, the required ROS messages for deriving the EDDI input are specified in order to cover that further processing of the received ROS messages is necessary. Therefore, a method template is generated for each EDDI input parameter. As shown in Figure 30, the EDDI input parameter RtE_A_E0 is based on the ROS Message RtE_E0. A processing method is then generated (“process_RtE_A_E0”) which receives the required ROS message as an input. Here, the developer must manually add the processing logic. For some messages this only requires simple renaming. However, the EDDI input can also be the result of complicated calculation based on multiple ROS messages. Therefore, this manual coding is required.



```python
#!/usr/bin/env python3
import rospy
from EDDIInput import EDDIInput
from std_msgs.msg import Bool
from eddi_messages.msg import Guarantee, ConSertOutput
from EDDIMonitor import EDDIMonitor

class RosNode():

    def __init__(self, monitor):⋯

    def RtE_E0_callback (self, data):
        self.simulator_values['RtE_E0'] = data

    def create_ros_msg(self, eddi_output):
        eddi_msg = ConSertOutput()
        for g in eddi_output:
            eddi_msg.guarantees.append(Guarantee(g['id'],g['active']))
        return eddi_msg

    def ros_main(self):
        while not rospy.is_shutdown():
            self.eddi_input.processing(self.simulator_values)
            output = self.monitor.execute_step(self.eddi_input)
            ros_msg = self.create_ros_msg(output)
            self.consertb_pub.publish(ros_msg)
            self.rate.sleep()

    def init_publisher(self):
        self.consertb_pub = rospy.Publisher('/consertb/guarantees', ConSertOutput)


    def init_subscriber(self):
        rospy.Subscriber('/rte/e0b', Bool, self.RtE_E0_callback)


def main():⋯


if __name__ == '__main__':⋯
```

Figure 31 Example of an EDDI ROS node

The RosNode.py contains all the logic of the ROS node. When initialized, subscriber and publisher are created respectively for all the specified simulator outputs and EDDI outputs within the configuration file. It further defines the ROS main loop where periodically an EDDI monitor step is executed and the results are then published. Concurrently, the ROS node is listening to the subscribed messages and updates the corresponding dictionary entry. This dictionary is then processed to create the EDDI input structure within the main loop. An EDDI monitor step depends on the model. For example, in the context of ConSerts this corresponds on the evaluation of all guarantees. The result then would be a list of guarantees. This result is then published within the ROS network. For other EDDI monitors corresponding results are created and then published. According to the specified frequency within the ROS configuration, the ROS node will sleep until the next iteration begins.

3.6.2 ROS Configuration File

The ROS configuration file contains all information that is required to generate the EDDI ROS node, which wraps a technique-specific inference monitor (i.e. ConSert, Dynamic Risk Assessment, SafeML) to be integrated into an existing MRS ROS application. It is described as a YAML file as shown in Figure 32.

```yaml
---
Model:
    id: ConsertB
    type: ConSert
    frequency: 10
    parameters:
      - guarantees:
        - id: System_A_G0_c2
SimulatorOutputs:
  - id: RtE_E0
    type: std_msgs.Bool
    topic: /rte/e0b
EDDIInputs:
  - id: RtE_A_E0
    type: bool
    requires: [RtE_E0]
    default: false
EDDIOutputs:
  - id: consertb
    topic: /consertb/guarantees
```

Figure 32 Example of the ROS configuration file

The file is divided into four sections:

Model: In the **Model** section of the YAML file, basic information is provided. The standard fields are **id, type, frequency** and **parameters.** These correspond to the model name, the model type (ConSert, Bayesian Network, SafeML) and the frequency of the ROS main loop. The **parameters** field describes model specific information.

SimulatorOutputs: In this section, values from an existing ROS network are described. These values might be used to create input values for the EDDI monitor. Each of these values have a name (**id**), a data type (**type**) and the topic (**topic**) within the ROS network. The ROS node will subscribe to the topic and propagate the received message for further processing

EDDIInputs: Here, a list of all input parameters of the EDDI monitor are described. Each of them has a name (**id**), a data type (**type**), a default value (**default**) and a field **requires.** The **requires** field describes a list of **id**s which are matched to the **id**s of the **SimulatorOutputs** values. An EDDI input parameter might depend on one or more existing messages.

EDDIOutputs: In this section, the output of the EDDI monitor is specified. The ROS node will publish the message under the specified **topic.** Additionally, a name (**id**) field is set.

3.6.3 Semi-automated EDDI Integration into ROS Applications with Model Based ROS (MROS)

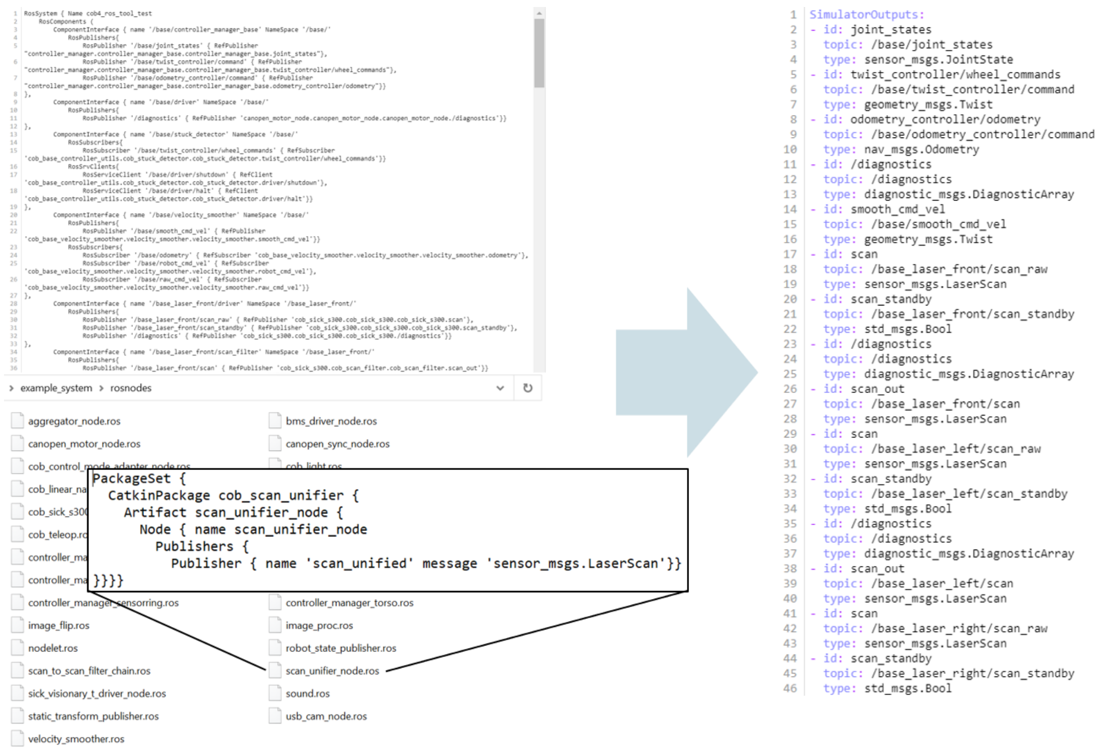
To facilitate the integration of new EDDI components into existing systems of ROS components, parts of the Model Based ROS (MROS) toolchain are applied. The goal is to generate fitting EDDI components rather than requiring to adapt the existing components.

The MROS model extractors can be applied to extract ROS models and ROS system models from an existing codebase using static code analysis. MROS tools for graphical modelling support the integration of new components into the system model. Adapted models are then used to generate the ROS configuration that is required for the generation of the adapted EDDI components. We apply the EDDI system parser to translate the ROS models and ROS system models into the configuration the is required to generate the EDDI components.

Given the ROS models and the ROS system model of a ROS application that is integrated with the models of the EDDI components the integration requires the adaption of the ROS configuration of the EDDI components. The eddi_config.yml is changed using the eddi_system_parser. The EDDI system parser uses the concrete syntax tree of the ROS model and ROS system model DSL to parse the model files and extract the id, topic name and message types of relevant inputs to the EDDI and saves them to the SimulatorOutput part of the eddi_config.yml (see Figure 32 Example of the ROS configuration file). The system parsing is invoked using the following command.

```
$ python eddi_system_parser.py -s <*.rossystem path> -m <*.ros model
directory> [-c <*.yml output path>]
```

The eddi_system_parser receives as options the path of the ROS system file with the -s option, the directory that contains all the ROS model files with the -m option and optionally the path of the resulting configuration file with the -c option. By default, the eddi_system_parser creates or uses the eddi_config.yml file in the current working directory and overwrites the SimulatorOutput section when called.



```
RosSystem { Name cob4_ros_tool_test
    RosComponents (
        ComponentInterface { name '/base/controller_manager_base' NameSpace '/base/'
            RosPublishers{
                RosPublisher '/base/joint_states' { RefPublisher
"controller_manager.controller_manager_base.controller_manager_base.joint_states"},
                RosPublisher '/base/twist_controller/command' { RefPublisher
"controller_manager.controller_manager_base.controller_manager_base.twist_controller/wheel_commands"},
                RosPublisher '/base/odometry_controller/command' { RefPublisher
"controller_manager.controller_manager_base.controller_manager_base.odometry_controller/odometry"}}
},
        ComponentInterface { name '/base/driver' NameSpace '/base/'
            RosPublishers{
                RosPublisher '/diagnostics' { RefPublisher 'canopen_motor_node.canopen_motor_node.canopen_motor_node./diagnostics'}}
},
        ComponentInterface { name '/base/stuck_detector' NameSpace '/base/'
            RosSubscribers{
                RosSubscriber '/base/twist_controller/wheel_commands' { RefSubscriber
'cob_base_controller_utils.cob_stuck_detector.cob_stuck_detector.twist_controller/wheel_commands'}}
            RosSrvClients{
                RosServiceClient '/base/driver/shutdown' { RefClient
'cob_base_controller_utils.cob_stuck_detector.cob_stuck_detector.driver/shutdown'},
                RosServiceClient '/base/driver/halt' { RefClient
'cob_base_controller_utils.cob_stuck_detector.cob_stuck_detector.driver/halt'}}
},
        ComponentInterface { name '/base/velocity_smoother' NameSpace '/base/'
            RosPublishers{
                RosPublisher '/base/smooth_cmd_vel' { RefPublisher
'cob_base_velocity_smoother.velocity_smoother.velocity_smoother.smooth_cmd_vel'}}
            RosSubscribers{
                RosSubscriber '/base/odometry' { RefSubscriber 'cob_base_velocity_smoother.velocity_smoother.velocity_smoother.odometry'},
                RosSubscriber '/base/robot_cmd_vel' { RefSubscriber
'cob_base_velocity_smoother.velocity_smoother.velocity_smoother.robot_cmd_vel'},
                RosSubscriber '/base/raw_cmd_vel' { RefSubscriber
'cob_base_velocity_smoother.velocity_smoother.velocity_smoother.raw_cmd_vel'}}
},
        ComponentInterface { name '/base_laser_front/driver' NameSpace '/base_laser_front/'
            RosPublishers{
                RosPublisher '/base_laser_front/scan_raw' { RefPublisher 'cob_sick_s300.cob_sick_s300.cob_sick_s300.scan'},
                RosPublisher '/base_laser_front/scan_standby' { RefPublisher 'cob_sick_s300.cob_sick_s300.cob_sick_s300.scan_standby'},
                RosPublisher '/diagnostics' { RefPublisher 'cob_sick_s300.cob_sick_s300.cob_sick_s300./diagnostics'}}
},
        ComponentInterface { name '/base_laser_front/scan_filter' NameSpace '/base_laser_front/'
            RosPublishers{
                RosPublisher '/base_laser_front/scan' { RefPublisher 'cob_sick_s300.cob_scan_filter.cob_scan_filter.scan_out'}}
```

```
PackageSet {
  CatkinPackage scan_unifier {
    Artifact scan_unifier_node {
      Node { name scan_unifier_node
        Publishers {
          Publisher { name 'scan_unified' message 'sensor_msgs.LaserScan'}}
}}}}
```

```yaml
SimulatorOutputs:
- id: joint_states
  topic: /base/joint_states
  type: sensor_msgs.JointState
- id: twist_controller/wheel_commands
  topic: /base/twist_controller/command
  type: geometry_msgs.Twist
- id: odometry_controller/odometry
  topic: /base/odometry_controller/command
  type: nav_msgs.Odometry
- id: /diagnostics
  topic: /diagnostics
  type: diagnostic_msgs.DiagnosticArray
- id: smooth_cmd_vel
  topic: /base/smooth_cmd_vel
  type: geometry_msgs.Twist
- id: scan
  topic: /base_laser_front/scan_raw
  type: sensor_msgs.LaserScan
- id: scan_standby
  topic: /base_laser_front/scan_standby
  type: std_msgs.Bool
- id: /diagnostics
  topic: /diagnostics
  type: diagnostic_msgs.DiagnosticArray
- id: scan_out
  topic: /base_laser_front/scan
  type: sensor_msgs.LaserScan
- id: scan
  topic: /base_laser_left/scan_raw
  type: sensor_msgs.LaserScan
- id: scan_standby
  topic: /base_laser_left/scan_standby
  type: std_msgs.Bool
- id: /diagnostics
  topic: /diagnostics
  type: diagnostic_msgs.DiagnosticArray
- id: scan_out
  topic: /base_laser_left/scan
  type: sensor_msgs.LaserScan
- id: scan
  topic: /base_laser_right/scan_raw
  type: sensor_msgs.LaserScan
- id: scan_standby
  topic: /base_laser_right/scan_standby
  type: std_msgs.Bool
```

Figure 33 Inputs to a ROS configuration file are extracted from a ROS system model file and corresponding ROS model files using the EDDI system parsing tool.

While the generation of the EDDI configuration is not fully automated yet the EDDI system parser supports the generation of EDDI configurations. Figure 33 shows a rather complex *.rossystem file and a folder containing the corresponding *.ros model files. Using the EDDI system parser the information is translated from the domain specific language into the a human readable YAML file format and reduced to a manageable level while retaining all the information that is required for the generation of ROS components that are tailored to the existing application.

4. OVERALL TOOLCHAIN EXAMPLE

In this section the overall toolchain is explain on an example to follow the previous parts step-by-step. Further, in next section the generator command line tool is presented which combines the EDDI Monitor generation with the ROS wrapper generation to simplify usability. Figure 34 gives an overview over the toolchain. Each steps of the toolchain is connected to the corresponding section within this report.

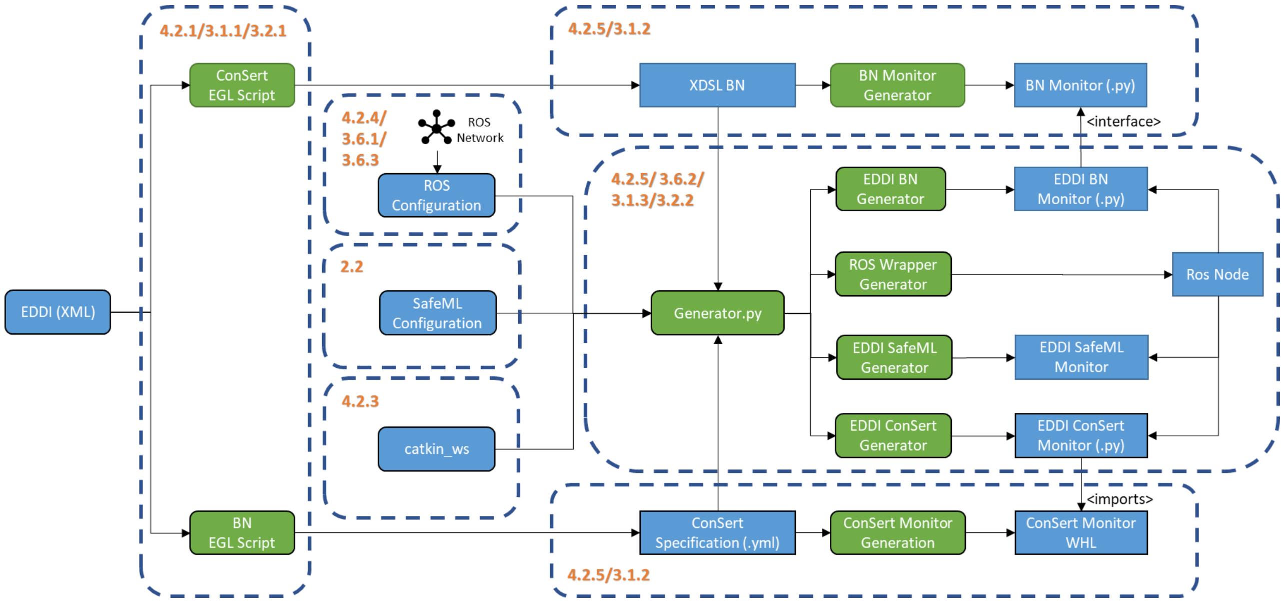


Figure 34 Overall Toolchain Overview

4.1 GENERATOR COMMAND LINE TOOL

An executable ROS Node can be generated with the generator.py command line tool. As shown in Figure 35, the generator.py script takes an existing catkin workspace directory, an existing EDDI model file and the ROS configuration file. The script then generates the ROS Wrapper and the EDDI Monitor.

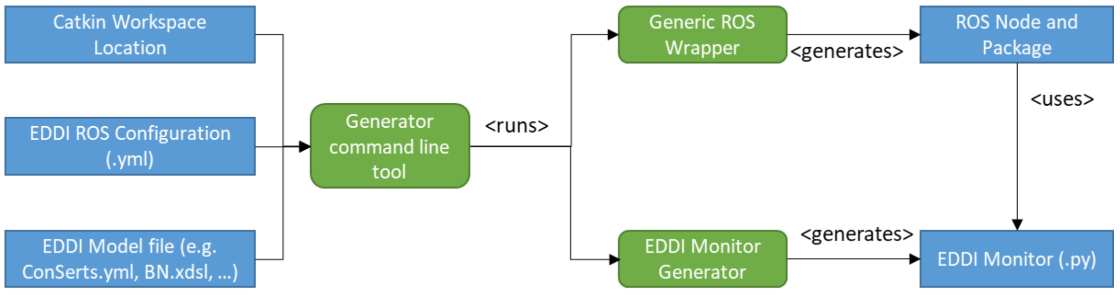

Catkin Workspace Location
EDDI ROS Configuration (.yml)
EDDI Model file (e.g. ConSerts.yml, BN.xdsl, ...)
Generator command line tool
<runs>
Generic ROS Wrapper
<generates>
ROS Node and Package
<uses>
EDDI Monitor Generator
<generates>
EDDI Monitor (.py)

Figure 35 Overview of the generator.py command line tool

The generator.py script can be started via command line with the following syntax:

```
generator.py -w <catkin_ws> -c <config.yml> -m <model>
```

catkin_ws: Path to the catkin workspace

config.yml: Path to the EDDI ROS configuration file

Model: Path to the model file

After execution, the ROS package is created within the catkin workspace and the EDDI ROS node can be launched with:

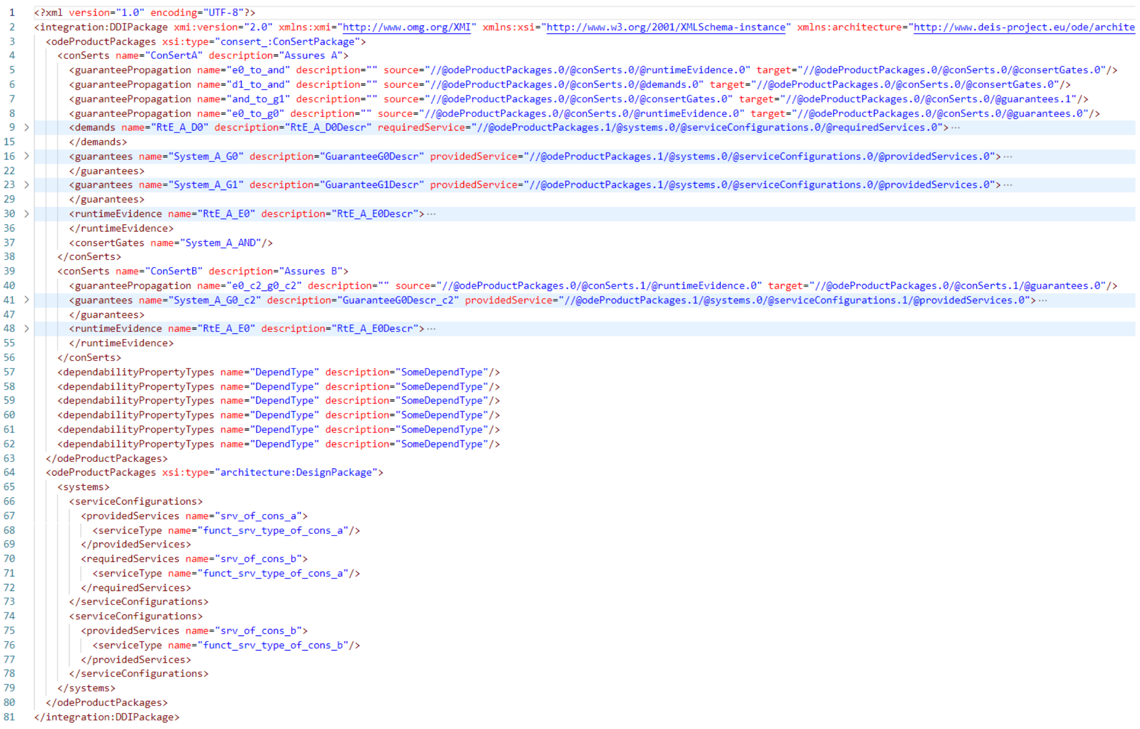
```
roslaunch eddi.launch
```

4.2 START-TO-END TOOLCHAIN EXAMPLE

In the following a start-to-end example of the overall toolchain is explained step by step. It is assumed, that an EDDI exists which specifies at least one runtime model of type Bayesian Network, ConSerts or SafeML. Further, an environment is required with an installed ROS distribution and Python (version 3 and newer) with pip. Depending on the model type additional dependencies might be required (details in their corresponding sections).

4.2.1 Generation of the Model File

Given an EDDI as an XML file. The first step requires to generate the corresponding model file.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<integration:DDIPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:architecture="http://www.deis-project.eu/ode/archite
  <odeProductPackages xsi:type="consert_:ConSertPackage">
    <conSerts name="ConSertA" description="Assures A">
      <guaranteePropagation name="e0_to_and" description="" source="//@odeProductPackages.0/@conSerts.0/@runtimeEvidence.0" target="//@odeProductPackages.0/@conSerts.0/@consertGates.0"/>
      <guaranteePropagation name="d1_to_and" description="" source="//@odeProductPackages.0/@conSerts.0/@demands.0" target="//@odeProductPackages.0/@conSerts.0/@consertGates.0"/>
      <guaranteePropagation name="and_to_g1" description="" source="//@odeProductPackages.0/@conSerts.0/@consertGates.0" target="//@odeProductPackages.0/@conSerts.0/@guarantees.1"/>
      <guaranteePropagation name="e0_to_g0" description="" source="//@odeProductPackages.0/@conSerts.0/@runtimeEvidence.0" target="//@odeProductPackages.0/@conSerts.0/@guarantees.0"/>
      <demands name="RtE_A_D0" description="RtE_A_D0Descr" requiredService="//@odeProductPackages.1/@systems.0/@serviceConfigurations.0/@requiredServices.0">···
      </demands>
      <guarantees name="System_A_G0" description="GuaranteeG0Descr" providedService="//@odeProductPackages.1/@systems.0/@serviceConfigurations.0/@providedServices.0">···
      </guarantees>
      <guarantees name="System_A_G1" description="GuaranteeG1Descr" providedService="//@odeProductPackages.1/@systems.0/@serviceConfigurations.0/@providedServices.0">···
      </guarantees>
      <runtimeEvidence name="RtE_A_E0" description="RtE_A_E0Descr">···
      </runtimeEvidence>
      <consertGates name="System_A_AND"/>
    </conSerts>
    <conSerts name="ConSertB" description="Assures B">
      <guaranteePropagation name="e0_c2_g0_c2" description="" source="//@odeProductPackages.0/@conSerts.1/@runtimeEvidence.0" target="//@odeProductPackages.0/@conSerts.1/@guarantees.0"/>
      <guarantees name="System_A_G0_c2" description="GuaranteeG0Descr_c2" providedService="//@odeProductPackages.1/@systems.0/@serviceConfigurations.1/@providedServices.0">···
      </guarantees>
      <runtimeEvidence name="RtE_A_E0" description="RtE_A_E0Descr">···
      </runtimeEvidence>
    </conSerts>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
    <dependabilityPropertyTypes name="DependType" description="SomeDependType"/>
  </odeProductPackages>
  <odeProductPackages xsi:type="architecture:DesignPackage">
    <systems>
      <serviceConfigurations>
        <providedServices name="srv_of_cons_a">
          <serviceType name="funct_srv_type_of_cons_a"/>
        </providedServices>
        <requiredServices name="srv_of_cons_b">
          <serviceType name="funct_srv_type_of_cons_a"/>
        </requiredServices>
      </serviceConfigurations>
      <serviceConfigurations>
        <providedServices name="srv_of_cons_b">
          <serviceType name="funct_srv_type_of_cons_b"/>
        </providedServices>
      </serviceConfigurations>
    </systems>
  </odeProductPackages>
</integration:DDIPackage>
```

Figure 36 EDDI XML example file describing two ConSerts

In Figure 36 and EDDI as an XML File is shown. Within this EDDI two ConSerts are specified. This EDDI could further describe more ConSerts but also other models like BNs. In this example only ConSerts are defined. Therefore, the steps from section 3.1 are followed. First, the egl.jar file is used to generate the ConSerts as YAML files. Therefore, the command

```
java -jar egl.jar -e <egl_script> -m <ecore model_file> -x <xml_file>
```

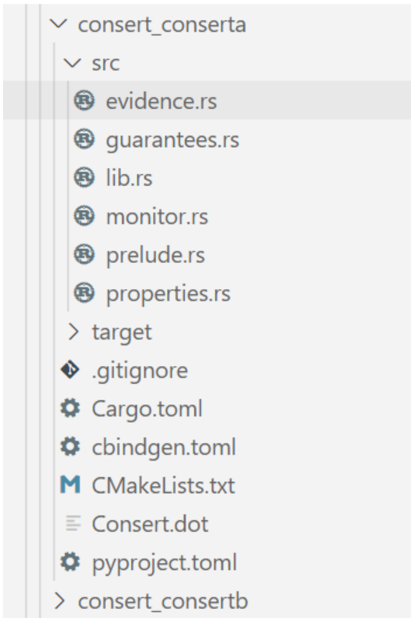
is executed. Thereby, the model_to_yml.egl, the generatedMergedODE.ecore and the above shown example.xml file paths are used respectively. In this case two files are created. The ConSertA.yml and the ConSertB.yml, each specifying the corresponding ConSert described in the EDDI file. Those were already shown in Figure 6. In case, other model types are described in the EDDI, the above command must be executed with the path to the corresponding egl script. The ecore model file path and the xml file remain the same.

4.2.2 Generate the Monitor

Now, the monitor is generated. This step differs based on the model type. See for more details the corresponding section. Here, for ConSerts, the consert-rs tool is executed. Therefore,

```
consert s.exe compile --py -i <consert_name.yml>
```

is executed for ConSertA.yml and ConSertB.yml. The result will be two folders within a target folder containing the related rust package. In Figure 37 the folder structure of the ConSertA is shown. The structure for ConSertB is analogous.



```
consert_conserta
  src
    evidence.rs
    guarantees.rs
    lib.rs
    monitor.rs
    prelude.rs
    properties.rs
  target
  .gitignore
  Cargo.toml
  cbindgen.toml
  CMakeLists.txt
  Consert.dot
  pyproject.toml
consert_consertb
```

Figure 37 Generated ConSerts as Rust file with python bindings

The next step requires the python library maturin to generate python Wheel files from this annotated Rust code. Therefore, maturin must be installed. Otherwise, it can be installed with

```
pip install maturin
```

Afterward, the Wheel files are built by calling

```
maturin build
```

within the consert_conserta and consert_consertb directory. Now, the Wheel files for each ConSert are generated. In the next step the ConSerts are installed from their Wheel files with

```
pip install consert_<consert_name>.whl
```

for both, ConSertA and ConSertB. These steps are exclusively for ConSert models. For BNs and other models the steps described in the related sections of chapter 3 must be followed.

4.2.3 Create a Catkin Workspace

In this example, the EDDI Monitor communicate with each other and with the simulator and other systems over the ROS Network. For the next steps, a catkin workspace is required. In case no such workspace already exists, it must be created. First, create the path

```
catkin_ws/src
```

Then initialize the workspace by calling

```
catkin init
```

from the workspace root directory. Then build the workspace with

```
catkin build
```

Now, source the setup file with

```
source devel/setup.bash
```

The last command must be executed whenever a new terminal is launched.

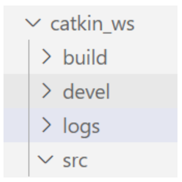
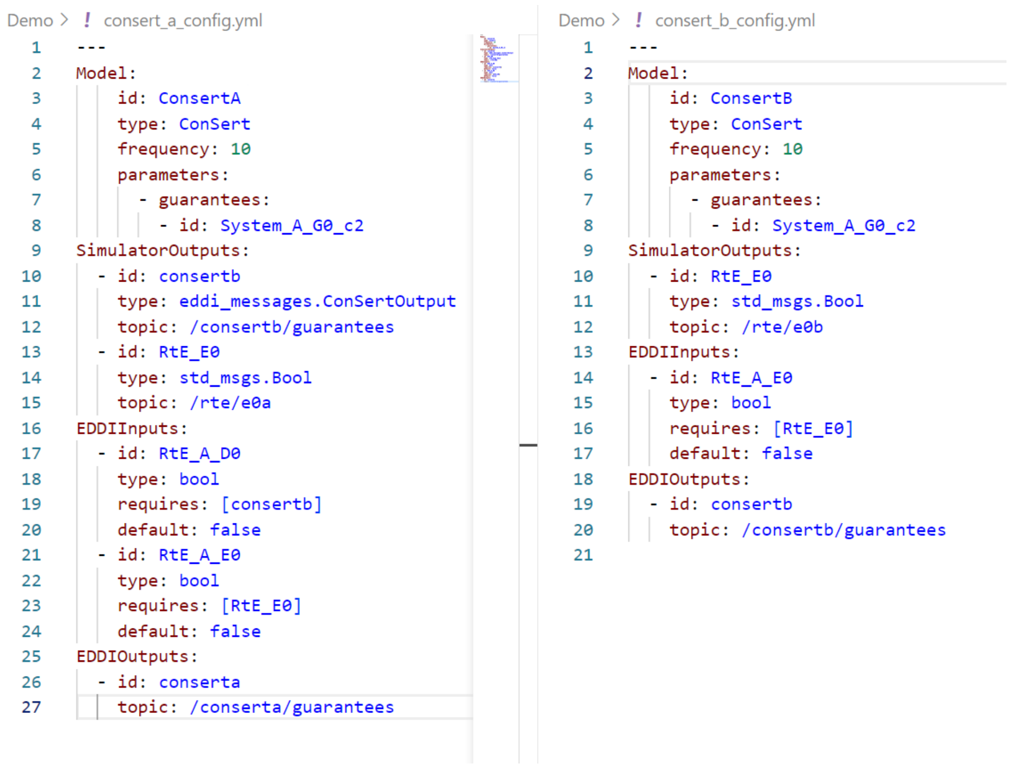


Figure 38 Catkin workspace directory

The resulting directory will then look similar to Figure 38.

4.2.4 Create the ROS Configuration File

In this tool chain, the generated EDDI monitors are executed within an existing ROS network. Therefore, some configuration is necessary. The file is described in section 3.6 in more detail. For each model such a configuration file is needed. In this example, a configuration file for ConSertA and one for ConSertB are required.



```yaml
---
Model:
    id: ConsertA
    type: ConSert
    frequency: 10
    parameters:
      - guarantees:
        - id: System_A_G0_c2
SimulatorOutputs:
  - id: consertb
    type: eddi_messages.ConSertOutput
    topic: /consertb/guarantees
  - id: RtE_E0
    type: std_msgs.Bool
    topic: /rte/e0a
EDDIInputs:
  - id: RtE_A_D0
    type: bool
    requires: [consertb]
    default: false
  - id: RtE_A_E0
    type: bool
    requires: [RtE_E0]
    default: false
EDDIOutputs:
  - id: conserta
    topic: /conserta/guarantees
```

```yaml
---
Model:
    id: ConsertB
    type: ConSert
    frequency: 10
    parameters:
      - guarantees:
        - id: System_A_G0_c2
SimulatorOutputs:
  - id: RtE_E0
    type: std_msgs.Bool
    topic: /rte/e0b
EDDIInputs:
  - id: RtE_A_E0
    type: bool
    requires: [RtE_E0]
    default: false
EDDIOutputs:
  - id: consertb
    topic: /consertb/guarantees

```

Figure 39 ROS Configuration File for ConSertA and ConSertB

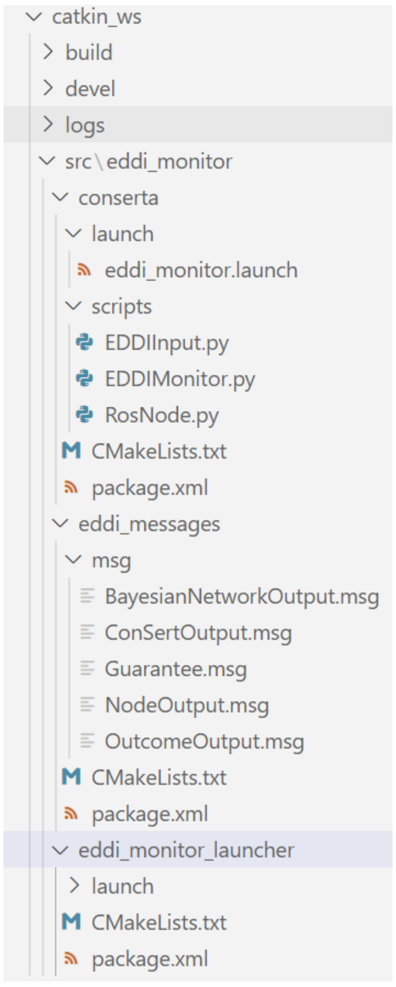
In this example, it is assumed, that ConSertB's guarantee "System_A_G0_c2" fulfills the demand "RtE_A_D0" of ConSertA. ConSertB will publish its guarantee within the ROS network. This message must be known by ConSertA. That is why it is also specified within the SimulatorOutputs section within the ConSertAs' config file. The link between the demand and the guarantee is then described within the EDDIInput section of ConSertA's configuration file. It is required, that the names of the EDDIInputs match the required inputs of the EDDI monitor. For ConSerts, that means, that the EDDIInputs match the names of the demands and runtime evidence. The config file can be created manually when the ROS network and the published messages are known. Alternatively, a major part of the config file can be generated as described in section 3.6.

4.2.5 Generate the ROS Nodes

The next step generates the EDDI ROS package and for each model a runnable ROS node. Therefore, the generator.py script is used with

```
generator.py -w <catkin_ws> -c <config.yml> -m <model>
```

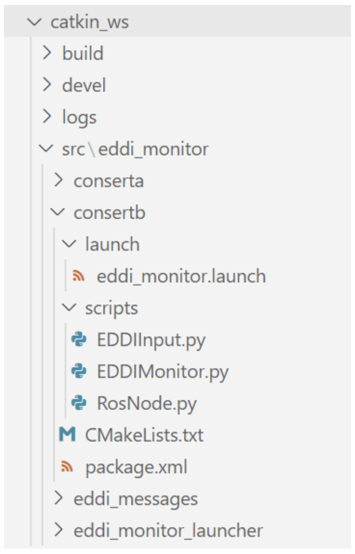
Where catkin_ws is the newly created or existing path to the catkin workspace. The config.yml and the model file. In this case the consert_a_config.yml and the ConSertA.yml are provided in the first step as parameters. The catkin workspace as shown in Figure 38 is then populated with several files and folders as shown in Figure 40. The first call of an empty catkin workspace will first generate the meta EDDI ROS package "eddi_monitor". Further, the "eddi_messages" package and the "eddi_monitor_launcher" were generated. Additionally, the ROS package with the ROS node for ConSertA was generated



```
catkin_ws
  build
  devel
  logs
  src\eddi_monitor
    conserta
      launch
        eddi_monitor.launch
      scripts
        EDDIInput.py
        EDDIMonitor.py
        RosNode.py
      CMakeLists.txt
      package.xml
    eddi_messages
      msg
        BayesianNetworkOutput.msg
        ConSertOutput.msg
        Guarantee.msg
        NodeOutput.msg
        OutcomeOutput.msg
      CMakeLists.txt
      package.xml
    eddi_monitor_launcher
      launch
      CMakeLists.txt
      package.xml
```

Figure 40 Catkin workspace after the generation of ConSertAs' ROS Node

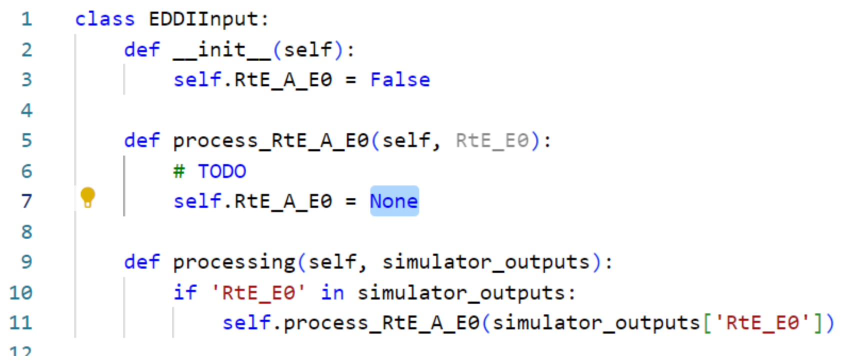
Next, the same command is executed but with the ConSertB.yml and the consert_b_config.yml files. This time, only the ROS package is created which correlates to the ROS node of ConSertB. It is then added in the "eddi_monitor" package as shown in Figure 41.



```
catkin_ws
  build
  devel
  logs
  src \ eddi_monitor
    conserta
    consertb
      launch
        eddi_monitor.launch
      scripts
        EDDIInput.py
        EDDIMonitor.py
        RosNode.py
      CMakeLists.txt
      package.xml
    eddi_messages
    eddi_monitor_launcher
```

Figure 41 EDDI ROS Package structure after ConSertB was generated

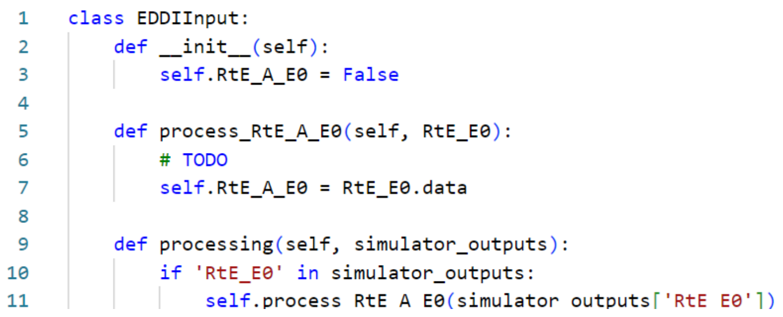
Now, after all required files were generated some minor adaptations are necessary. Within each ROS node, there is a EDDIInput.py file defining processing function for each EDDIInput specified within the configuration file. There, the developer must manually adapt the code such that the data which is propagated to the EDDI monitor is in the right format.



```
1   class EDDIInput:
2       def __init__(self):
3           self.RtE_A_E0 = False
4
5       def process_RtE_A_E0(self, RtE_E0):
6           # TODO
7           self.RtE_A_E0 = None
8
9       def processing(self, simulator_outputs):
10          if 'RtE_E0' in simulator_outputs:
11              self.process_RtE_A_E0(simulator_outputs['RtE_E0'])
```

Figure 42 EDDIInput.py of ConSertB before manual adaptation

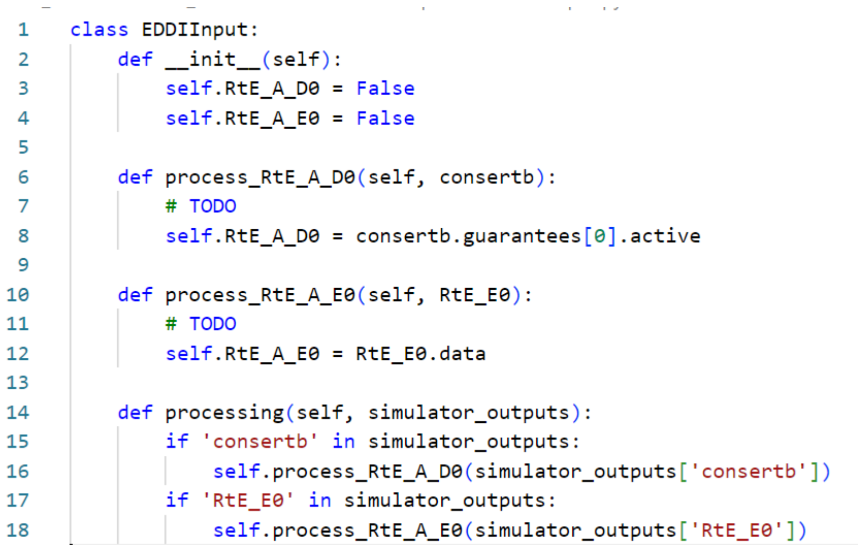
Figure 42 shows this file, where in line 7 the code must be adapted manually. In this example, the Boolean value of the standard message type Bool must be extracted from the ROS message. Therefore, the line changes as shown in Figure 43.



```
1   class EDDIInput:
2       def __init__(self):
3           self.RtE_A_E0 = False
4
5       def process_RtE_A_E0(self, RtE_E0):
6           # TODO
7           self.RtE_A_E0 = RtE_E0.data
8
9       def processing(self, simulator_outputs):
10          if 'RtE_E0' in simulator_outputs:
11              self.process_RtE_A_E0(simulator_outputs['RtE_E0'])
```

Figure 43 EDDIInput.py of ConSertA after manual adaptation

Similarly, the EDDIInput file for ConSertA must be adapted accordingly. In Figure 44 a possible solution is shown.



```python
class EDDIInput:
    def __init__(self):
        self.RtE_A_D0 = False
        self.RtE_A_E0 = False

    def process_RtE_A_D0(self, consertb):
        # TODO
        self.RtE_A_D0 = consertb.guarantees[0].active

    def process_RtE_A_E0(self, RtE_E0):
        # TODO
        self.RtE_A_E0 = RtE_E0.data

    def processing(self, simulator_outputs):
        if 'consertb' in simulator_outputs:
            self.process_RtE_A_D0(simulator_outputs['consertb'])
        if 'RtE_E0' in simulator_outputs:
            self.process_RtE_A_E0(simulator_outputs['RtE_E0'])
```

Figure 44 EDDIInput.py of ConSertA after manual editing

4.2.6 Running the ROS Nodes

Now the ROS nodes can be started. Therefore, catkin is built again with

```
catkin build
```

Then, it must be ensured, that the RosNode.py files are executable. Therefore, run

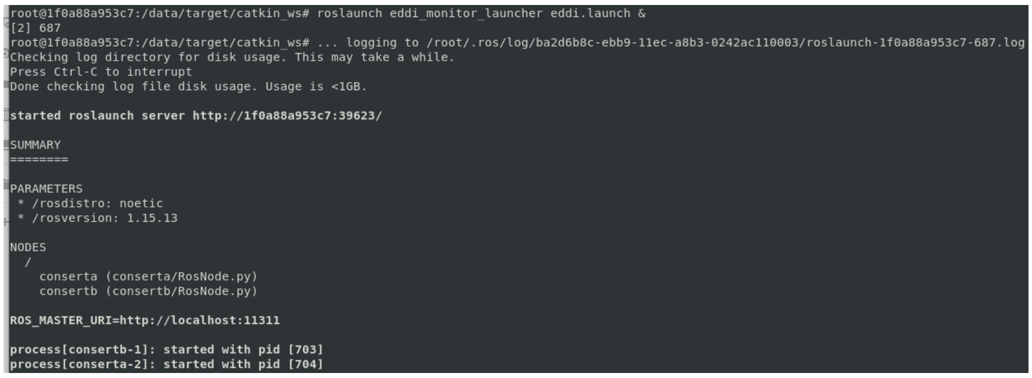
```
chmod +x src/eddi_monitor/<consert_name>/scripts/RosNode.py
```

Afterward, roscore is started with

```
roscore &
```

Now, launch the nodes with

```
roslaunch eddi_monitor_launcher eddi.launch &
```



```
root@1f0a88a953c7:/data/target/catkin_ws# roslaunch eddi_monitor_launcher eddi.launch &
[2] 687
root@1f0a88a953c7:/data/target/catkin_ws# ... logging to /root/.ros/log/ba2d6b8c-ebb9-11ec-a8b3-0242ac110003/roslaunch-1f0a88a953c7-687.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://1f0a88a953c7:39623/

SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.13

NODES
  /
    conserta (conserta/RosNode.py)
    consertb (consertb/RosNode.py)

ROS_MASTER_URI=http://localhost:11311

process[consertb-1]: started with pid [703]
process[conserta-2]: started with pid [704]
```

Figure 45 Launched EDDI ROS nodes

In order to test if everything works, message can be published manually with

```
rostopic pub /rte/e0b std_msgs/Bool "data: false"
```

for ConSertB's runtime evidence (RtE). For ConSertA's RtE a message is published with

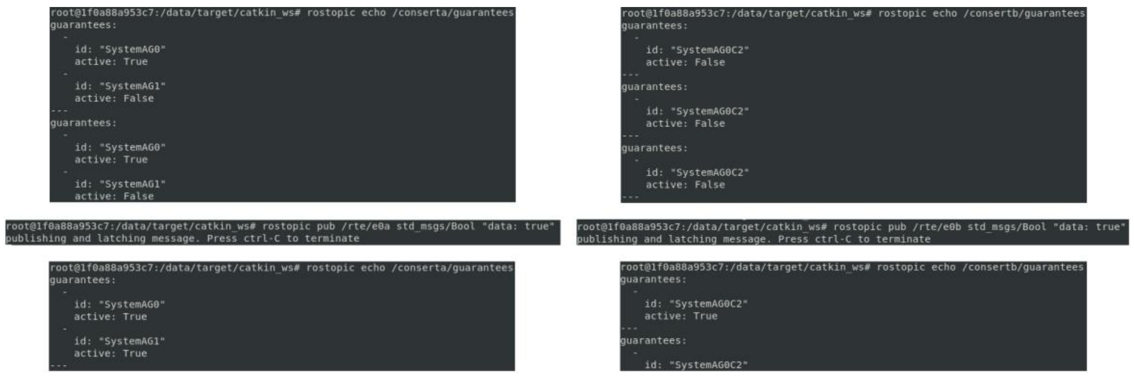
```
rostopic pub /rte/e0a std_msgs/Bool "data: false"
```

The guarantees can be printed with

```
rostopic echo /consertb/guarantees
```

```
rostopic echo /conserta/guarantees
```

respectively. Printing evaluated guarantees of ConSertA and ConSertB prior and subsequent to manually publishing RtEs is shown in Figure 46.



```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic echo /conserta/guarantees
guarantees:
  -
    id: "SystemAG0"
    active: True
  -
    id: "SystemAG1"
    active: False
---
guarantees:
  -
    id: "SystemAG0"
    active: True
  -
    id: "SystemAG1"
    active: False
```

```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic pub /rte/e0a std_msgs/Bool "data: true"
publishing and latching message. Press ctrl-C to terminate
```

```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic echo /conserta/guarantees
guarantees:
  -
    id: "SystemAG0"
    active: True
  -
    id: "SystemAG1"
    active: True
---
```

```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic echo /consertb/guarantees
guarantees:
  -
    id: "SystemAG0C2"
    active: False
---
guarantees:
  -
    id: "SystemAG0C2"
    active: False
---
guarantees:
  -
    id: "SystemAG0C2"
    active: False
---
```

```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic pub /rte/e0b std_msgs/Bool "data: true"
publishing and latching message. Press ctrl-C to terminate
```

```
root@1f0a88a953c7:/data/target/catkin_ws# rostopic echo /consertb/guarantees
guarantees:
  -
    id: "SystemAG0C2"
    active: True
---
guarantees:
  -
    id: "SystemAG0C2"
```

Figure 46 Example of manually publishing RtE messages and printing evaluated guarantees messages of ConSertA (left) and ConSertB

5. CONCLUSION

This deliverable described the first iteration of the SESAME tool-chain to semi-automatically generate runtime EDDIs, i.e. executable components to perform dynamic dependability management, from design-time EDDI representations. The realized toolchain consists on the one hand of generator components for several individual runtime dependability techniques, which are finally combined to represent the runtime EDDI capable of performing runtime dependability management. In addition to the platform-independent techniques, wrapper generators have been developed to integrate the platform-independent runtime EDDI code into the ROS platform. Due to this split between technique logic and platform logic, porting runtime EDDIs to different target robotic platforms is facilitated.

One assumption taken for this first toolchain iteration was to treat the runtime EDDI as one component that is deployed either on a cloud server communicating with the MRS or on one robot of the MRS, i.e. a centralized dependability management. To be more flexible with respect to deployment of runtime EDDIs, it is envisioned for the second iteration (in harmonization with the concepts researched in D7.3) to support distributed execution of runtime EDDIs, i.e. that modular deployment and execution onto several robots of the MRS is possible. This requires the implementation of communication protocols and the realization of further user intervention into the runtime EDDI generation process to integrate intended distribution schemes.

In addition, the initial prototypes developed for dynamic reliability assessment, perception uncertainty monitors with SafeML and the conditional event monitor, will be integrated into the outlined toolchain architecture so that sophisticated semi-automated generation of these monitors will also be fully based on design-time EDDI artifacts. On the macro-level SESAME scope, the runtime toolchain described in this deliverable will be integrated into the SESAME tool platform described in D8.3.