SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D5.4 Tailorability of EDDIs

Version 1.0
29 June 2022
Final

Public Distribution

Fraunhofer IESE

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 draft with outline and first content	1 May 2022
0.2	Outlined more sections	04 June 2022
0.3	Tool adapter content added	09 June 2022
0.4	EDDI ROS tailorability content added	23 June 2022
0.5	Finalizing first version	24 June 2022
0.6	First version ready for internal review	24 June 2022
0.7	Review from FORTH received	26 June 2022
0.8	Review from LU received	29 June 2022
1.0	Review changes integrated, ready for submission	29 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. Technologies and related techniques for adapting the EDDI and associated infrastructure should facilitate the deployment and integration of EDDIs, such that the corresponding effort and resource investment is reduced. Notably, as each MRS application often has unique characteristics, tailoring the EDDI components to fit is a necessary activity. Simultaneously, the proposed approach supports reusability of the EDDI components, as the coupling between EDDI design-time tools and between EDDI runtime components and their platform is reduced.

In this deliverable, we discuss existing technologies that facilitate the above activities, and propose a set we deem appropriate for tailoring EDDIs to the use cases in SESAME.

More detail on design-time EDDIs and related tooling is presented in WP4 deliverable D4.4. Further information on runtime EDDIs and related infrastructure is presented in WP7 deliverables D7.1 and D7.2.

LIST OF ABBREVIATIONS

SESAME	Secure and Safe Multi-Robot Systems	MRS	Multi-Robot System
DDI	Digital Dependability Identity	ODE	Open Dependability Exchange
EDDI	Executable Digital Dependability Identity	ROS	Robot Operating System
IDL	Interface Definition Language	DLL	Dynamically Linked Library
SO	Shared Object	XML	Extensible Markup Language
ConSerts	Conditional Safety Certificates	BN	Bayesian Network
EMF	Eclipse Modelling Framework		

1. INTRODUCTION

1.1 OVERVIEW

Integration is a typical part of a systems development lifecycle, and robotics systems development is certainly no exception to this rule. With runtime EDDIs, we are proposing the integration of additional components onto existing, complex MRS architectures. This process introduces an additional cost in development resources dedicated to performing the integration task itself, as well as resources dedicated to validating that the integration was correct.

But integration also occurs during development. For instance, when composing an architecture of heterogeneous MRS and/or their components, different toolchains may be necessary for dependability assurance. For example, analysing and deriving appropriate requirements for both testing and formal method verification might be meaningful, but those two approaches may involve different toolchains that are usually incompatible. It is meaningful that the inputs and results of different toolchains can still be easy to combine, especially if this process should be repeated for further MRS development.

Task 5.3 is responsible for identifying effective and making available means of tailoring EDDI components such that these tool and system integration costs become less of a burden on the development process. The proposed approach is derived from existing ideas found in programming languages (e.g. mixin layers and generics support) and model-based engineering (e.g. metamodeling templates and model subtyping). That being said, none of the above concepts is directly adopted for use here. Instead, the principle of indirection, which these concepts represent at different layers of abstraction, is inherited, adapted and applied for our purposes.

The discussion of the tooling in this deliverable is focused on the EDDI tailorability and reusability aspects. Further guidance and details on using the tooling itself can be found in SESAME deliverables D4.4 (for the ODE tool adapter) and D7.2 (for the MROS-based ROS tailorability). Brief guidance for the MROS-based tools can also be found in D8.3.

In the following section, section 0, we discuss further what challenges and existing options are known and related to the integration of EDDI components onto dependable MRS. In section 0, we discuss specifics about our approach towards EDDI tailorability. Finally, we conclude in section 0 with a brief summary and discussion of the next steps.

2. MOTIVATION AND BACKGROUND

MRS robots, and their constituent systems and components, form a complex runtime network. EDDI runtime components need to embed within this network, acting as monitors and intermediaries for control decisions. To support a range of heterogenous MRS applications, this means that interfaces between the EDDIs and the MRS must consider some of these possibilities:

a. The interfaces are fully implemented on an individual project basis by the system stakeholders. Making changes to the EDDI interfaces requires enacting corresponding manual changes to the interfaces. The effort required may be a barrier or at least disincentivize the adoption of EDDIs.

b. Require that target MRS adhere to existing standards that govern communication interfaces between MRS. The MRS developers must invest effort in complying with the interface specification, but this process can be standardized with guidelines and reusable components. Unfortunately, arriving at generic interfaces that fit all kinds of MRS applications is challenging, and arguably not the focus of the project. That being said, EDDIs will try and provide support for common interface standards found in the project's use cases; the first target shall be use cases featuring the ROS, see sections 2.2 and 3.2.

c. Tool support assists with the interfacing; the interfaces are specified generically through e.g. Interface Definition Languages (IDLs), and code generation allows most of the process to be streamlined, requiring fewer code changes in the host MRS to integrate the EDDIs.

Furthermore, having highly tailorable EDDIs offers additional potential benefits:

- EDDIs should become more easily re-usable across development projects, fostering the accumulation of private and public EDDI libraries.

- Manual integration is oftentimes a tedious process, involving repeatably coding of boilerplate code e.g. for input/output transformation. Tailorable EDDIs should facilitate this process, and allow effort to be focused on more critical tasks instead.

- Errors during the integration can be more easily overlooked, as the EDDI is an additional component outside the nominal development lifecycle (albeit potentially also critical). Tailorable EDDIs should allow semi-automatic processes to generate the necessary runtime components and 'glue code', which makes the results more predictable and easily checkable.

- Auto-generated code is more easily optimized compared to ad-hoc manual integration code; therefore, there are also potential efficiency improvements when considering tailorable EDDIs.

We have opted for the IDL with code generation, as it provides flexibility in both the direction of the MRS stakeholders, as well as the EDDI as a standard. On the MRS stakeholder side, developers can adjust the interface specification and EDDI component generation to their application's needs. On the EDDI's side, modular tools can be developed with distributed effort to support a large variety of MRS applications, with

new applications expected to require at most only retargeting the code generation for the new platforms. Further details are discussed in section 0; for the remainder of this section, we review and discuss existing work relevant to adapting to MRS applications. Before doing so, we touch upon the relationship of our work with that found in other SESAME WPs.

2.1 RELATION TO OTHER WP EFFORTS

Introducing EDDI/ExSces and other SESAME methods, tools and components to an MRS application means that an understanding of how the existing application is structured both conceptually and technically is required.

Other parts of the SESAME methodology address understanding the conceptual structure of the MRS application, for specific concerns. Specifically, WP1 is responsible for specifying project, tool, and use case requirements, such that the EDDI models, tools, and runtime components can be developed according to the needs of the use case stakeholders. WP1 provides us with a clearer understanding on the expected targets of tailorability (tools and MRS), but we still need to identify a technical design and solutions that deliver on these requirements. Each of the remaining WPs from WP2 to WP7 addresses specific aspects throughout the MRS lifecycle; they introduce models, methods, tools, and design patterns that need integration to be applied. WP8 is responsible for orchestrating this integration on the project level; EDDI tailorability targets lower-level mechanisms that facilitate the technical integration between EDDI tools, and between EDDIs with MRS applications.

2.2 INTEGRATING WITH ROBOT OPERATING SYSTEM APPLICATIONS

The Robot Operating System (ROS) [1] is arguably one of the most popular options for implementing MRS applications. Its robot communication infrastructure, as well as its robust suite of supporting software libraries allow significant ease of access, flexibility, and scalability that can be adopted by both hobbyists, as well as professional developers.

Binding external components into an existing ROS MRS requires, as a first step, gaining an understanding of the target architecture. The following questions could be useful to be answered:

- Which nodes are there?
- Which topics are there?
- What message types are supported/exchanged? At what rate does communication occur?
- Which parameters are expected, what types and value ranges are valid?
- How does the execution flow, and what runtime characteristics does it feature?

Once a clear view of the target architecture, its elements and their properties, as well as its dynamics, is obtained, there should be a much clearer pathway towards integrating EDDIs onto it.

2.2.1 Built-in ROS tools for architecture inspection

As mentioned, ROS features a rich set of open-source software libraries, as well as many standardized command-line tools. Some of these tools are of particular interest, as they enable semi-automatic investigation of the ROS architecture of a given MRS. These include:

- rospack[1], a command-line tool that displays information about specific ROS packages, and also enables listing of package and plugin dependencies. This can be used to investigate existing code dependencies.

- rostopic[2], a command-line tool that displays information about specific ROS topics, including a list of topics currently available, the types of messages supported by a topic, bandwidth consumed by a topic, etc.

- rosgraph[3], a command-line tool that displays information about the current ROS node graph. Its graphical counterpart is rqt_graph.

- rqt_graph[4], a graphical tool that depicts ROS nodes and other elements in a directed graph. An example of how the interface appears can be seen in Figure 1. Additionally, when using ROS with Python or C++, statistics can be directly sampled and annotated automatically on the elements of the nodes, which allows for live inspection and monitoring of ROS applications.

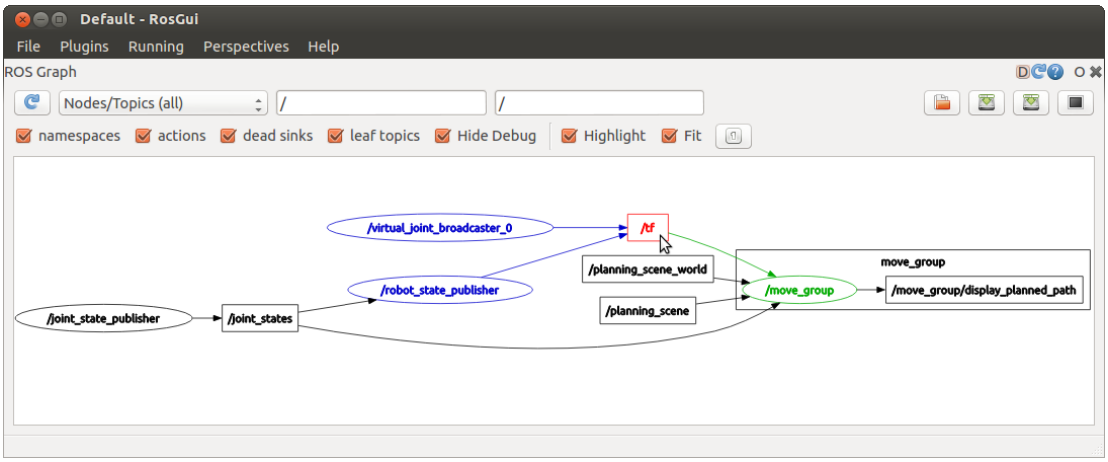

Figure 1: Example of rqt_graph[5]

While the built-in ROS tools are very useful in understanding how a ROS system is structured and operates, they are not sufficient on their own to effectively integrate EDDI components onto existing MRS applications. Their fundamental limitation is that, while they do support information gathering and graphical visualization of existing architectures, they don't support interactive modelling for planning out changes to the MRS application; these need to be performed offline, and then the MRS application

[1] http://wiki.ros.org/rospack
[2] http://wiki.ros.org/rostopic
[3] http://wiki.ros.org/rosgraph
[4] http://wiki.ros.org/rqt_graph
[5] http://wiki.ros.org/rqt_graph

must be rebuilt, deployed and re-evaluated with the tools. The approach we propose in section 3.2 addresses this limitation by supporting interactive modelling of the MRS changes and extending the process into tailoring of the EDDI interfaces for the runtime components.

2.2.2 Built-in ROS tools for integrating external components

ROS also features some support for integrating executable components onto an existing application. The class_loader[6] package enables developers to integrate C++ code at runtime using runtime libraries (e.g. 'Dynamically Linked Libraries' – DLLs or 'Shared Objects' – SOs), without requiring a rebuild of the application. This is possible by specifying classes (in the external code module) that subclass a predetermined interface already built in the ROS application. The integrating code can then load the DLL or SO at runtime, inspect it to retrieve appropriate classes that inherit from the known interfaces, and then create new objects from them.

Another option is pluginlib[7], which builds upon the functionality of class_loader, by enabling convenient querying of the available plugin packages via the command-line. This means that developers can make their plugin available (again, as a runtime library e.g. DLL/SO), create a plugin description XML file, and then publish the plugin to the ROS ecosystem. Once published, the plugin packages can be located via built-in ROS commands.

In theory, either of these options could be used for specifying plugins based on EDDI components, which can be loaded and used by the MRS application at runtime. An overview of this workflow can be seen in Figure 2.

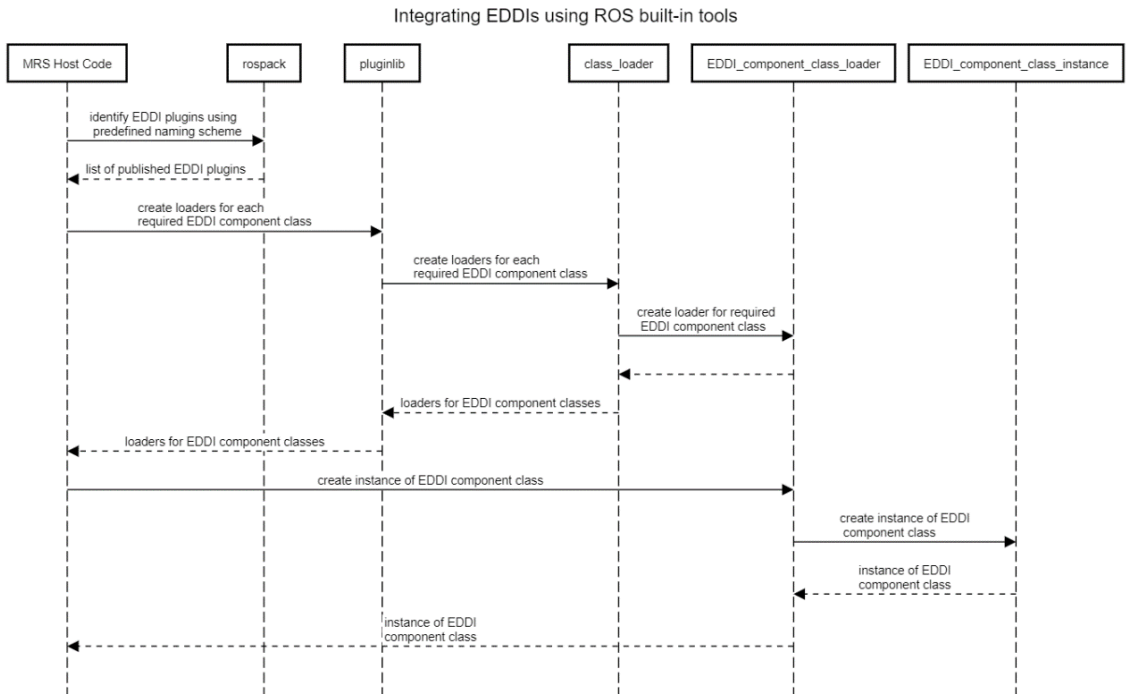

Figure 2: Conceptual Example of using ROS built-in tools for EDDI component integration

[6] http://wiki.ros.org/class_loader
[7] http://wiki.ros.org/plugianlib

In the above example, the MRS host code can use rospack to find which EDDI plugin packages have been published in the ROS system. The code should presumably have a mechanism for knowing which plugin packages, components, and classes it requires. Once the target plugins have been identified by file path, loaders (i.e. factories) for each of the desired EDDI components can be created using pluginlib. Pluginlib invokes class_loader to dynamically load each EDDI component's DLL or SO into memory, and then create and return a class loader for the EDDI component to the host code. Finally, using the class loader, instances of the EDDI component classes can be created. We should remind and highlight that the above should be possible without rebuilding the host application.

The above approach has limitations for SESAME, which is why we have not adopted it:

- The above solution is supported for C++, but it's unclear whether there's equivalent direct support for this approach in Python. This would make the implementation more complex in those cases, as C++ binding interfaces would have to be used for EDDI components developed in Python.

- Integrating into existing MRS applications with this approach requires that the MRS application incorporate additional runtime mechanisms that perform filesystem access, and DLL/SO loading during the system setup. While this might not be critical for most cases, we wanted to avoid introducing this kind of requirement onto host applications.

- The approach requires that the host application is built with a generic interface in place, from which the EDDI component classes must inherit. While this can enable the EDDI component classes to provide varied interfaces (e.g. by including generic interface discovery functions), it may also introduce runtime overhead, that could be problematic in MRS systems with either high-performance requirements, or limited-resource availability. Once again, we wanted to avoid introducing such requirements onto the host applications, especially given that EDDI runtime components should minimize their execution footprint on the host application given their responsibility for dependability monitoring.

We discuss our approach in section 3.2, which, we believe, circumvents the above limitations for the most part.

2.3 INTEGRATING WITH OTHER PLATFORMS

ROS is hardly the only platform relevant to MRS. Even within SESAME, there are use cases that either do not use ROS, or have additional platforms being used that EDDI components should interact with. For instance, simulation platforms should also be considered for tailorability, as they are an important part of an MRS' development. Tailoring EDDIs for simulation environments, such as Gazebo[8], Unity[9], and/or Unreal Engine[10] would allow significantly more opportunities for adopting and applying EDDIs in further MRS applications.

[8] https://gazebosim.org/
[9] https://unity.com/
[10] https://www.unrealengine.com/en-US

For the moment, we focus our efforts on supporting ROS, but will consider extending tailorability support for some of these simulation platforms as the SESAME project continues.

3. EDDI TAILORABILITY IN SESAME

3.1 THE DDI/ODE COMMON TOOL ADAPTER

In an MRS, multiple robots shall interoperate in a safe manner. For interoperating safely and correctly at runtime, executable models have to be exchanged and processed. Such models have to be created at design time using modelling tools. For instance, a ConSert model can be created with safeTbox[11], or GeNIe[12] can be used to define Bayesian Networks (BNs). As other types of models can also be defined with different tools and several artefacts need to be processed at runtime within the same use case, a standardized and integrated model format, the EDDI, has been introduced, which is defined by the updated ODE metamodel, see SESAME deliverable D4.2. An EDDI model can therefore be used as a common interchangeable communication format for MRS.

Figure 3 shows a scenario where robots need to have both a ConSert and a Bayesian Network model to be processed at runtime to interoperate in an MRS correctly and safely. Both models are included within the robot's EDDI model. However, at design time, ConSert and BN models are defined in a different tool (safeTbox and GeNIe) and are represented with different formats. Thus, there is a need to somehow transform the modelling-tool-specific representation of the models into the standardized EDDI design-time model. Design-time EDDI models can be further transformed into EDDI software components that can be deployed and executed on MRS at runtime. As this chapter describes the design-time part, for simplicity in this section, where there is ambiguity, design-time EDDI models are called *EDDI*.

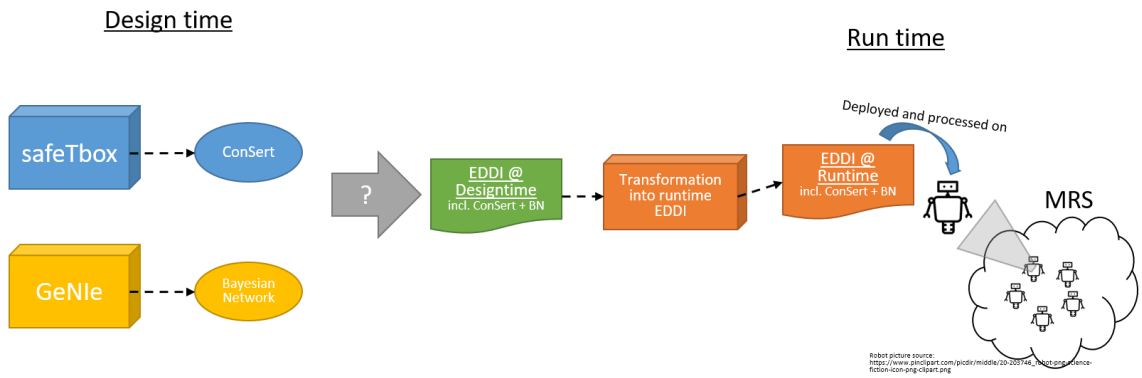

Figure 3: How to get from design time to runtime models?

The ODE metamodel has been defined using the Eclipse IDE and the Eclipse Modelling Framework (EMF)[13]. The instantiated EDDI file is serialized into an EMF model file. The Eclipse IDE can be used for manually creating an EDDI file that conforms to the ODE metamodel. This method of EDDI creation means that the models (e.g. ConSert & BN) would be defined visually in an arbitrary modelling tool (e.g. safeTbox & GeNIe) and are afterwards adapted manually into an EDDI model.

[11] https://www.safetbox.de
[12] https://www.bayesfusion.com/genie/
[13] https://www.eclipse.org/modeling/emf/

A manual EDDI model creation is time-consuming and prone to errors. Therefore, the EDDI/ODE Tool Adapter has been introduced to support the work with EDDI models at design time. It provides services for exporting and importing EDDI models from and to a modelling tool. Furthermore, it provides a service for executing Epsilon[14] scripts on existing DDI files. Epsilon is a family of scripting languages for executing model-based software engineering tasks. For instance, with the help of Epsilon scripts, EDDI models can be created, manipulated, merged, transformed and even validated.

Figure 4 shows the usage of the Tool Adapter for solving the problem in previously mentioned scenarios. After the ConSert and the BN models are created, they are exported from their modelling tools into separate EDDI models using the Tool Adapter. In the next step, the service for executing an Epsilon script is invoked on the Tool Adapter. The script requires both EDDI models as input, one containing the ConSert model and the other containing the BN model, to merge them into a single EDDI that can be deployed on the robot system.

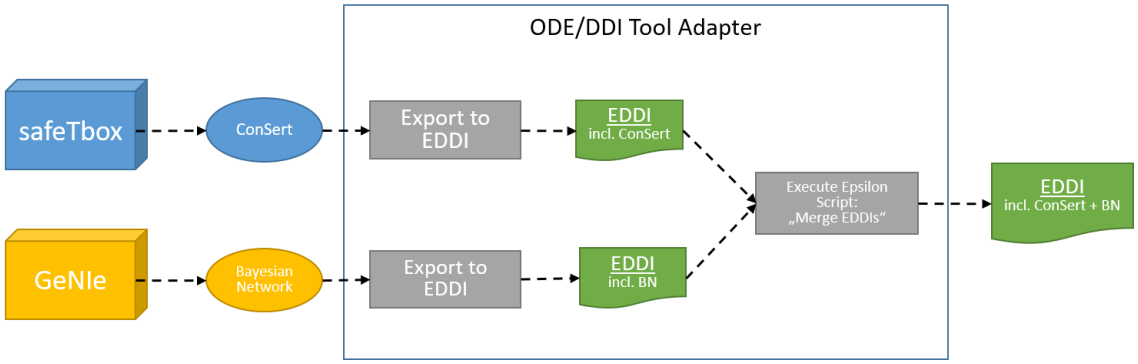

Figure 4: Tool Adapter usage to merge ConSert and BN into one EDDI

The Tool Adapter's service interfaces are designed using Apache Thrift[15], which allows tool interfaces to be defined in its own Interface Definition Language (IDL). Once the IDL specification is available, Thrift is then used to generate language-specific code for allowing the invocation of the service from any client application written in a language that is supported by Thrift. Thus, the services for exporting and importing EDDI models and executing Epsilon scripts on EDDI models are defined using Thrift. The import and export services need an exchange format for the model between the client (modelling tool) and server (Tool Adapter) applications. This exchange format is also defined using the Thrift IDL, from which language-specific code is generated as well. Specifically, it is the ODE/EDDI format transformed into Thrift data structures. For the example scenario, this implies that the modelling tools have to implement an adapter for transformations between internal and Thrift models for using the export and import services of the Tool Adapter. Figure 5 shows the updated export section of the previous example, where the modelling tools now transform their internal models into their Thrift representation to call the Export EDDI service of the Tool Adapter.

For further information regarding the Thrift service contract, data type definition and adapting a modelling tool, so that it can invoke the Tool Adapter services, please refer to SESAME D4.4.

[14] https://www.eclipse.org/epsilon/
[15] https://thrift.apache.org

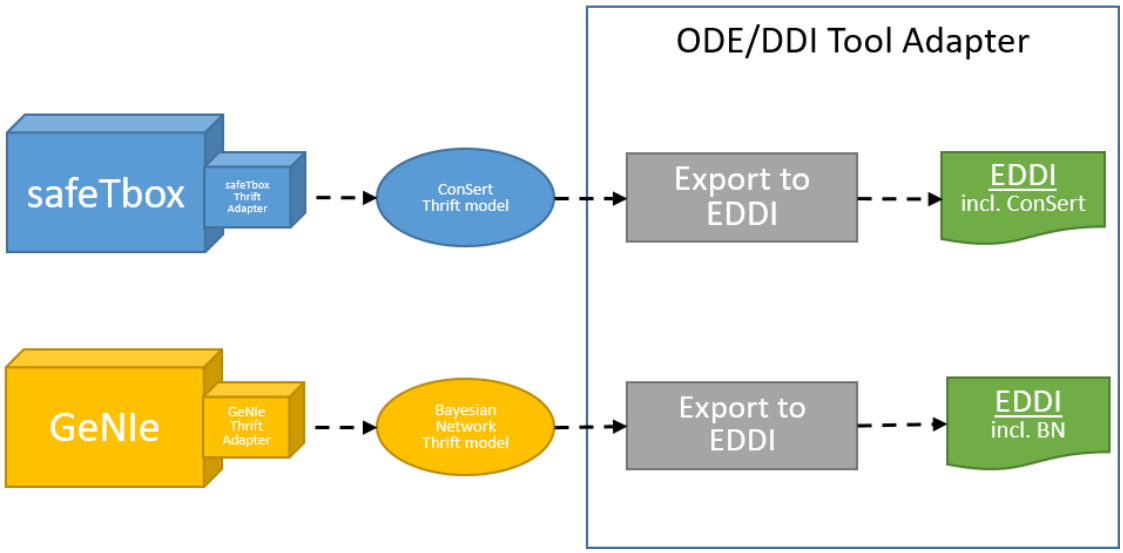

Figure 5: Export DDI model from modeling tools using Thrift exchange format

As mentioned before, EDDI model files are in fact EMF models that conform the ODE metamodel. Thus, the Tool Adapter has to transform the Thrift representation of a received model into the ODE (EMF) representation before it can be exported into an EDDI file. Hence, the Tool Adapter was implemented in Java, also using the Eclipse IDE, in order to perform the ODE metamodel transformations more directly.

An overview of the Tool Adapter's service interface and what internal components contribute to the service implementation is shown in Figure 6. In the following, each service and its Tool-Adapter-internal implementation are described briefly:

- **Export EDDI Model:** The client application invokes the service and provides the model to export in the Thrift EDDI representation. This Thrift representation is then transformed into the ODE EMF representation before it is exported as an EDDI model file.

- **Import EDDI Model:** The service expects the path to the EDDI model file as an input parameter. The Tool Adapter then loads the EDDI model file as an EMF model and transforms it into its Thrift EDDI representation before sending it to the client application that invoked the import service.

- **Execute Epsilon Script:** Epsilon Scripts can be executed on EDDI models, e.g. for manipulating or validating the models. The service expects information about the script execution as an input. This includes the path to the script file, as well the paths to the EDDI models that shall be affected by the script execution. Internally, the Epsilon Script Executor then uses the input information to perform the script execution on the EDDI model files.

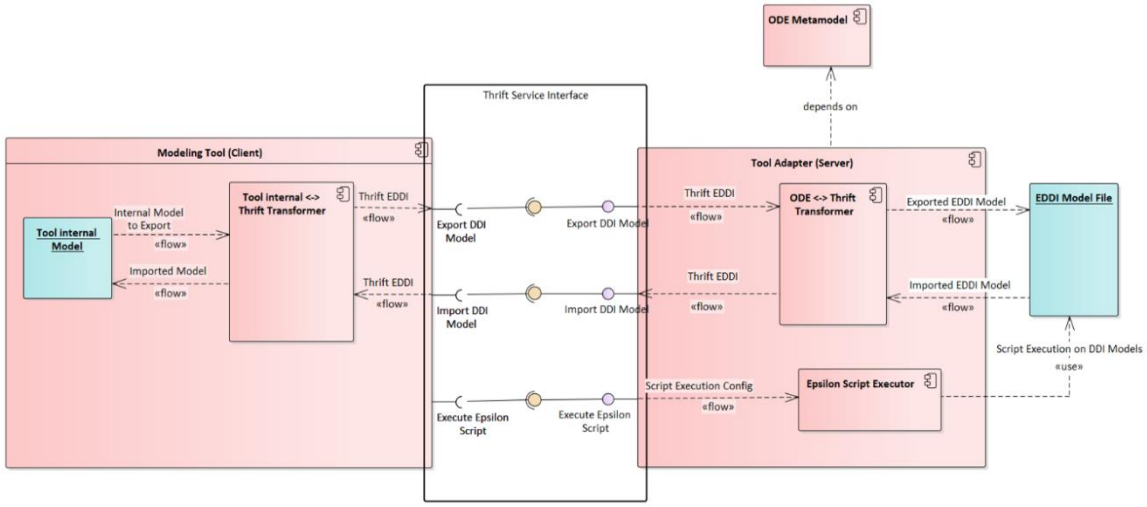

Figure 6: ODE/DDI Tool Adapter Services Overview

3.2 TECHNIQUES FOR ADAPTING TO APPLICATIONS FEATURING ROS

The ROS is a widespread middleware used in robotics applications. ROS is used to decouple processes in a distributed system. The publisher-subscriber interface allows developers to decouple nodes in space as well as in time and maintain synchronization. Furthermore, the ROS middleware supports communication between nodes that are written in different programming languages. Data can be exchanged through language-agnostic ROS messages. Using a middleware like ROS significantly reduces the number of adaptations necessary when new components are integrated into an existing ROS application.

In principle our developed runtime components can be applied independently of ROS. They are designed to be used with any middleware. To adapt the runtime components for use in ROS applications, we developed ROS Node Wrappers that automatically generate ROS nodes from the runtime components (their EDDI models to be precise). Integrating new ROS components with an existing ROS application only requires correct wiring between the components. In the case of the publisher-subscriber interface, the wiring is performed by adapting the topic names to which individual nodes publish and subscribe and by equalizing the message type between publishers and subscribers.

The following set of tools is applied to further facilitate the integration and wiring with existing ROS systems.

3.2.1 Static code analysis

Static code analysis is applied to an existing application to extract a model of the system. To extract the ROS system models, a compiled catkin workspace is analysed. The static code analyser parses the ROS packages and extracts ROS models of all the nodes. The output of the static code analysis is *.ros files for each node and, optionally, it is possible to also extract a *.rossystem file that describes the wiring between the respective ROS components.

3.2.2 Graphical Modelling

Visual manipulation of ROS models facilitates the integration process. ROS models can be adapted without requiring to interact with the textual model representations and without requiring understanding of the domain-specific language. Visualizations of nodes and topic connections between the node convey a better understanding of the whole system.

In the following, a standard workflow that builds on the output of the static code analysis and that produces a model of an integrated system is described.

The MROS modelling tool[16] provides an Eclipse Sirius[17] workbench for creating and adapting ROS models and ROS system models. To integrate EDDI components into a ROS system first a new ROS project is created by clicking the button provided by the workbench.

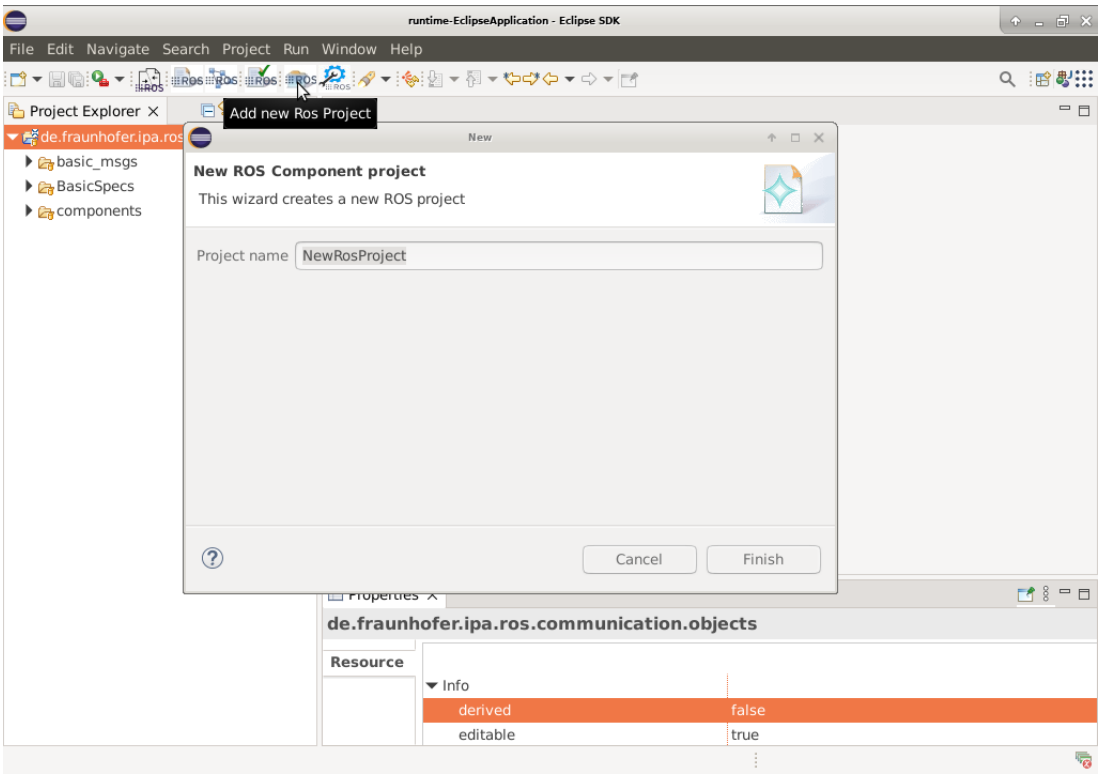

Figure 7: Creation of a new ROS project

In the new project, a file named `representation.aird` which contains Sirius representation data, is automatically created. *.aird files are opened with the Eclipse Aird editor. The Aird editor opens an alternative UI that simplifies the interaction with multiple different Sirius representations.

[16] https://github.com/ipa320/ros-model/blob/master/docu/Installation.md
[17] https://www.eclipse.org/sirius/

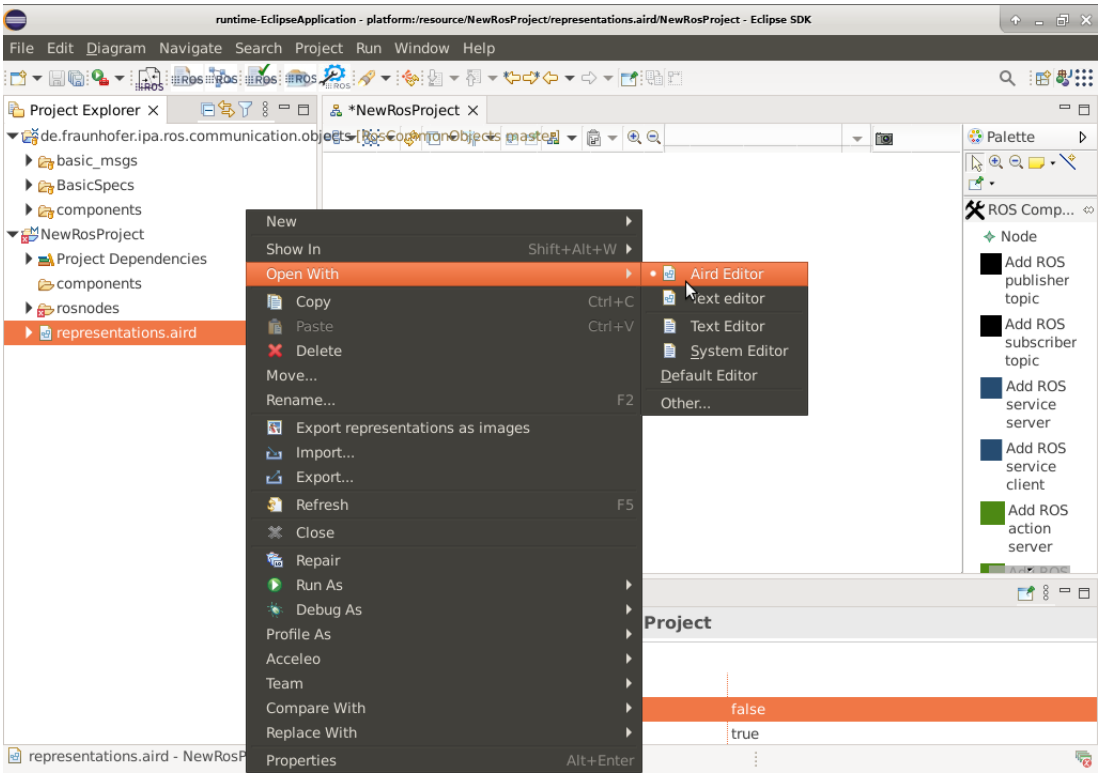

Figure 8: Opening the Aird editor

The Aird editor opens up a new window that allows model and model representation management. Through the editor, it is possible to both create new and add existing models. Clicking the *Add* button opens a dialog box which allows us to select representations files from the filesystem. Thereby, the *.ros files from the previous step are added to the project.

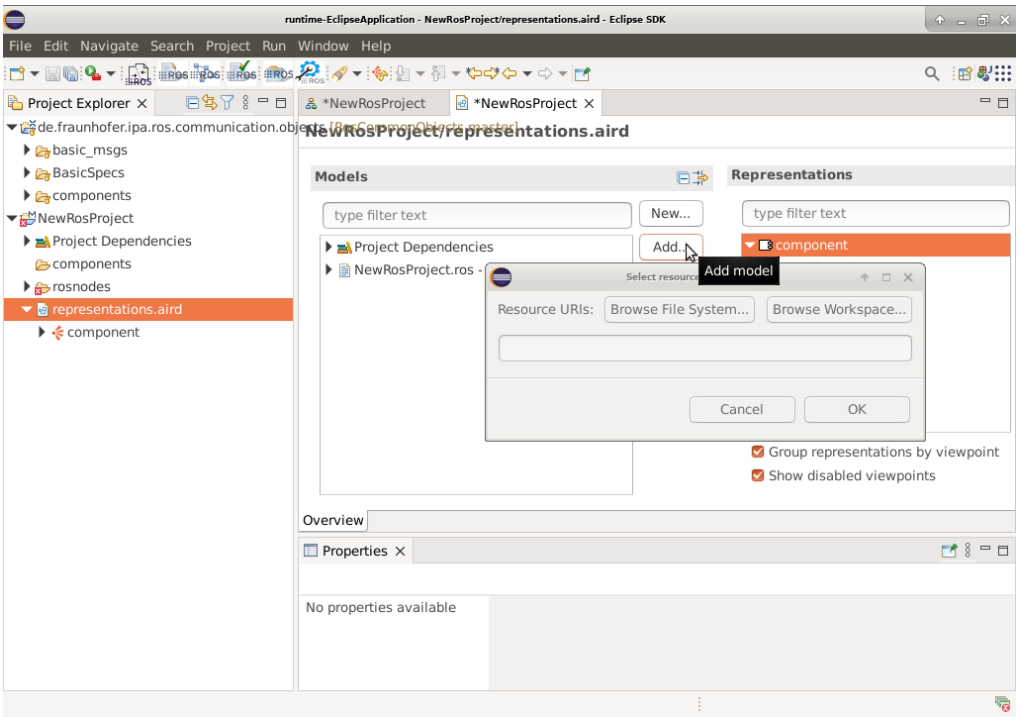

Figure 9: Adding existing ROS models

For all ROS components where no ROS model is available, a new model can be created. By clicking *New*, a dialog box opens with a list of metamodels. Select *ros* to create a ROS model or rossystem to create a new ROS system model.

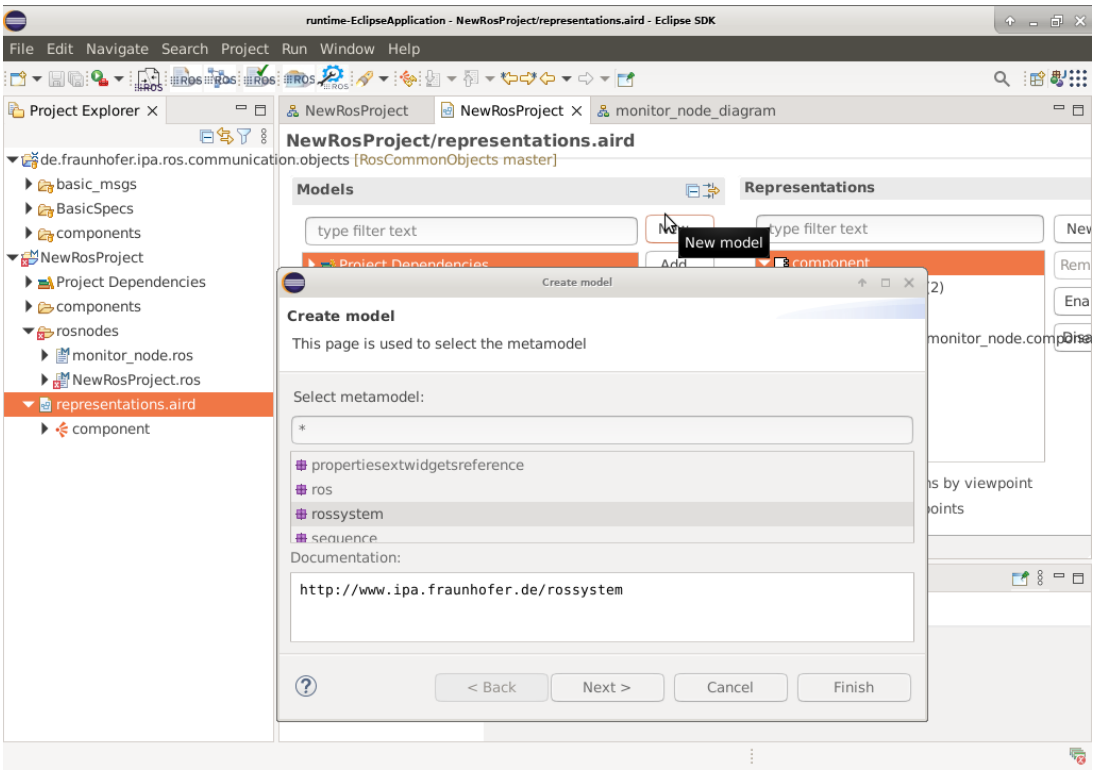

Figure 10: Creation of new ROS models

The new models do not have a graphical representation yet. This representation is required for graphical modelling. To add a representation to a ROS model or a ROS system model click *New* in Aird editor, next to the explorer for the representations. In the dialog that opens up, the representation type *Artifact diagram* or *RosSystem* as well as the respective model can be selected.

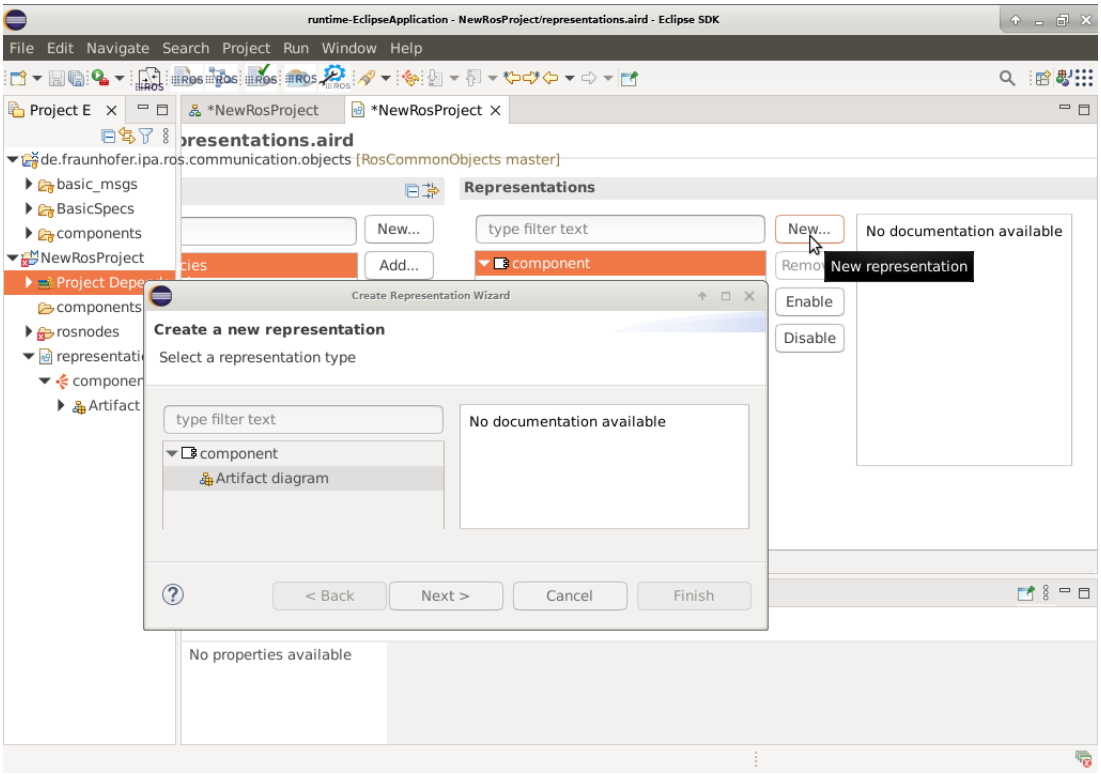

Figure 11: Creation of graphical model representations

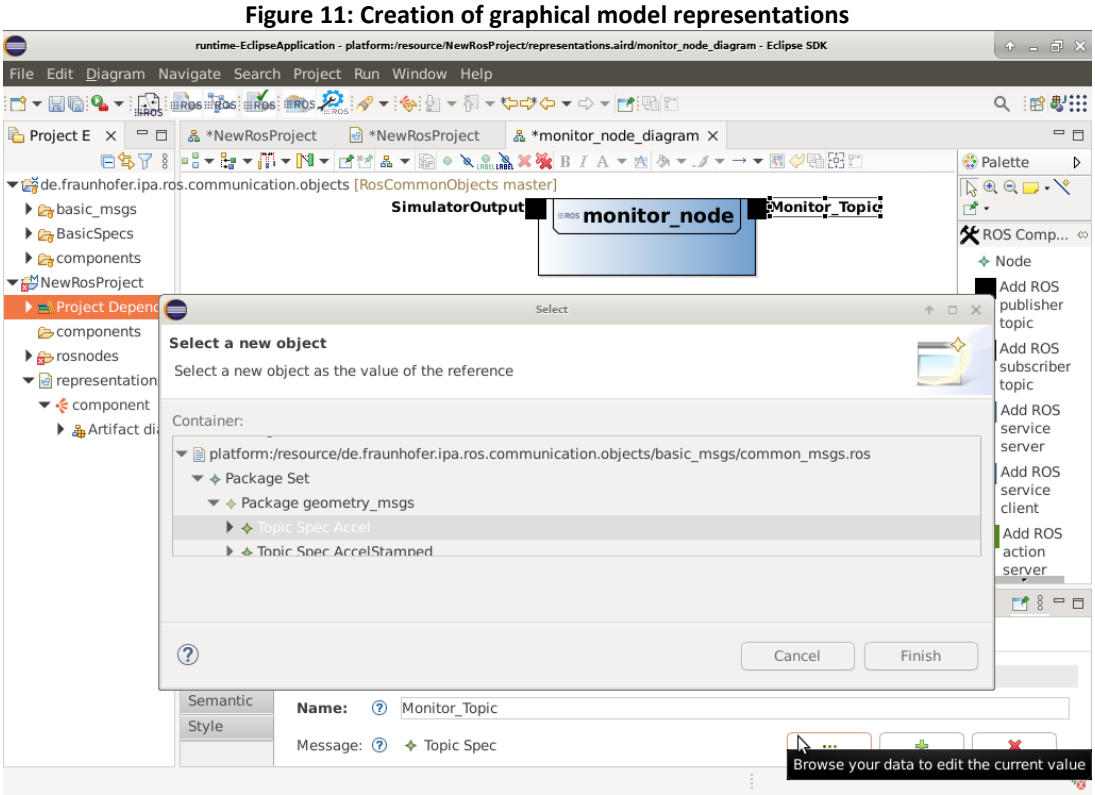

Figure 12 shows a node that has a ROS publisher called *MonitorTopic* and a ROS subscriber called *SimulatorOuput*. Using the editor, it is possible to add new publishers and subscribers to the ROS node. By clicking on the ports, it is possible to adapt the topic names and the topic message types. This way, it is possible to adapt ROS models to be able to integrate with existing ROS applications. New components are adapted

such that the publisher and subscriber names and the message types fit the existing application.

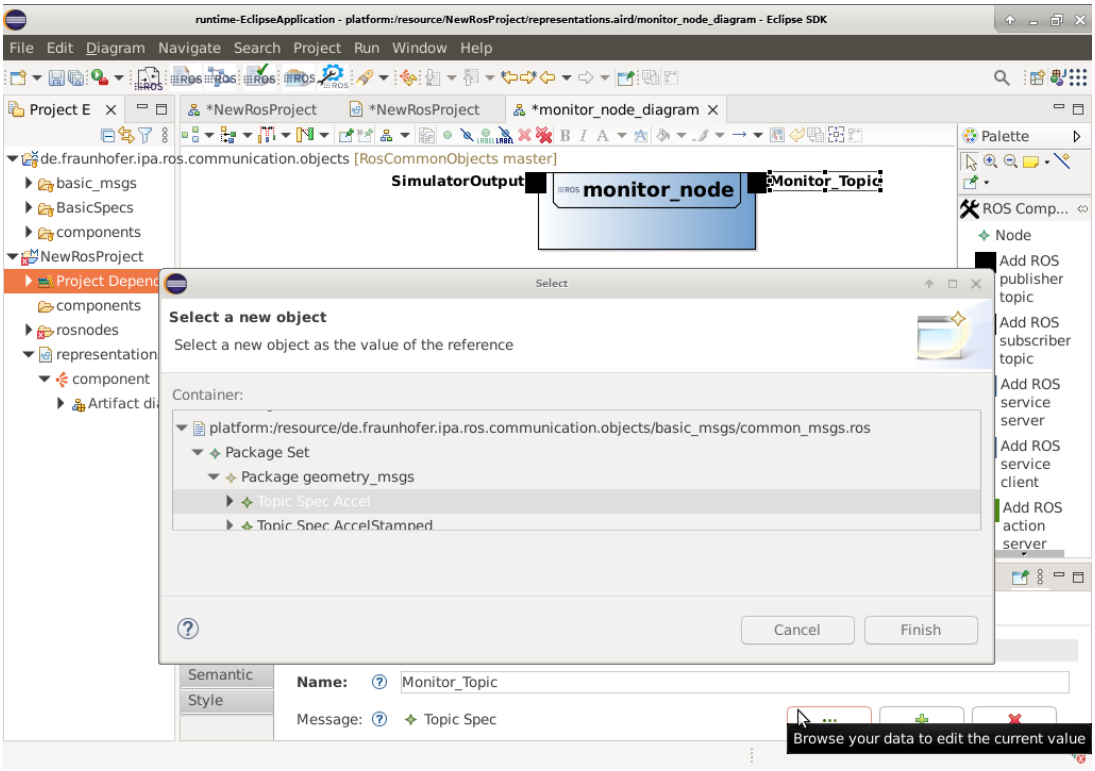

Figure 12: Adaption of ROS node topics

After adapting the individual ROS models, topic connections can be established between the components using a ROS system model. A new ROS system and a respective representation can be created using the button provided by the workbench or, as described previously, using the Aird editor. It is possible to add ROS systems as components in the new system.

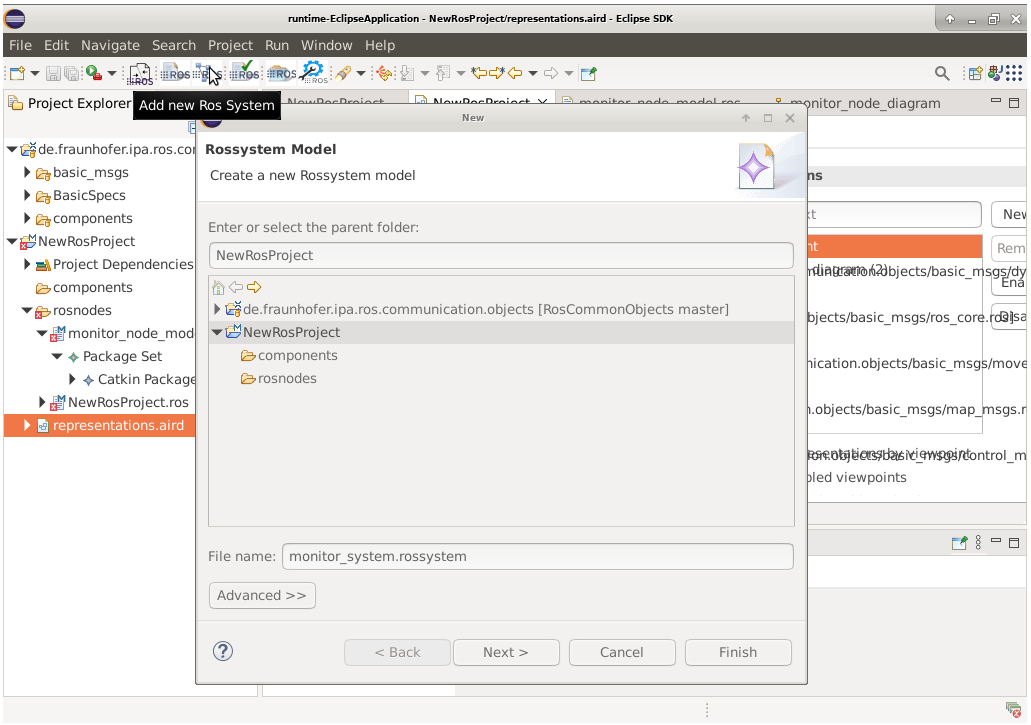

Figure 13: Creation of a ROS system model

The representation of the ROS system allows modelling of ROS components and connections. Clicking *New Component* opens a dialog that adds a component to the ROS system. Each component requires a name and a ROS model. Optionally, a namespace can be specified.

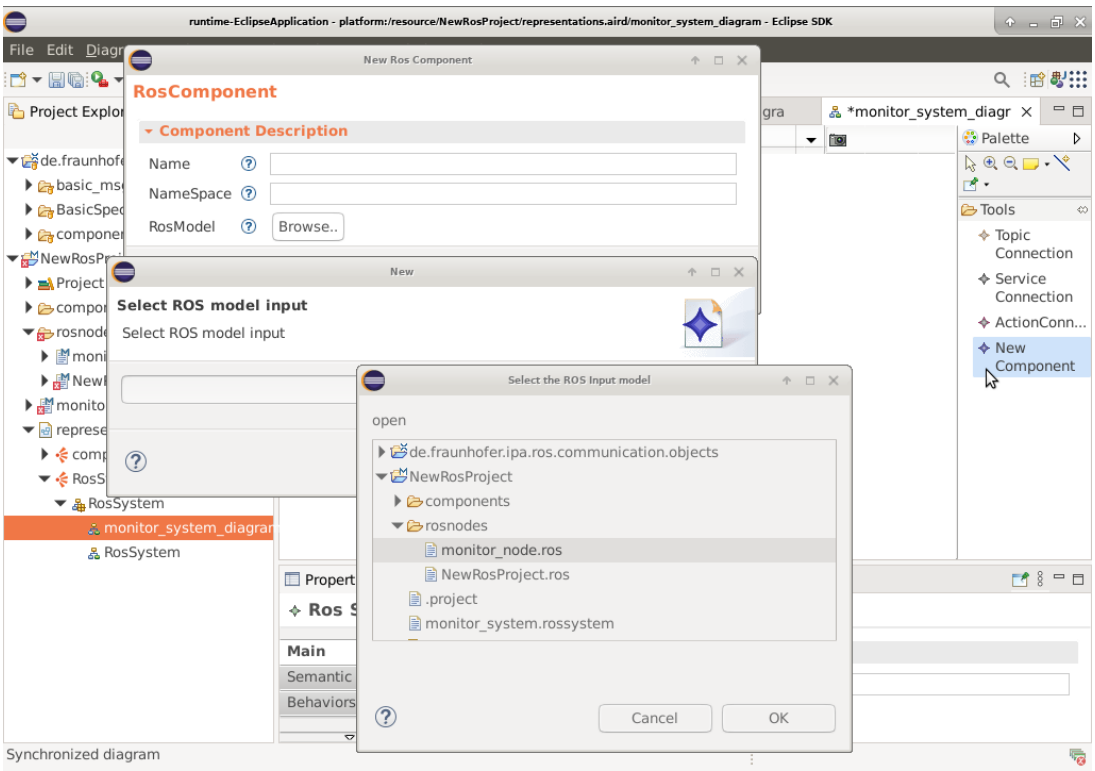

Figure 14 :Adding a ROS component to a ROS system model

Topic connections can be established between the components by clicking *Topic Connection* in the editor and the respective publishers and subscribers. Topic connections can only be created between publishers and subscribers that have the same message type. The ports e.g. the publishers and subscribers and message types are specified in the respective ROS model. They can be adjusted by editing the respective ROS model. The ports of the ROS component are then updated.

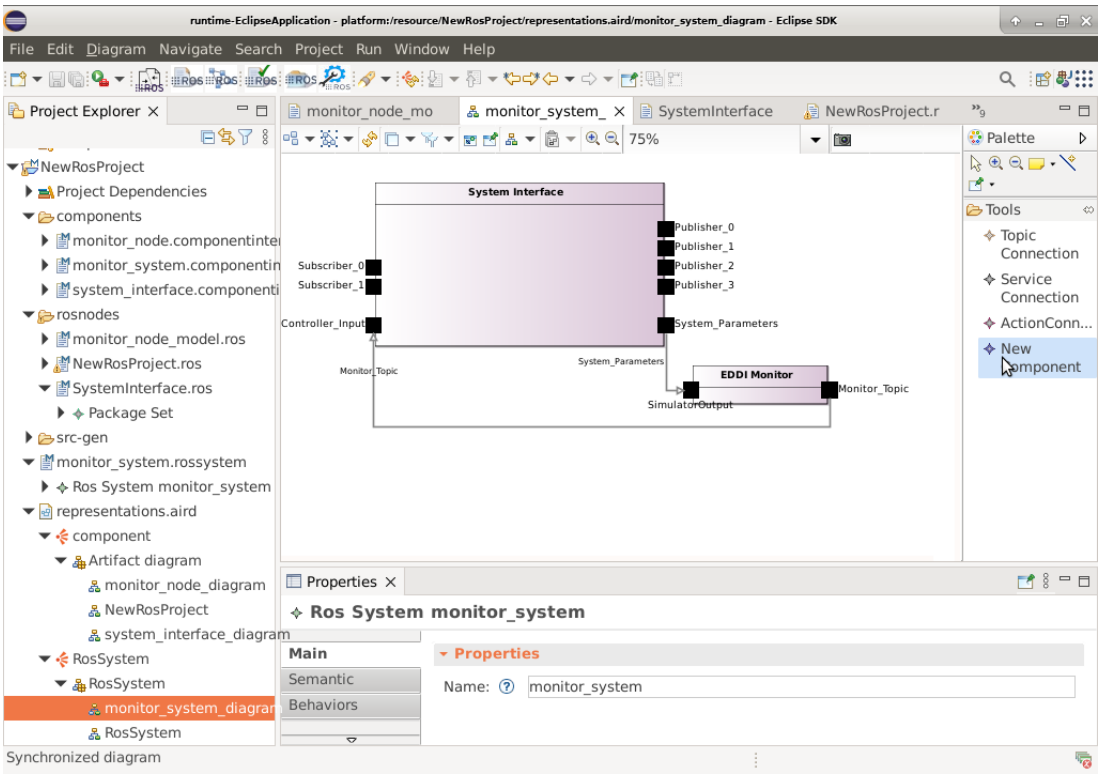

Figure 15: Example ROS system with components connected by topic connections

The result of the graphical modelling step is a *.rossystem file and corresponding *.ros files that describe the overall system where the existing application is integrated with new adapted components.

3.2.3 ROS Configuration

The EDDI components are generated according to a ROS configuration file. The EDDI components are adapted by changing this configuration file. The next step of the system integration consumes the *.rossystem file to create a tailored EDDI configuration file. This step can be automated since the ROS system model contains the information that is necessary adapt the EDDI to the existing system.

To assist the generation of the EDDI configuration, we provide an EDDI-system-parsing tool. This tool uses the concrete syntax trees of the domain-specific language (DSL) in which the model files are described. It is used to parse the integrated model files to extract the *id*, *topic name* and *message type* of relevant inputs to the EDDI. These inputs are then added to the EDDI configuration and saved in the SimulatorOutput field of the eddi_config.yml file. The configuration file is then ready to be consumed by the ROS Node Wrapper tool and the EDDI generation tools to generate EDDIs in ROS that are specifically tailored to a target application.

The EDDI components are tailored towards a specific ROS system by changing the configuration file. This configuration step of the system integration consumes the *.rossystem file to create a specifically tailored EDDI configuration file. This step can be automated since the ROS system model contains the information that is necessary to adapt the EDDI to the existing system. To assist the generation of the EDDI configuration we provide a the EDDI system parsing tool. This tool uses the concrete syntax trees of the domain-specific language (DSL) in which the ROS models and the ROS system models are described. It is used to parse the adapted ROS model files and ROS system model files to extract the *id*, *topic name* and *message type* of relevant inputs to the EDDI. These inputs are then added to the EDDI configuration and saved in the SimulatorOutput field of the eddi_config.yml file.

4. SUMMARY AND FUTURE WORK

In this deliverable, we have discussed our initial approach towards improving EDDI tailorability in SESAME. Our tailorability approach considers both development-time activities involving EDDI-related toolchains, as well as tailorability for EDDI runtime components integrated onto MRS applications.

For the EDDI toolchain-related activities, we discuss how the existing ODE tool adapter has been extended and used to support SESAME EDDI tailorability across the development toolchains. Further details on the associated tooling can be found in SESAME deliverable D4.4.

For the EDDI runtime component integration aspects, we focus our attention on supporting EDDI integration with MRS applications using the ROS platform. We discuss challenges and existing options for integrating with ROS architectures, highlighting their limitations. We further explain how our proposed approach is a comparative improvement. Our approach involves using the MROS framework to evaluate an existing ROS architecture, and then relevant EDDI-runtime-component configuration files can be generated using corresponding tooling we have developed. Additional details on the associated tooling can be found in SESAME deliverables D7.2 and D8.3.

As the SESAME project continues, we intend to further extend our approach to improve EDDI tailorability for additional platforms found in SESAME use cases.

REFERENCES

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and N. A., "ROS: an open-source Robot Operating System," *ICRA workshop on open source software*, vol. 3, no. 3.2, p. 5, 2009.