



Technological University of Denmark(DTU)
Informatics and Mathematical Modelling (IMM)

System Specification - Revised
02162 - Software Engineering 2
December 21, 2007

Group 1 participants:

Mathias Holm Andersen, s052761
Jonas Hansen, s052905
Bo Puggaard Hansen, s050759
Martin Ancher Müller, s052718
Anders Pedersen, s052690
Michael Sørensen, s052683
Thomas Lyngby Vestergaard, s052551
Peter Wind, s052425

Contents

1	Introduction	4
1.1	Conventions	4
1.2	Audience	4
2	Overall description	5
3	System features	7
3.1	Component editor	8
3.2	Deployment editor	9
3.3	Simulator	10
3.4	Dashboard	11
3.5	Interaction editor	12
4	Non-functional requirements	13
4.1	Implementation constrains	13
4.2	Documentation	13
4.3	Quality	13
5	User interface	15
5.1	Technologies	15
5.2	GUI parts	15
5.3	Handbook	19
6	Architecture	27
6.1	Deployment model	27
6.2	Deployment editor	30
6.3	Simulation	31
7	Glossary	35
	Appendix	36
A	Use cases	36
A.1	Insert component	36
A.2	Insert bus	37
A.3	Insert ECU	38
A.4	Insert connection	39

A.5 Initialize simulation	40
A.6 Delete component/ECU/connection	41
B UML class diagram - Complete	42

1 Introduction

This report documents the requirements of the software system which is being developed in 02162 Software engineering 2.

The software system is intended to improve the development of embedded systems by supplying developers with a CASE (Computer-Aided Software Engineering) tool for designing, modeling and simulating embedded systems.

This is done by allowing the developer to create and visualize the embedded system and test it by running a simulation of the embedded system design.

This could reduce cost for a company that designs and produces embedded systems.

As an example the system could be used in the automotive industry to design and test the many embedded systems existing in automobiles today. Likewise the system could also be used in aeronautics or generally in the transportation manufacturer industry.

1.1 Conventions

The priority of the requirements specified in section 3 and 4 will be indicated by keyword: **shall**, **should** and **would be nice**.

- **Shall** — Requirements that must be implemented as a minimum, resources must be allocated to these requirements.
- **Should** — Requirements that adds to the software systems functionality but not required for it to work, if possible resource should be allocated.
- **Would be nice** — Requirements which are considered ideas and will be implemented if sufficient resources exists, alternatively they could be done in later versions of the software.

These words will be emphasized using **bold** format. It is only when this format is used that the keywords have the meanings indicated above. All classes from the domain will be presented in *italic*.

1.2 Audience

The intended readers of this document are persons affiliated with companys that develop embedded systems.

2 Overall description

The overall description contains a brief introduction to the software systems and the subsystems it consists of, identification of the actors and description on how the software system works.

The actors of the software system has been indentified as:

- Developer: An actor who is involved in every part of the development process of the embedded system.
Example: An engineer employed at a company that produces embedded systems for cars.
- End user: An actor to whom the final embedded system is presented or an actor who presents embedded systems for others.
Example: A car manufacturer interested in buying the embedded system developed by a developer, or salespersonal from the devloping company

Within the scope of this project, an embedded system consists of a number of components, which can be categorized in three types:

- Software component: A component capable of evaluating input messages through a program and based on this create one or more output messages.
- Actuator: A hardware component capable of carrying out a physical task, triggered by input messages.
- Sensor: A hardware component capable of detecting a specific physical property, and sending messages based on this property.

Components can be combined using connections and a combination of components is called a deployment.

This software system allows developers to design/model these components in a component editor and combine them in a deployment editor. The deployment can then be tested using a dashboard, which is a visualization of a how a deployment would act in an actual embedded system, and it allows the actor to manipulate the deployment. To interact with the simulation, the developers can use the interaction editor.

Both, the dashboard and the interaction editor, are based on an underlying simulator. The simulator contains an algorithm which simulates the message flow between components, and based on these messages, the states of the components are changed as specified in their respective automaton.

The different subsystems and their associations can be seen in Figure 1.

The complete software system consisting of five subsystems will be explained in details in section 3.

The example in Figure 2 illustrates how the software system could be used to develop and test an embedded system, in this case how wipers could turn on based on rain sensors.

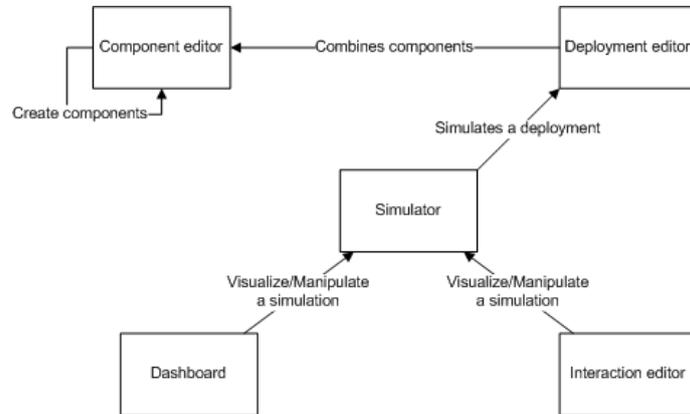


Figure 1: Association between the subsystems

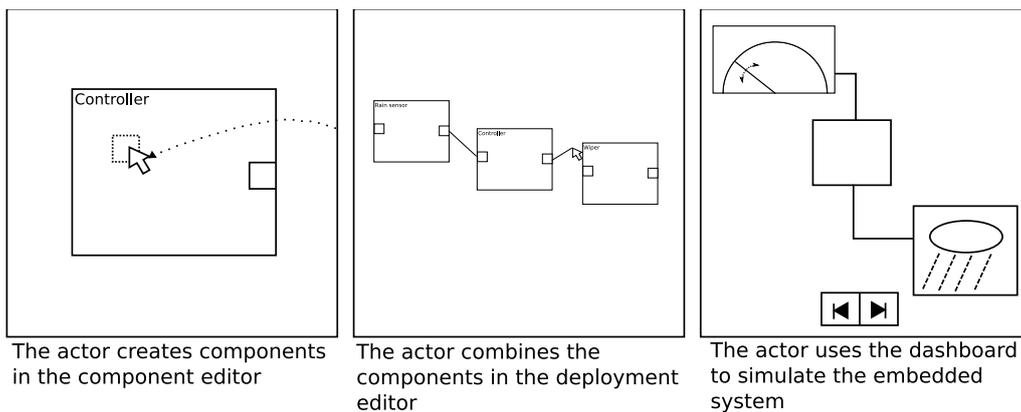


Figure 2: Illustration of how the software system could be used

3 System features

The primary use of the software, as mentioned in section 2, is to develop a model for an embedded system either for testing or presentation (or both). This primary use can be explained by use cases for each of the subsystems in the software system.

An overview of the use cases can be seen in Figure 3. The figure illustrates the top-level use cases of all the different sub-systems as well as the association between the actors and the system.

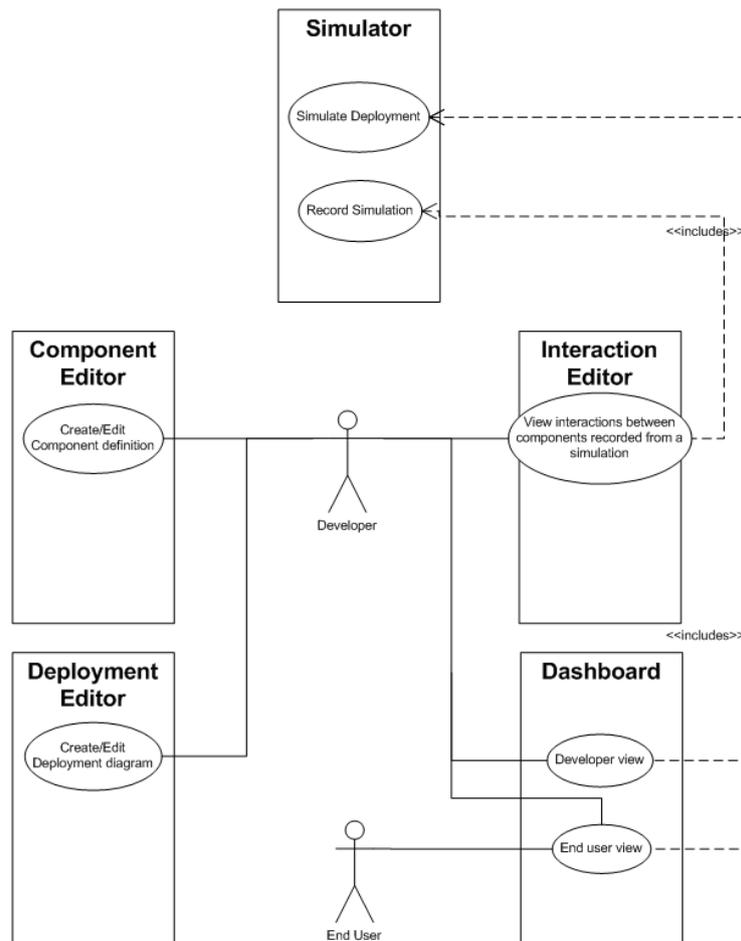


Figure 3: Overall use case diagram

The complete software system is developed by four teams, and the responsibilities for this team is the deployment editor and the simulator. Thus, only use cases referring to the subsystem “deployment editor” has been made, to cover the responsibility of this teams’ part of the project. The “simulator” has no direct user interactions and thus no use cases exists. Other use cases can be found in the appropriate project teams’ requirements specifications.

The purpose of sections 3.1-3.5 are to provide a detailed description of the software system. This includes stipulating the requirements for the subsystems.

3.1 Component editor

The component editor is a graphical tool which enables the user to define and edit component types through a "drag and drop" interface. The behavior of a component is based on an automaton, which is created within the component editor. The component editor has the following requirements.

3.1.1 Functional requirements

1. A component **shall** be one of either:
 - (a) Software component — *A component capable of evaluating input messages through a program and based on this create one or more output messages.*
 - (b) Actuator — *A component capable of carrying out a physical task, triggered by input messages.*
 - (c) Sensor — *A component capable of detecting a specific physical property, and send a message based on this property.*
2. A component **shall** have a unique name.
3. A component **shall** be able to have an arbitrary number of ports attached — *A component must be able to communicate with other components. In order not to restrict the possibilities of the components the number of ports must not be limited.*
 - (a) A port **shall** have a unique name within the component — *Helps the user identify the purpose of a port.*
 - (b) Each port **shall** have an input buffer with capacity changable trough the deployment editor by a developer — *Enables the definition of a port to limit the number of buffered messages.*
 - (c) Components **shall** be able to send and receive messages through ports attached to the component — *Communication is based on messages delivered from one port to another port.*
 - i. Each port **shall** have a finite set of message types it is allowed to send and a finite set of message types it is allowed to receive — *Enables compatibility-check between ports.*
 - ii. A message **shall** be defined by a text string.
4. The internal behavior of a component **shall** be defined using an automaton — *Component behavior is easily defined using an automaton.*
 - (a) Transitions **shall** be triggered either by:
 - i. An incomming message.,
 - ii. A spurious trigger, which automatically changes the state of a component wihtout any input.
 - (b) A Transition **shall** be able to cause the component to send multiple messages — *Enables a component to communicate a change of state through one or more ports.*
 - (c) An automaton **shall** have an initial state — *By definition, an automaton must have an initial state.*
 - (d) An automaton **shall** be able to be non-deterministic if several transitions are possible
5. It **shall** be possible to store and change the defined components for future use.

3.2 Deployment editor

The deployment editor is the part of the software system where deployment diagrams are created. A deployment diagram combines components created in the component editor and thereby modeling an embedded system for which a simulation can be initialized. As an example this could be an automotive embedded system which starts and stops the wipers of a car depending on whether it rains or not. Such a system would consist of a number of components like rain sensors, software components and wiper actuators.

3.2.1 Functional requirements

1. A developer **shall** be able to design a complex embedded system through a graphical editor by combining components using connections and busses — *The use of a graphical editor is intended to improve the general usability of the software.*
2. Components in a deployment *shall* be instances of components designed in the component editor.
3. Components in a deployment *shall* have port instances matching the ports defined in the component definition.
4. An ECU(Electronic Computational Unit) **shall** act as a container for software components — *A software component can be seen as a computer program and needs a host to be executed on.*
5. A connection **shall** connect two compatible ports — *Two ports are compatible if the set of acceptable messages for the sending port is included in the set of acceptable messages for the receiving port.*
6. A connection **shall** be attached to a bus if the connection is not between two ports within the same ECU — *In an embedded system busses enable communication between different physical devices.*
7. A bus **shall** have a buffer — *A buffer represents a buffers memory.*
 - (a) A user **shall** be able to determine the size of the buffer when the bus is inserted or edited. — *It should be possible to test an embedded system with variable buffer size. The developer should be able to deal with bottlenecks by supplying larger buffers.*
8. There **shall** apply some constraints to the combination of components, ECUs, connections and busses.
 - (a) A software component **shall** only exist within an ECU — *A software component needs to be run on hardware and it is this hardware that is simulated by the ECU*
 - (b) A sensor or an actuator **shall** never exist within an ECU — *Sensors and actuators do not use computational power.*
9. It **shall** be possible to start a simulation on a deployment directly from the editor.
10. It **should** be possible to use "drag and drop" in the graphical editor to combine components.
11. It **should** be possible to "undo/redo" operations made by the user in the graphical editor.

12. It **would be nice** to be able to change the graphical appearance of components in the editor — *To make it easier for a developer to identify the purpose of components.*

3.2.2 Use cases

The uses of the deployment editor is summed up in the use case diagram in Figure 4.

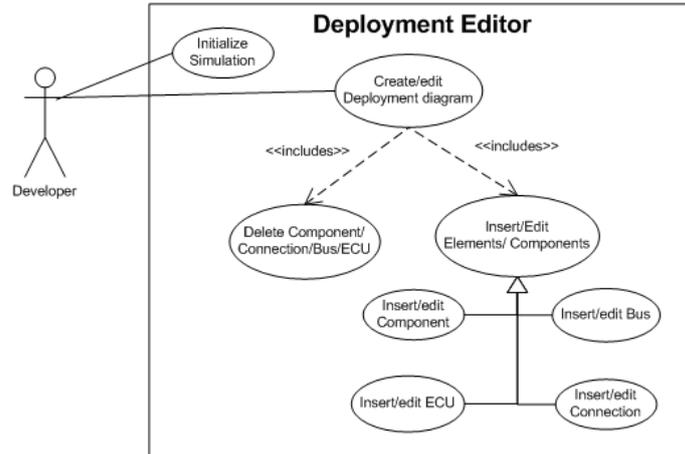


Figure 4: Deployment editor - Use case diagram

Each of the different use cases are described details in Appendix A.

3.3 Simulator

The simulator is the part of the software system where the embedded system is simulated. It simulates a deployment created in the deployment editor, and it does not have a graphical interface but rely on the dashboard and interaction editor for visualization.

3.3.1 Functional requirements

1. A simulation **shall** be able to Simulate a step forward in time. A step is defined in the simulation algorithm which can be seen in Figure 5. Concepts from the domain model has been inserted where relevant, a detailed description of the domain model is in Section 6.3.
2. It **would be nice** if the simulation supports the following operations;
 - (a) Record all simulated steps — *To allow the user to view the history of the simulation.*
 - (b) Go back and forth stepwise in a recorded simulation — *For example if the components at some point get into inconsistent states, the user will want to find the exact step where the inconsistency occurred.*
 - (c) Manual method of changing states in sensors, and thereby cause a reaction by the complete embedded system — *Lets the user test different scenarios for the embedde system.*

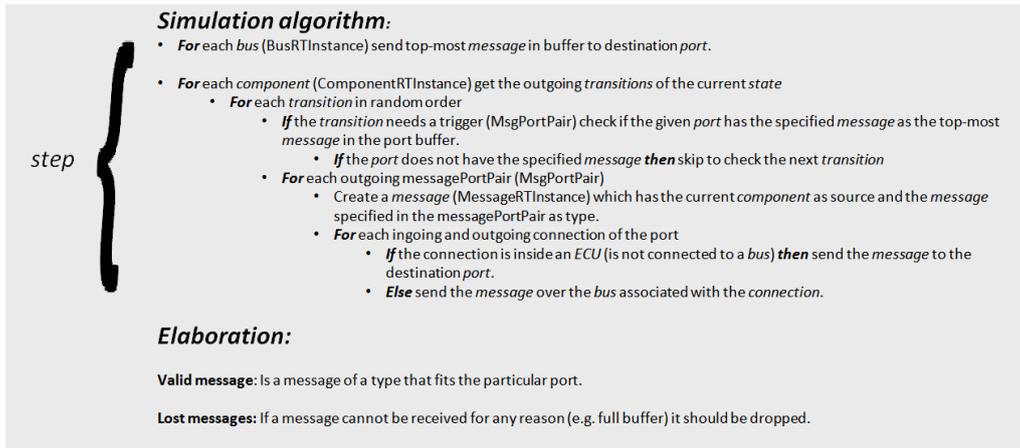


Figure 5: Sketch of simulation algorithm

- (d) Randomized method of changing states in sensors, and thereby cause a reaction by the complete embedded system — *So the simulation can run without user input, and possible try sensor input combinations the developer would not have thought off.*

3.4 Dashboard

The dashboard is the part of the software system where the simulation is visualized. The dashboard has two main user groups, the developers and the end user. The developer could use the dashboard for debugging the embedded system. The end user could use the dashboard to present the embedded system or be presented an example of the embedded system using the dashboard.

3.4.1 Functional requirements

1. The dashboard **shall** use a simulation as an underlying model
2. The dashboard **shall** have a graphical user interface (GUI).
3. The dashboard **should** include illustrations for each component type.
4. The dashboard **should** allow the user to activate/deactivate sensors.
5. The dashboard **should** have different detail modes.
 - (a) All details. — *This mode gives the developer access to all information given by the simulation, and the ability to set any state of any component as desired including deleting and adding specific messages from busses*
 - (b) Low detail. — *Only allows the user to switch states of sensors, and see states of actuators*
6. *It would be nice* if the dashboard included animations for actuators as opposed to static illustrations. — *Animations makes it easier to see that the actuators are in motion.*

7. The dashboard **should** be able to change illustrations for component types. — *e.g. if a given illustration or animation gives a better understanding of the component in a specific situation, it would be nice to be able to change one or more from within the dashboard*

3.5 Interaction editor

The interaction editor is a part of the software system where the interactions between components are illustrated. The interactions can either be recorded from a simulation or entered manually. Only the interactions described in the deployment will be allowed to be entered manually. This could for example be used by a developer to see how the components interact in order to track down errors.

3.5.1 Functional requirements

1. The interaction editor **shall** be able to create an interaction diagram based on either:
 - (a) The interactions recorded in a simulation — *Will be useful for getting a quick overview of the recorded simulation*
 - (b) The interactions specified manually — *So the developer has better control over what the diagram shows. For example modifying a diagram to illustrate a point.*
2. It **would be nice** to have the option to export the diagram to another format. For example a picture format.

4 Non-functional requirements

To clarify requirements not related to the actual programming of the software system, a number of requirements are identified in the following subsections.

4.1 Implementation constraints

The software **shall** be implemented as a plug-in to the Eclipse framework using Eclipse build in features (eg. PDE, SWT and JFace). It is required that it is based on the following three technologies:

- EMF (Eclipse Modeling Framework) v. 2.3.0
- GEF (Graphical Editing Framework) v. 3.2.101
- GMF (Graphical Modeling Framework) v. 2.0.1

4.2 Documentation

The development **shall** be documented by meeting the following deliveries:

- Requirements specification by week 41
- System specification by week 44
- Prototype by week 46
- Prototype integration by week 48
- Final software delivery by week 51, including
 - Test documentation
 - Revised system specification
 - User manual

The deliveries **should** all comply with the specific demands of each delivery as practiced in Software Engineering 1¹ and Software Engineering 2² at DTU.

4.3 Quality

The quality insurance procedures of this project will be limited to two main areas:

- The insurance of quality in documentation
- The insurance of quality of implementation

1. It is intended to ensure documentation quality by imposing the measures listed below:

¹<http://www.kurser.dtu.dk/presentation.aspx?menulanguage=dk&coursecode=02161-5>

²<http://www.kurser.dtu.dk/presentation.aspx?menulanguage=dk&coursecode=02162-3>

- (a) Every document *shall* be approved by the delivery leader as well as the team leader.
 - (b) Any part of a document are produced by at least two persons, thus trying to decrease the amount of typical errors as well as improving the creative process.
 - (c) Each delivery is reviewed after feedback and appropriate measures are taken and documented, so that errors are caught as early as possible in the development.
2. It is intended to ensure quality of the implementation by imposing the measures listed below:
- (a) A quality manager is appointed to enforce that the policies for the data repository are upheld. These policies should be able to reduce the amount of untested and/or non-working code which is committed, as well as preventing that no binary code files are committed.
 - (b) All implementations are done in pairs of developers. The intention of this is to decrease the number of typical errors that often follows when using new technology.
 - (c) The implementation is divided into a number of milestones, which number is yet to be determined. After each milestone the quality of the implementation is reviewed and decisions for possible improvements are discussed and if necessary implemented.
 - (d) A testing manager is appointed to develop a number of rules which all tests **should** comply with, and he will try to enforce that these rules are upheld.
 - (e) An optimal test strategy **should** be developed for maximizing the numbers of discovered errors before the software is released. The general test strategy will at least consists of an acceptance test combined with continuous unit testing of manually implemented methods during the development. It is the intentio that all manually implemented code regarding the model will have a statement coverage of 100%.
 - (f) The commenting of all manually implemented Java code shall use the javadoc standard.³

³<http://java.sun.com/j2se/javadoc/>

5 User interface

The following section will describe the different parts which the Deployment Editor's graphical user interface (GUI) consists of and how this should look like. Furthermore it will be described how to solve important tasks within this GUI. As mentioned earlier, the Deployment Editor is used for designing an embedded system. This shall of course be as straight forward as possible to do and our tool should eliminate much of the errors which could occur when developing this embedded system. Therefore it is important that the GUI will be thought through and made easy and simple to use. To make sure that the software system is easy to use it a usability test should be made - however this will go under the *nice to have* category.

5.1 Technologies

The GUI shall be implemented as an Eclipse⁴ Plugin which gives us access to all the functionality in Eclipse. Details on Eclipse Plugins are referred to the Seminar Paper on Eclipse⁵. The different parts of the Deployment Editor will be implemented in different ways which all are part of an Eclipse perspective, which are listed below.

- **Editor:** Used for resource editing and creation. (Will be explained in section 5.2.2 Resource Editor)
- **View:** Elements that are used primarily for showing static information relating to the current editor resource. (See Section 5.2.1, 5.2.3, 5.2.4 Project Explorer, Resource Properties, Diagram Overview)
- **Dialog:** Like a pop-up window where you can set the properties of the plug-in behavior or edit the properties of a resource. (See Section 5.3 Handbook for examples)
- **Wizard:** Opens in a pop-up window as well. A sequence of pages which have to be filled out correctly before the actor can go on. Used when creating or editing resources. Examples are shown in the handbook, (Section 5.3 Handbook for examples)

5.2 GUI parts

This section will cover the different parts in the Deployment Editor and describe the function of each. The Deployment Editor consists of five main parts which are shown in Figure 6:

The Project Explorer, Resource Editor and the Resource Properties are the three most important areas because those areas are responsible for most of the interaction with the actor of the software. A description of the different parts will be provided in the following sections. These will describe the function of the part and what the actor shall be able to do. A description of how the different actions are done is given in Section 5.3.

⁴<http://www.eclipse.org/>

⁵Seminar Paper on Eclipse, by Bo Puggaard Hansen and Mikael Andersen

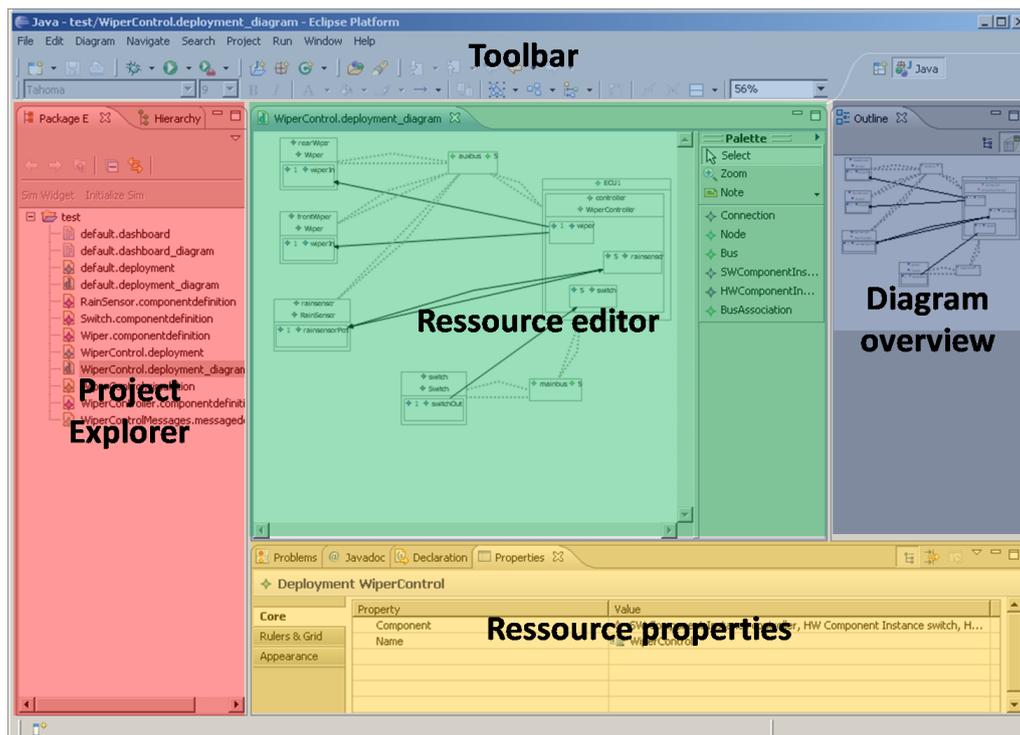


Figure 6: Overview of the Deployment Editor

5.2.1 Project Explorer

The Project Explorer will be implemented as a view, and will function as an overview of the project and all its associated files are shown. Not just files for the Deployment Editor are shown, but all the Components and Component Diagrams are listed too. By selecting a file, the program should determine which type of file we are dealing with and open the editor associated with the certain type. (See Figure 7).

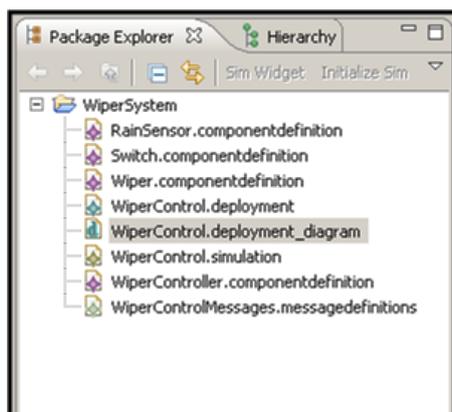


Figure 7: An example on how the Project Explorer could look like in our project

In our final implementation it would be nice to have different icons depending on which type of file we are dealing with. This would make it easier to recognize the different parts of our project.

5.2.2 Resource Editor

As the name says, this will be an editor. Here you will see a graphical representation of the Deployment Diagram, and the actor shall be able to select the different components which are inserted in the diagram. (See Figure 8).

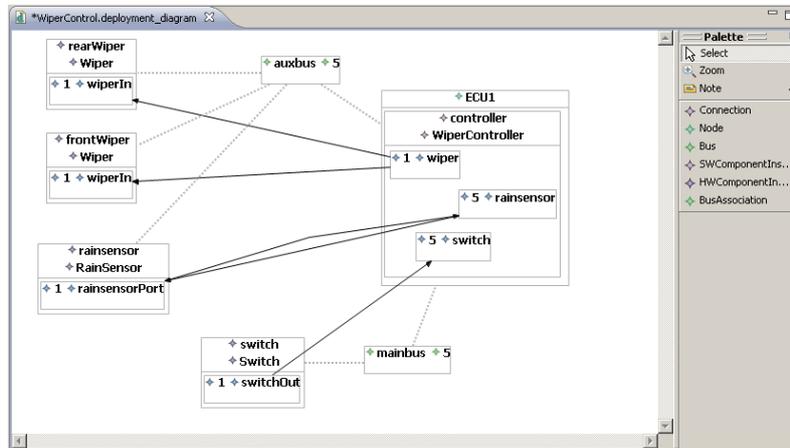


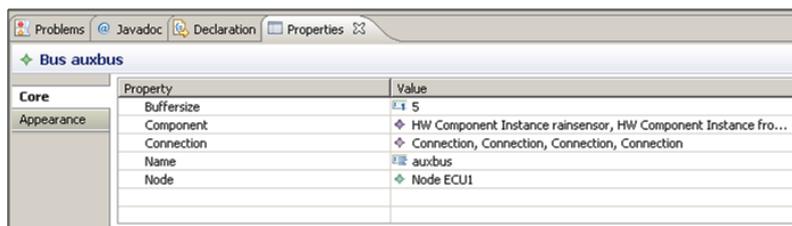
Figure 8: An example of a resource editor.

The Resource Editor is split into two parts (The left and right side on the sketch above). The left area is where the graphical representation of the Deployment Diagram is shown. The actor shall be able to select a component and move it around using the drag and drop technique and more generally viewing the diagram and manipulate it. When moving components which has one or more ports associated, the port will automatically be moved together with the component. The right area (Called Palette on the figure) contains two things: Tools and list of the component types to insert - both represented as buttons. The tools is used by the actor to manipulate the graphical representation of the Deployment Diagram. This includes zooming, selecting or a marquee-function which allows the actor to select multiple elements in the editor at once. As for the list of components these will be listed by looking at what is created in the Component Definition Editor. The actor shall then be able to insert the listed components into the left area of the editor. (See Handbook, Section 5.3). If it increases overview or usability, sets of tools can be grouped in submenus of the palette.

5.2.3 Resource Properties

This will be implemented as a view. The view is connected to the Resource Editor, and the content will be different depending on what type of element is selected in the Resource Editor. The main use of the Resource Properties is, as the name says, to edit the properties of the different components/elements in the diagram. An example below could occur if a bus has been selected (See Figure 9):

It shall then be possible for the actor to edit such information as above for each element in the Deployment Diagram. Properties can be read only and in that case, will only be there for information. If properties change the structure of the Deployment, the change will be shown in the diagram frame.



Property	Value
Buffersize	5
Component	HW Component Instance rainsensor, HW Component Instance fro...
Connection	Connection, Connection, Connection, Connection
Name	auxbus
Node	Node ECU1

Figure 9: Sketch of how the Resource Properties could look like when a bus is selected.

5.2.4 Diagram Overview

This is implemented as a view, and does not have an actually function. If the Deployment Diagram reaches a size greater than the editor-frame, the overview will show a box symbolising what part of the diagram is visible and a miniature image of the entire diagram. By dragging the box, the view in the diagram frame will be moved accordingly. (See Figure 10).

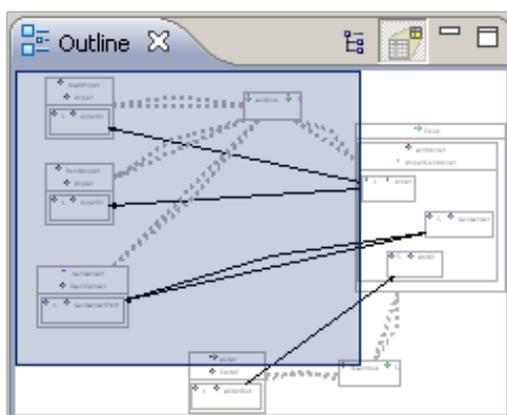


Figure 10: An Example of the Diagram Overview.

5.2.5 Toolbar

The toolbar is the place where all the general functions are placed. This includes commands like creating, saving and loading. We have also chosen to put the undo/redo-functions and things like starting the simulator from a given deployment diagram. (See Figure 11)



Figure 11: The toolbar could look like this.

5.2.6 Right-click menu

This is not shown on figure 6. If an item in the diagram has relevant options, these can be placed in a menu which is accessible by right-clicking the item. This could be the option to delete the item or start some action, for which the properties pane is not suited. Content of the menu will therefore be depending on the type of component is selected (see Figure 12).

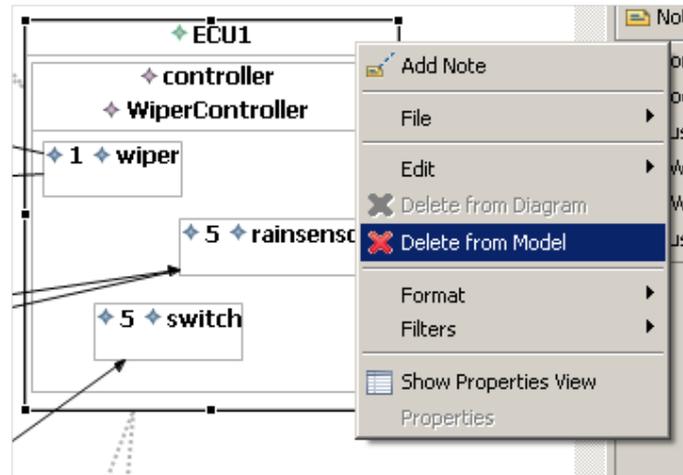


Figure 12: Right-click menu

5.3 Handbook

The hand book is intended to present the product (software) from the actors point of view. It has been made in the way of a tutorial. This means that it describes how to solve a number of the most important tasks in the application in a step-by-step way supplied with screen shots, so that the actor can follow the tutorial while working hands on the application.

5.3.1 How do I create a new deployment diagram?

Right click on the project in the Project Explorer and select New > Other (see Figure 13)

A wizard will now appear. Select CASETool > CASETool Deployment Diagram (group 1) and press Next. (See Figure 14)

Now specify the Parent Folder and File Name and press Finish. (See Figure 15)

An empty deployment diagram with your chosen name will now be shown in the editor.

5.3.2 How do I insert a Node/ECU into the deployment diagram?

Left-click on Node in the Palette. Point the cursor to the place in the deployment diagram where you want to place the node and left-click again. Now you have to specify a name for the new Node. This name should be unique within a deployment (see Figure 16).

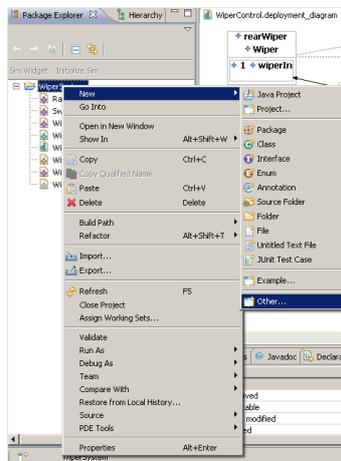


Figure 13: Right-click menu

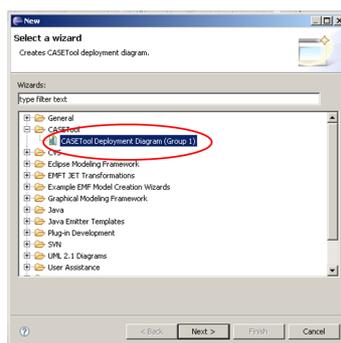


Figure 14: Wizard step 1



Figure 15: Wizard step 2

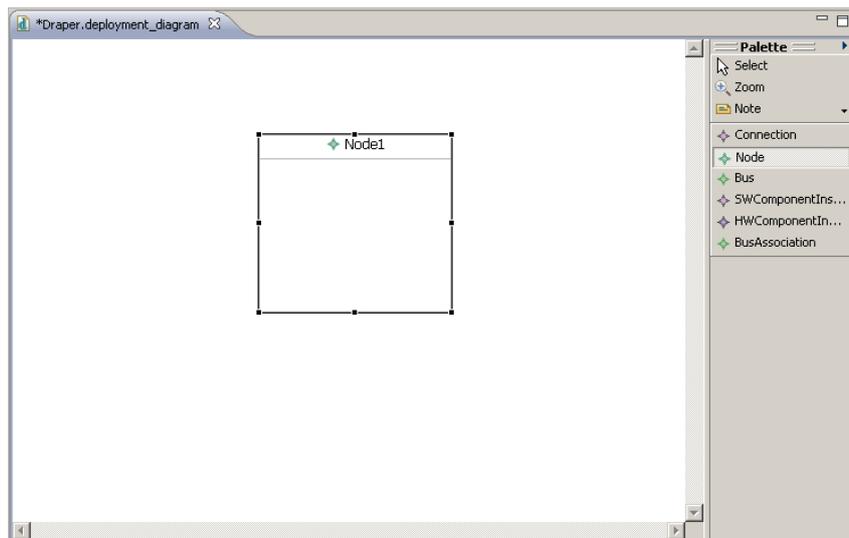


Figure 16: Insertion of a Node

5.3.3 How do I insert a hardware component into the deployment diagram?

Left-click on Hardware Component in the palette. Point the cursor to the place in the deployment diagram where you want to place the hardware component and left-click again. Now you have option to specify a name for the hardware component. This name should be unique within a deployment.

5.3.4 How do I insert a software component into the deployment diagram?

A software component should be placed inside a Node. Left-click on Software Component in the palette. Point the cursor to the Node in the deployment diagram that you want to place the software component inside and left-click again. Now you have to specify a name for the new software component. This name should be unique within a deployment.

5.3.5 How do I specify which component definition a component is an instance of?

Left-click on a component in the editor. In the properties view is located a property called definition. Change this property to the desired component definition. Hardware components can only select hardware definitions and software components can only select software definitions. When a new type has been selected the component creates instances of all the ports specified in the definition (see Figure 17).

5.3.6 How do I insert a bus into the deployment diagram?

Left-click on Bus in the palette. Point the cursor to the place in the deployment diagram where you want to place the bus and left-click again. Now you have to specify a name for the new bus. This name should be unique within a deployment.

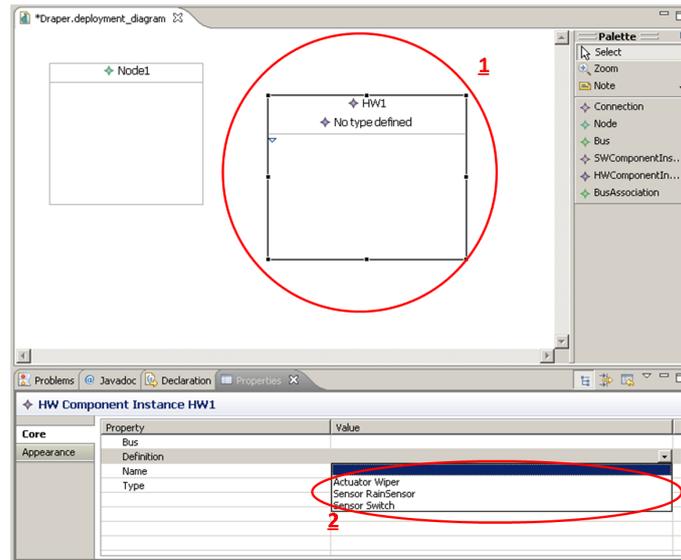


Figure 17: Change component definition

5.3.7 How do I associate a component with a bus?

Left-click on the bus association item in the palette. If the component is a hardware component, drag a connection from the hardware component to the bus. If the component is a software component, drag a connection from the node containing the component, to the bus.

5.3.8 How do I insert a connection between ports?

Inside a component ports are represented as squares. Left-click on the Connection in the palette. Click and hold down the mouse button as you drag the connection from one port to another. If the two ports are compatible a connection is created. As long as the connection is not associated with a bus, it is shown with a red color.

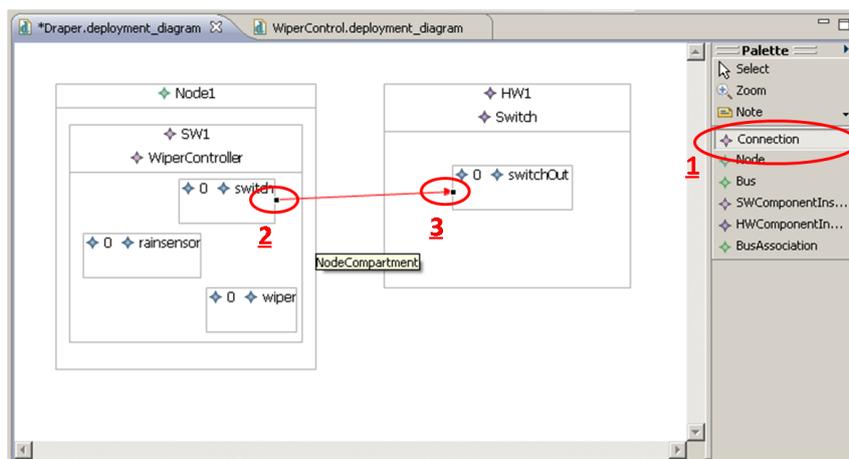


Figure 18: Insert a connection

5.3.9 How do I associate a connection with a bus?

Left-click on a connection and change the bus property in the properties view. If no busses are shown in the list it is because the components of the ports are not associated with a common bus.

5.3.10 How do I move an object in the deployment diagram?

This is done by dragging the object to the place where you wish put it.

5.3.11 How do I delete a Node/ECU?

Mark the Node in the deployment diagram and press Delete. Deleting a Node will also result in deletion of the software components inside the Node and their connections.

5.3.12 How do I delete an actuator, sensor or software component?

Mark the actuator or sensor in the deployment diagram and press delete. Deleting an actuator, sensor or software component will also result in deletion of all the connections related to this component.

5.3.13 How do I delete a bus?

Mark the bus in the deployment diagram and press Delete.

5.3.14 How do I delete a connection?

Mark the connection in the deployment diagram and press Delete.

5.3.15 How do I change the buffer size of a bus?

Mark the bus (in the deployment diagram) for which you want to change the buffer size. Now look in the Properties window and find Buffer size. Here you can specify the Value you wish (see Figure 19).

5.3.16 How do I change the buffer size of a port?

Mark the port (in the deployment diagram) for which you want to change the buffer size. Now look in the Properties window and find Buffer size. Here you can specify the Value you wish.

5.3.17 How do I change the source or target port of a connection?

Select the line in the Editor representing the connection. Now drag the end of the connection you want to change to the port you want it changed to.

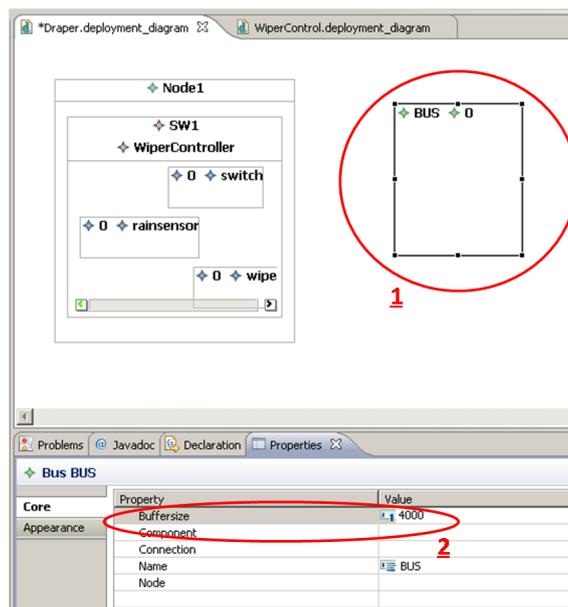


Figure 19: Change buffer size

5.3.18 How do I rename a component, ECU, bus or connection?

This can be done in two ways:

- Either you change the label in the Editor
- Or you mark the component and then changes the name in the Properties window

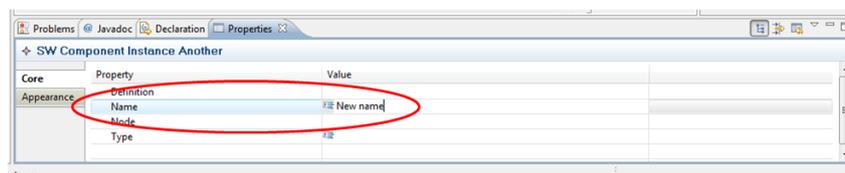


Figure 20: Edit name of in the Property window

5.3.19 How do I undo/redo a change?

In the toolbar there are located buttons for undoing and redoing the latest changes. Alternatively these keyboard shortcuts can be used: Undo (Ctrl+Z) and Redo (Ctrl+Y) (see Figure 21).

5.3.20 How do I save a deployment diagram?

Press the Save in the toolbar.

5.3.21 How do I load a deployment diagram?

In the Project Explorer find the deployment diagram-file you wish to open and double-click on it. The Editor, Properties, and Overview should now open.

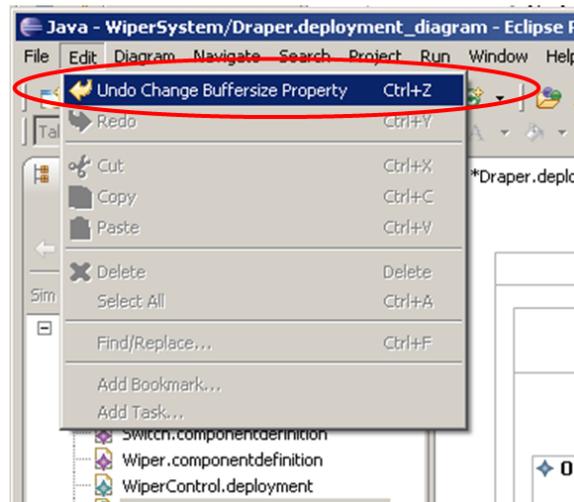


Figure 21: Undo changes

5.3.22 How do I initialize a simulation from a deployment?

Left-click on the deployment_diagram file in the project explorer and click on the 'Initialize Sim' button (see Figure 22).

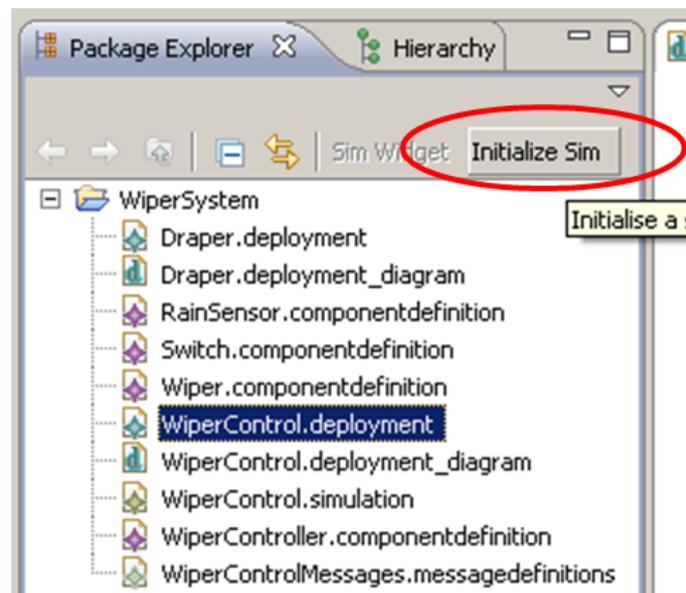


Figure 22: Initialize a simulation

5.3.23 How do I start a simulation of a deployment?

Left-click on the .simulation file in the project explorer and click on the 'Sim widget' button (see Figure 23). The widget opens and consists of a set of buttons which controls the simulation (see Figure 24).

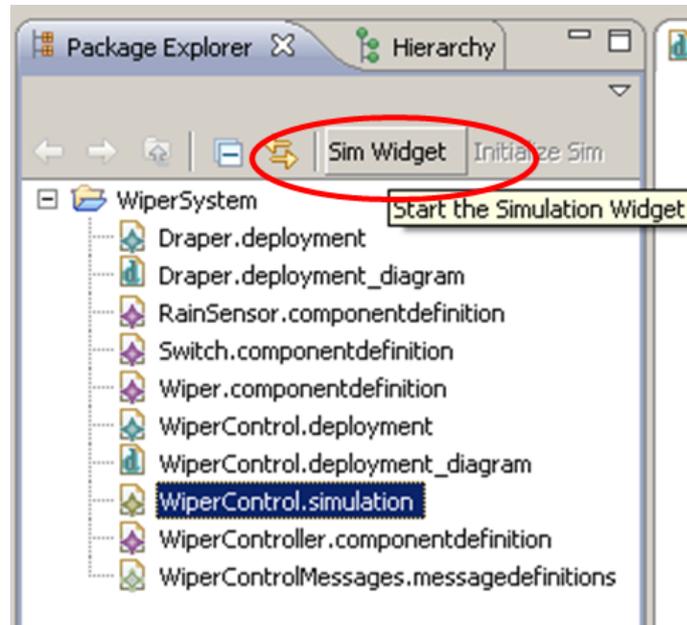


Figure 23: Start the simulation widget



Figure 24: Simulation widget

6 Architecture

The architecture is an overview of the components⁶ in the software. The architecture is first described from a high level perspective with all the components of the software and their interfaces. Each component will then be described in detail.

The architecture of the software system can be seen as eight cooperating components. The eight components and their interfaces can be seen in Figure 25. The interfaces between elements have been given the name of the component that gives the interface to the other component. Packages above the components gives an identification of where the component “belongs” in the software system.

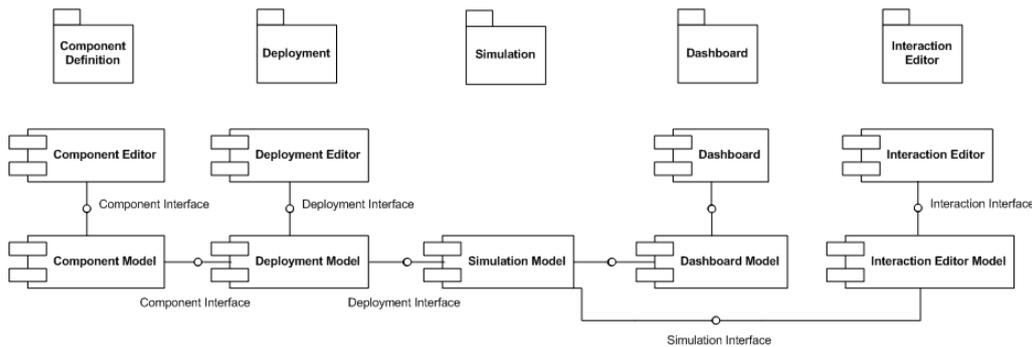


Figure 25: Overall component diagram

The description of the architecture is divided into three subsections:

- Deployment model - A detailed description of the model.
- Deployment editor - An extension of the chapter on the graphical user interface 5.
- Simulation - A detailed description of the *Simulation*.

The sections regarding the deployment model and the simulation contains and explains a subset of the complete UML class diagram. The complete UML class diagram in a whole can be seen in Appendix B.

6.1 Deployment model

Each section will describe, in more technical terms, how this component works. For each class in the components there will be a description of

- What does the class represent
- How is the component associated to other classes in the deployment
- What attributes does the class have
- What overall methods must the class have

Figure 26 shows the classes in the deployment model and their associations.

⁶In terms of a UML component diagram

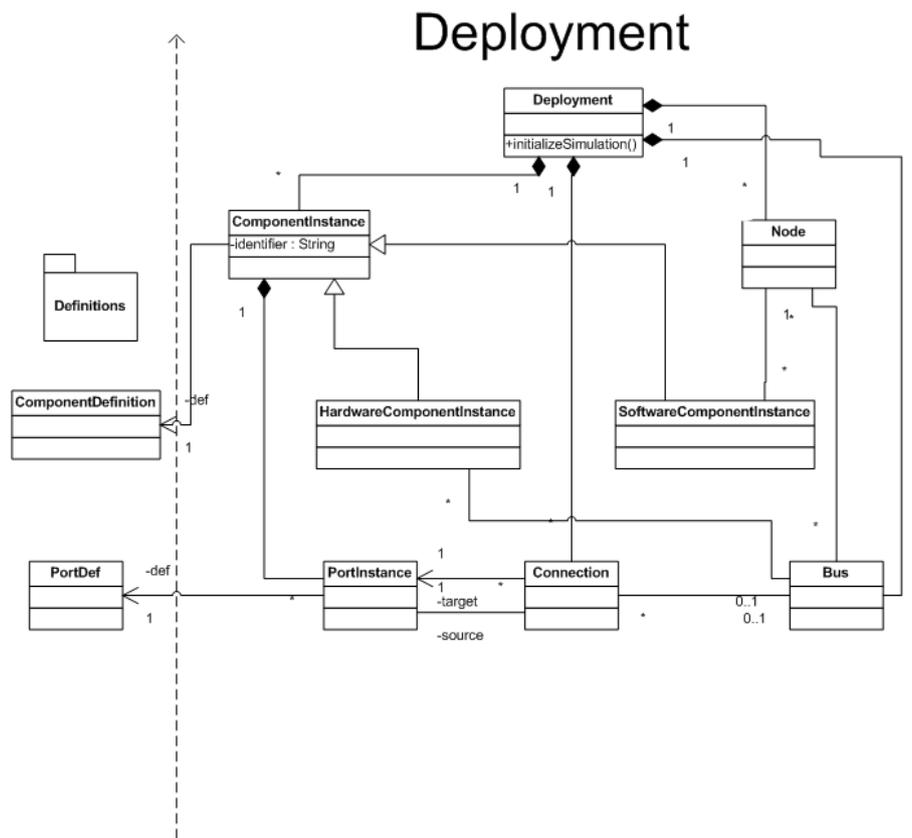


Figure 26: UML diagram of deployment editor

6.1.1 Deployment classes

The following describes the classes of the deployment model followed by a description of how the implementation is intended.

Deployment class A *Deployment* is an aggregation of four sets of objects of the types:

- component : *ComponentInstance*
- node : *Node*
- connection : *Connection*
- bus : *Bus*

A *Deployment* object represents an actual deployment of embedded components.

The *Deployment* has a name, to ease reference when making more instances of the same *Deployment*.

To initialize a simulation of a given *Deployment*, the *initSimulation* method is provided.

ComponentInstance abstract class The *ComponentInstance* represents a concrete embedded component in a given deployment. A *ComponentInstance* consists of a set of *PortInstances* to enable communication with other deployed *ComponentInstances*.

A *ComponentInstance* is associated with a *ComponentDefinition*, since no *ComponentInstance* can exist without being in a *Deployment*.

Because several *ComponentInstances* of same type can occur in the same *Deployment*, a unique name, enables reference to a specific *ComponentInstance*.

Since the *ComponentInstance* is abstract, any embedded component must be represented either as a *HWComponentInstance* or a *SWComponentInstance*.

HWComponentInstance class The *HWComponentInstance* represents a type of embedded component, which will be either a *sensor* or an *actuator*. To enable communication to other devices, the *HWComponent* is associated to a number of *Busses*.

SWComponentInstance class The *SWComponentInstance* represents a type of embedded component on a *Node*, capable of doing some computation based on input messages. A *SWComponentInstance* cannot exist without being part of a *Node*. The *Node* delivers the ability to execute computations.

Node class A *Node* represents an embedded source of computational power. A *Node* has associations to:

- A *Deployment*, since no *Node* can exist without being a part of a *Deployment*.
- A set of *SWComponentInstances*, for which the *Node* supply computations.
- A set of *Busses*, to enable the *Node* to communicate with other parts of the *Deployment*.

Every *Node* has a unique name to identify it from eventual other *Nodes* in the *Deployment*

Connection class The *Connection* resembles an embedded connection. A *Connection* can enable two *PortInstances* to exchange data.

A *Connection* has associations to:

- A *Deployment* which has the *Connection*, since a *Connection* is a part of the *Deployment*.
- A source and a target *PortInstance* which can send and receive messages through this connection.
- An optional *Bus*, which will act as a buffer for the flow of data through the *Connection*. This is not needed if the *Connection* is between two *PortInstances* within one *Node*

A *Connection* has no internal variables. A *Connection* can be identified by its source and target *PortInstance*.

PortInstance class The *PortInstance* represents the socket in which the *Connection* is "plugged" into the *ComponentInstance*. The *PortInstance* has association to:

- in: A set of ingoing *Connections* from which it can receive messages.
- out: A set of outgoing *Connections* to which messages can be sent.
- component: The *ComponentInstance*, the *PortInstance* is embedded on. A *PortInstance* is a part of a *ComponentInstance*
- definition: The *PortDefinition* describing the type of the *PortInstance*.

The *PortInstance* has a buffersize that limits the amount of messages which can wait in the buffer without risk of loosing messages. The *PortInstance* has a type that refers to a *PortDefinition*.

Bus class The *Bus* represents an embedded buffer which handles the transportation of messages. A *Bus* has associations to:

- node : A set of *Nodes* for which the *Bus* can transport messages.
- deployment : The *Deployment* which the *Bus* is a part of.
- connection : The set of *Connections* which can utilize the buffer of the *Bus*.
- component : A set of *ComponentInstances* because the *Bus* has to know where to send messages.

The *Bus* has a buffersize to limit the amount of messages the *Bus* can contain before a risk of loosing messages is present. The *Bus* has a name for identification, since a *Deployment* can have several *Busses*.

6.1.2 Deployment implementation

The following will describe what parts of the implementation will be handled by EMF, GMF and manual implementation.

Structure All classes in the above section will be generated by EMF. All data to represent the structure of the model will be present when generation is done.

Volatile methods and attributes EMF will leave all volatile methods as stubs for the development team to implement. The last paragraph for each of the classes in the above section describe volatile methods or attributes, which will have to be implemented by hand.

6.2 Deployment editor

The GUI part of the editor is described in detail in the 5.

The following describes in more details, more specific parts of the customization of the generated editor.

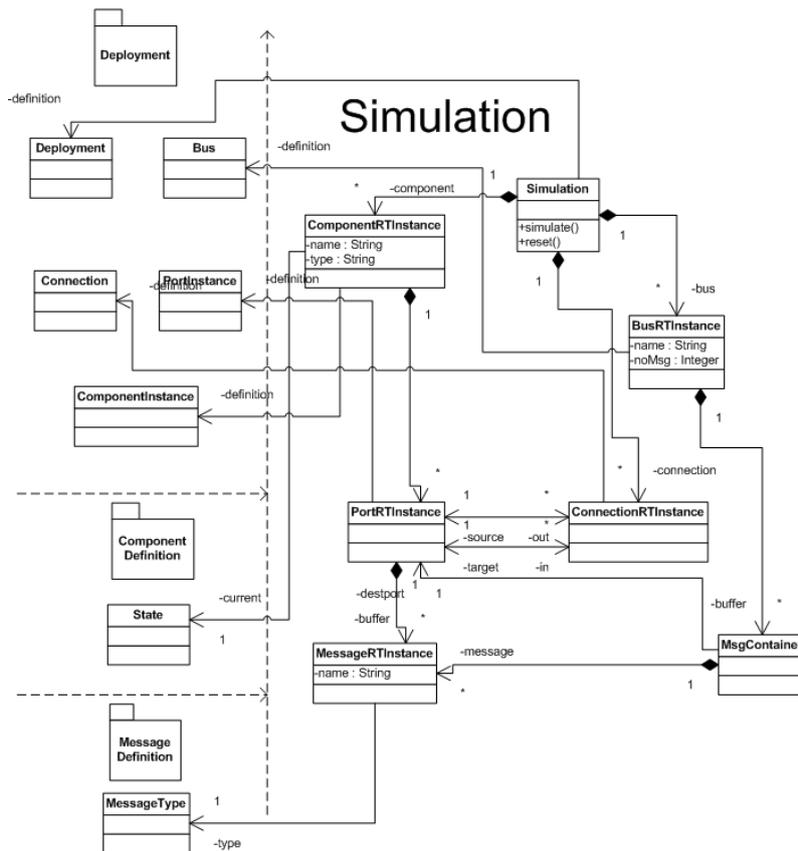


Figure 27: UML diagram describing the simulator

Port placement Since a *PortInstance* is the boundary for communication between *ComponentInstances*, a *PortInstance* should be graphically placed on the edge of its containing *ComponentInstance*.

This will be done using a custom *LayoutEditPolicy* in the *ComponentInstance*, which will enforce the position of contained *PortInstances*.

6.3 Simulation

The intention of this section is, in details, to describe how the simulator will be represented and implemented in the final software system. The section will not describe the requirements and functionalities of the simulator as these have already been covered in section 3.3.

The simulator, unlike the other parts of the software, only contains a single component which is the simulation model. This model consists of all the classes needed to simulate a deployment in run-time, and it provides an interface which other components can use to get access and manipulate with simulation data. As an example the dashboard relies heavily on this interface to be able to visualize a simulation.

The complete UML class diagram of the simulator can be seen in Figure 27, and each of the classes will be covered in the following subsections. A general note on almost all the classes are, that they are run-time instances of the objects in the deployment model. Thus there are many similarities in the way classes are connected with references i.e. When two objects are connected in within the deployment, they are also connected in the simulation.

This section will in the end describe what benefits the development will gain from using the EMF technology when implementing the simulator.

6.3.1 Simulation class

A *Simulation* object is an instance of main class of the simulation. The class has a single reference, deployment, to the *Deployment* class which it is intended to simulate.

The *Simulation* is a container for all the classes relevant for a simulation and is a direct container of the following:

- component : A set of *ComponentRTInstances*
- bus : A set of *BusRTInstances*
- connection : A set of *ConnectionRTInstances*

The intended purpose of the *Simulation* is to simulate a *Deployment*. It contains the two methods:

- *setDefinition()* : Initializes the simulation and creates runtime instances of all instances in the selected deployment.
- *simulate()* : Uses the simulation algorithm to simulate a step. This simulation algorithm is defined by the sketch in Figure 5 on page 11.
- *reset()* : Resets the *Simulation* by emptying all buffers in *BusRTInstances* and *PortRTInstances*, and sets the current state of all *ComponentsRTInstances* to their initial state.

6.3.2 ComponentRTInstance class

A *ComponentRTInstance* is intended to be a run-time instance of a *ComponentInstance*, therefore it has a single reference, definition, to the *ComponentInstance* and a single reference, simulation, to the *Simulation* in which it is contained.

The *ComponentRTInstance* is a container for a set of *PortRTInstances* for the same reason as the containment between *ComponentInstance* and *PortInstance* in deployment.

A reference, current, to *State* in the componentdefinition package is used to keep track of the state in which the *ComponentRTInstance* is.

ComponentRTInstance has two volatile attributes :

- name : Retrieves the name of the *ComponentInstance* which the *ComponentRTInstance* is a run-time instance of.
- type : Retrieves the type of the *ComponentInstance* which the *ComponentRTInstance* is a run-time instance of.

6.3.3 BusRTInstance class

A *BusRTInstance* is intended to be a run-time instance of a *Bus*, it has a single reference, definition, to the *Bus* and a single reference, simulation, to the *Simulation* in which it is contained.

The *BusRTInstance* is a container for a set of *MsgContainers* so that the *BusRTInstance* is able to represent the concept of a buffer with its messages.

BusRTInstance has two volatile attributes :

- name : Retrieves the name of the *Bus* which the *BusRTInstance* is a run-time instance of.
- noMsg : Retrieves the number of *MsgContainers* from the buffer of the *BusRTInstance*.

6.3.4 ConnectionRTInstance class

A *ConnectionRTInstance* is intended to be a run-time instance of a *Connection*, therefore it has a single reference, definition, to the *Connection*.

The *Connection* which the *ConnectionRTInstance* is a run-time instance of has references, source and target. These references indicate the two ports which the *ConnectionRTInstance* is a connection between.

6.3.5 PortRTInstance class

A *PortRTInstance* is intended to be a run-time instance of a *PortInstance*, therefore it has a single reference, definition, to the *PortInstance*. A *PortRTInstance* is contained within a *ComponentRTInstance* and is represented by the single reference, component.

The *PortInstance* which the *PortRTInstance* is a run-time instance of has two multiplicity-many references, in and out, to *ConnectionRTInstances*. These references indicate all the ingoing and outgoing connections attached to the *PortRTInstance*.

The *PortRTInstance* is a container for a set of *MessageRTInstances*. This containment represents the buffer of the *PortRTInstance*.

PortRTInstance has two volatile attributes:

- name : Retrieves the name of the *PortInstance* which the *PortRTInstance* is a run-time instance of.
- noMsg : Retrieves the number of *MessageRTInstances* in the buffer of the *PortRTInstance*. In practice this is done by counting the number of *MessageRTInstances* in the set, buffer.

6.3.6 MsgContainer class

A *MsgContainer* is intended to be used in the buffer of a bus (*BusRTInstance*). It contains a single reference, *message*, to a *MessageRTInstance*, and a single reference, *destport*, to a *PortRTInstance*. The reference, *destport*, is the *PortRTInstance* which the *MessageRTInstance* is to be delivered to in a simulation step.

6.3.7 MessageRTInstance class

A *MessageRTInstance* is intended to be a run-time instance of a *MessageType* (from the *messagedefinition* package), therefore it has a single reference, *type*, to the *MessageType*.

MessageRTInstance has a volatile attribute, *name*, that retrieves the name of the *MessageType* which *MessageRTInstance* is a run-time instance of.

6.3.8 Consequences of using the EMF technology

The *Simulation* only consists of a model which other components use to simulate a deployment. The model is created as an Ecore model and with the help of the code generator of EMF, the model code is generated automatically. What remains to be implemented is volatile attributes and methods.

The following volatile attributes need to be implemented:

- *ComponentRTInstance.name*
- *ComponentRTInstance.type*
- *PortRTInstance.type*
- *PortRTInstance.noMsg*
- *BusRTInstance.name*
- *BusRTInstance.noMsg*
- *MessageRTInstance.name*

The following methods need to be implemented:

- *Simulation.simulate()*
- *Simulation.reset()*

EMF provides a persistence layer which can store data through XMI (as default, other serializations can be implemented). To ensure that a simulation is correctly stored, it is important to have proper containment references between the objects. In the *Simulation* all objects are directly or indirectly contained in the *Simulation* object, and it is therefore only necessary to store the *Simulation* object to ensure that the complete simulation is stored.

7 Glossary

Actuator — *A component capable of carrying out a physical task, triggered by input messages.*

Automaton — *Loosely following definition from Introduction to Automata Theory, Languages, and Computation, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Third Edition, Addison-Wesley, 2007. In our case an automaton is an I/O finite state machine.*

Bus — *A bus (bidirectional universal switch) is a hardware device that transfers data between components.*

CASE tool — *Computer-aided software engineering Tool. The tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, that are most frequently thought of as CASE tools.*

Compatibility (In regards to ports) — *Two ports are compatible if the set of acceptable messages for the sending port is included in the set of acceptable messages for the receiving port.*

Component — *A component can either be a Actuator, a Sensor or a Software Component*

Component Editor — *The component editor is a graphical tool which enables the user to define and edit component types through a "drag and drop" interface.*

Connection — *A connection between two ports.*

Dashboard — *A dashboard is a visualization of a how a deployment would act in an actual embedded system, and it allows the actor to manipulate with the deployment.*

Deployment Editor — *A deployment diagram combines components created in the component editor and thereby modeling an embedded system for which a simulation can be initialized.*

ECU — *Electronic Computational Unit. Sometimes also referred to as a CPU. The component that is able to perform calculations at hardware level.*

Embedded System — *An embedded system is a special-purpose computer system designed to perform one or a few dedicated functions - often on hardware level.*

Interaction Editor — *To observe interactions between components in a deployment, the developers can use the interaction editor.*

Port — *Components communicate with other components through attached ports. A port can have an arbitrary number of connections to other ports.*

Sensor — *A component capable of detecting a specific physical property, and send a message based on this property.*

Simulator — *The simulator contains an algorithm which simulates the message flow between components, and based on these messages, the states of the components are changed as specified in their respective automaton.*

Software Component — *A component capable of evaluating input messages through a program and based on this create one or more output messages.*

A Use cases

A.1 Insert component

Title: Insert component		
Purpose	This use case describes how to insert a component into a deployment diagram.	
Actor	1. A developer.	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided. One or more component definitions should exist.	
Scenario		
Action	Actor	System
1.	The developer selects a component to insert from a list of possible components. The component can either be an actuator, a sensor or a software component.	
2.	The developer inserts the selected component into the deployment diagram.	The system inserts the component into the deployment diagram.
Postcondition	A component has been inserted.	
Alternative and/or exceptional course of events		
Action	Actor	System
A. Insertion of a Software Component outside of an ECU		
2.	The developer inserts a software component into the deployment diagram outside of an ECU.	If the developer tries to insert a software component outside of an ECU a warning is shown and the software component is not inserted.

A.2 Insert bus

Title: Insert Bus		
Purpose	This use case describes how to insert a bus into a deployment diagram.	
Actor	1. A developer.	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided.	
Scenario		
Action	Actor	System
1.	The developer selects a bus to insert from a list.	
2.	The developer inserts the bus into the deployment diagram.	The system creates and inserts the bus into the model of the deployment diagram.
Postcondition	A bus has been inserted.	
Alternative and/or exceptional course of events		
Action	Actor	System
A. Setting the buffer size of the bus different from the default value.		
3.	The developer selects the bus which should have changed its buffer size.	
4.	The developer sets the buffer size of the bus.	The system sets the buffer size of the selected bus to the selected value in the model of the deployment diagram.
1. Attaching a connecting to the bus		
3.	The developer selects the bus.	
4.	The developer chooses the connection which should be attached to the bus from a list.	The system attaches the connection to the bus and removes previous bus attachments.

A.3 Insert ECU

Title: Insert ECU		
Purpose	This use case describes how to insert an ECU into a deployment diagram.	
Actor	1. A developer.	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided.	
Scenario		
Action	Actor	System
1.	The developer selects an ECU from a list.	
2.	The developer inserts the ECU into the deployment diagram.	The system created and inserts the ECU into the model of the deployment diagram.
Postcondition	An ECU has been inserted into the deployment diagram.	

A.4 Insert connection

Insert Connection		
Purpose	This use case describes how to insert a connection into a deployment diagram.	
Actor	1. A developer.	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided. At least one component with a port and one bus has to be inserted into the diagram.	
Scenario		
Action	Actor	System
1.	The developer selects a connection from the list.	
2.	The developer inserts the connection between two ports of components (could be two ports of the component).	The system prompts the developer to select which bus the connection should be attached to.
3.	The developer selects which bus the connection should be attached to.	The system creates the connection and inserts it between the two ports.
Postcondition	A connection has been inserted between two ports of two components.	
Alternative and/or Exceptional course of events		
Action	Actor	System
A. A Connection is inserted between two incompatible ports.		
5.	The developer selects a connection from the list.	
6.	The developer inserts the connection between two ports of two components.	If the developer tries to connect two incompatible ports he is shown a warning and the connection is not created. In this context incompatible means that the set of acceptable messages from the target port of the connections contains all the possible messages from the source port of the connection.
B. A Connection is inserted between two port of two components located inside the same ECU		
1.	The developer inserts the connection between two ports of two components located inside the same ECU	The system inserts the connection between the two ports without prompting the developer for which bus the connection should be attached to. It is not required that two components inside the same ECU connect through a bus.

A.5 Initialize simulation

Title: Initialize simulation		
Purpose	This use case describes how to initialize a simulation of a deployment.	
Actor	1. Developer	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided.	
Scenario		
Action	Actor	System
1.	The developer chooses to run a simulation.	The system executes the simulation algorithm as described in the System feature – Simulation section.
Postcondition	The simulation has been initialized	

A.6 Delete component/ECU/connection

Title: Delete component/ECU/connection		
Purpose	This use case describes how to delete a component, ECU or a connection.	
Actor	2. Developer	
Trigger		
preconditions	The plug-in is installed, started and ready to use. A deployment diagram must be provided. The component, ECU or connection targeted for deletion must exist.	
Scenario		
Action	Actor	System
2.	The developer selects the component, ECU or connection, and deletes it.	<p>The system removes the selected component, ECU or connection, and performs a cleanup based on the type deleted:</p> <ul style="list-style-type: none"> • Component - <i>All connections referring to the component is deleted, if the components is attached to an ECU this reference is removed.</i> • ECU - <i>All components within the ECU is deleted.</i> • Connection – <i>All references to connection is removed.</i>
Postcondition	The selected component, ECU or connection is deleted.	

B UML class diagram - Complete

