Version 5.1

EAST-ADL Tutorial

MetaCase Document No. EAT-5.1

11th Edition, August 2015

MetaCase
Ylistönmäentie 31
FIN–40500 Jyväskylä
Finland

Tel: +358 14 641 000
Fax: +358 420 648 606
E-mail: info@metacase.com
WWW: http://www.metacase.com

Contents

Preface

This tutorial familiarizes the reader with the features offered by MetaEdit+ to support EAST-ADL, an automotive architecture description language.

The reader will be provided with a step-by-step exercise on how to start using EAST-ADL in MetaEdit+. We will start by looking the basic use of MetaEdit+ and then move towards more advanced functions for editing and checking EAST-ADL models. We will also take a look at how to run various reports, like documentation and model checking.

To complete this tutorial you need:

- MetaEdit+ Modeler or MetaEdit+ Workbench (see Section 1.1 for installing MetaEdit+)
- EAST-ADL repository (see Section 1.1 for using EAST-ADL repository)
- A basic knowledge about EAST-ADL language. The description of the language is available at http://east-adl.info/
- 1-2 hour of time to complete the exercises in Chapters 2 to 4. Chapter 5 does not include any exercises but introduces the different modeling languages of EAST-ADL and how the sample models included in the repository can be inspected.

While this tutorial shows the basic modeling features of MetaEdit+ for a more comprehensive description of the modeling features, please refer to the 'MetaEdit+ User's Guide' – available with the MetaEdit+ or via web pages at http://www.metacase.com.

For more information about MetaEdit+ for EAST-ADL, including articles, videos, sample models and white papers, is available at http://www.metacase.com/solution/east-adl.html.

If you want to extend EAST-ADL further – by adding notational symbols, additional constraints, generators or by modifying dialogs and toolbars for the modeling tool − you should have MetaEdit+ Workbench from http://www.metacase.com.

1 Introduction

EAST-ADL is an architecture description language that describes automotive electronic systems. It covers vehicle features, requirements, functions, hardware and software components and related communications. Its extensions focus on verification and validation, dependability and error modeling, etc.

MetaEdit+ allows engineers to create, edit and browse the various kind of architecture models created with EAST-ADL language. A sample of hardware architecture can be seen in Figure 1-1.

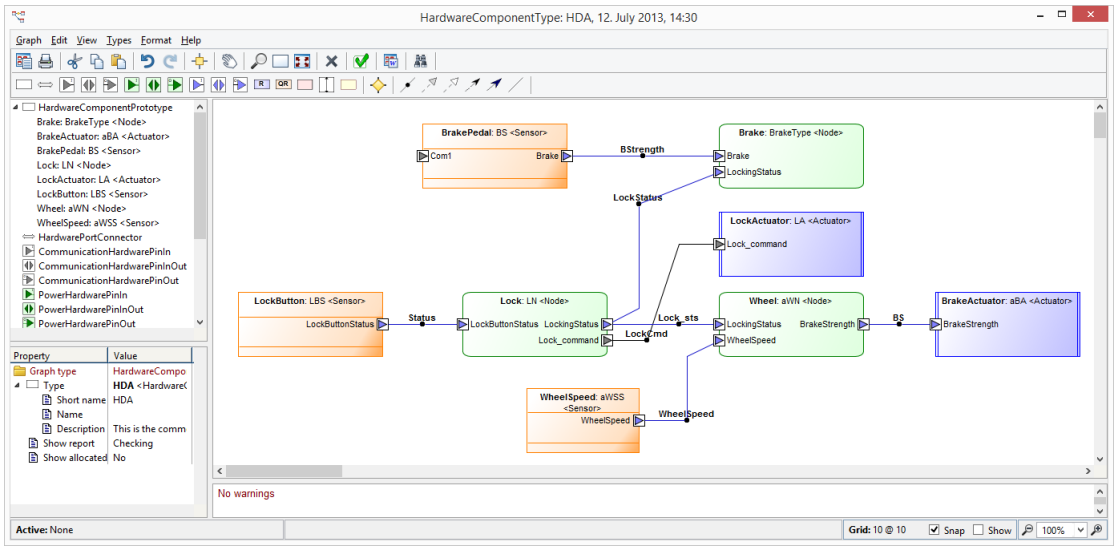


Figure 1-1. A sample hardware architecture

In addition to model creation and editing, MetaEdit+ enables static analysis, document generation, generation of other models and code from the EAST-ADL models as well as export to requirements tools via ReqIF and to various analysis and simulation tools, like Simulink, HIPHOPS and UPPAAL using their own formats.

MetaEdit+ supports also collaborative development with multi-user capabilities as well as language and generator development. This tutorial focuses on modeling and model checking functionality with EAST-ADL in the single user environment. For inspection of the more advanced features see MetaEdit+ User's Guide or contact MetaCase.

1.1 DOWNLOADING AND INSTALLING METAEDIT+ AND EAST-ADL LANGUAGE

For exploring EAST-ADL tutorial thoroughly, you need MetaEdit+ tool and EAST-ADL repository. Contact MetaCase (info@metacase.com) to receive instructions for downloading these.

After downloading the installation package run the installation program and follow the installation instructions. MetaEdit+ installation program provides you also the User's Guides of MetaEdit+. You can open these guides by pressing **F1** key when using MetaEdit+.

After installing MetaEdit+ tool you should extract the EAST-ADL repository file into the same folder you have installed MetaEdit+. As default MetaEdit+ working directory should contain '*demo*' repository coming with the standard MetaEdit+ installation. In Windows this working directory is under your '…*YOUR USERNAME*\My Documents\MetaEdit+ 5.1'. After extracting the zipped repository you should have two separate folders like: '*demo*' and '*EAST-ADL*' repositories on the same level (both of these repositories should have subfolders named as '*areas*', '*backup*' and '*users*' and two files '*manager.ab*' and '*trid*'). Make sure you have read and write rights to the directory.

In case that you already have existing EAST-ADL repository (downloaded earlier), download just the patch files (`*.met` and `*.mxs`) and follow the patching instructions from the Section 7 in this manual.

2 Logging into MetaEdit+

MetaEdit+ is a repository-based application and therefore a connection with the repository must be established during the start-up. When started, MetaEdit+ opens a Startup Launcher as shown in Figure 2-1.

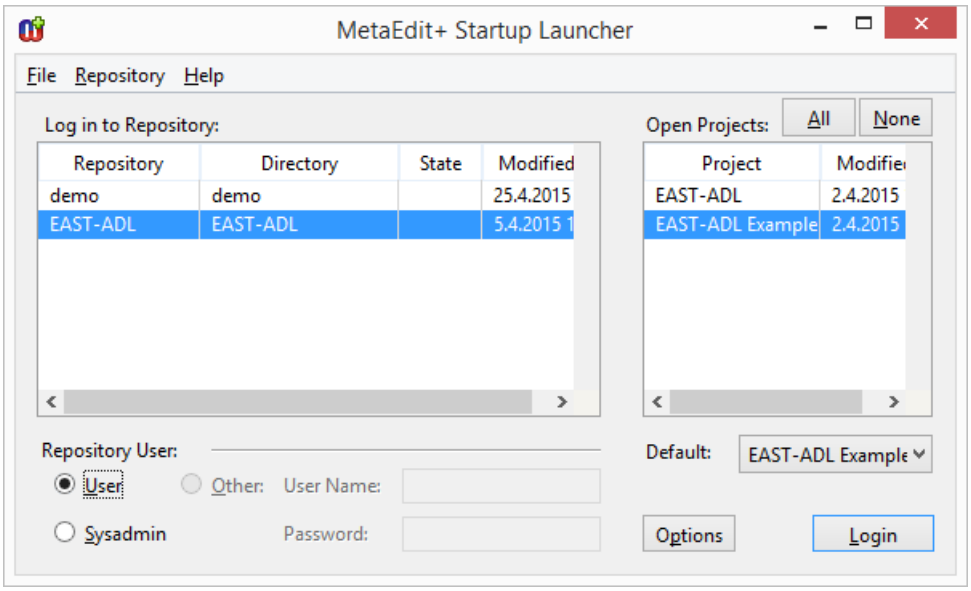


Figure 2-1. Startup Launcher

Make sure that the 'EAST-ADL' repository is selected from the **Log in to Repository** list and 'User' is selected as **Repository User**. Select 'EAST-ADL Examples' from the **Open Projects** list (as in Figure 2-1) and press **Login** button.

If you are using MetaEdit+ evaluation version and this is your first login, a dialog will open for entering your evaluation license code. Enter the evaluation license code exactly as given in the evaluation instructions (sent to you by email when downloading the evaluation version) and press **OK**.

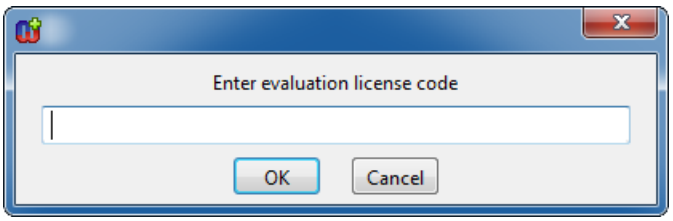


Figure 2-2. Evaluation license code

After a successful login the Startup Launcher will close and you should see the main MetaEdit+ Launcher (Figure 2-3). MetaEdit+ is now ready to be used.

Launcher provides a number of browsers and by default Graph Browser is opened. Each browser provide different possibility to navigate the models and model elements, set filter when working with large models and use different hierarchies for navigation.

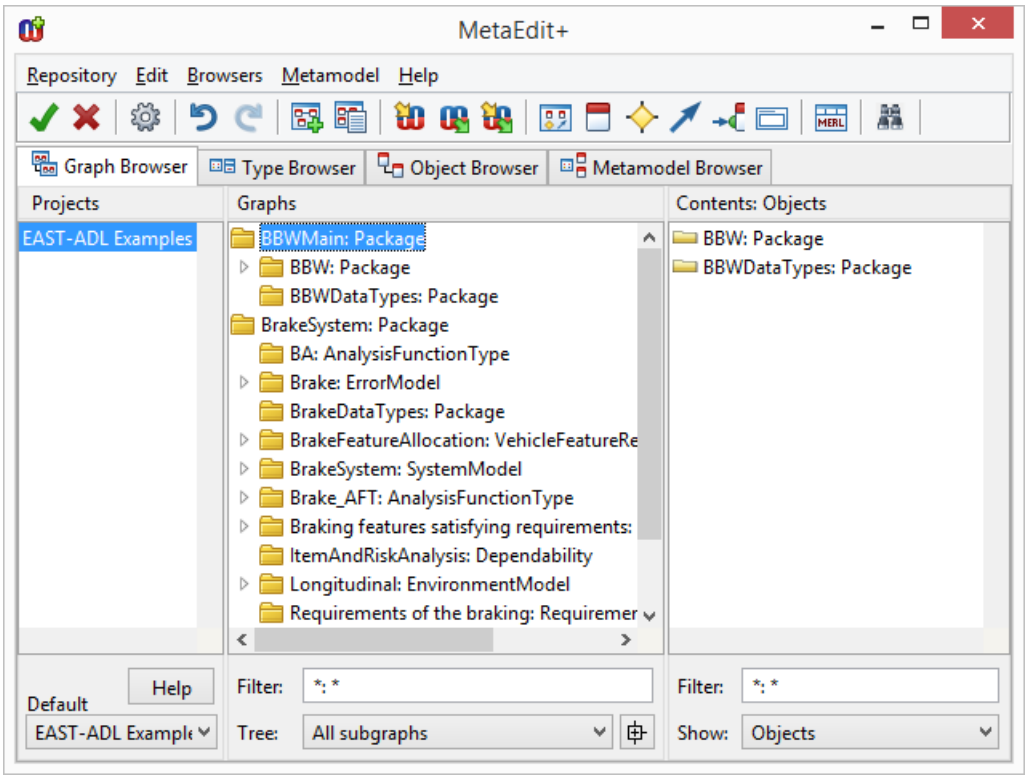


Figure 2-3. MetaEdit+ Launcher

If you want to quit MetaEdit+ before completing this tutorial, please see Section 3.7 for exit instructions.

3 Basic modeling functions

In this Chapter we create a small design model and then in Chapter 4 inspect more advanced modeling capabilities.

By default the models we create are stored into the '*EAST-ADL Examples*' project. That is the project we opened and it is also marked as the default project in MetaEdit+ Launcher (see setting in bottom left, Figure 2-3). You can also create any number of new projects for your work. Later in Section 5.1 we look how to work with other projects.

3.1 CREATING A EAST-ADL DIAGRAM

Let's start modeling by creating hardware architecture for a small system. Create a new graph by selecting '**Create Graph**' button from the toolbar. When MetaEdit+ asks you to choose the graph type, choose '**HardwareComponentType**' from the list (as in Figure 3-1), and press **OK**.

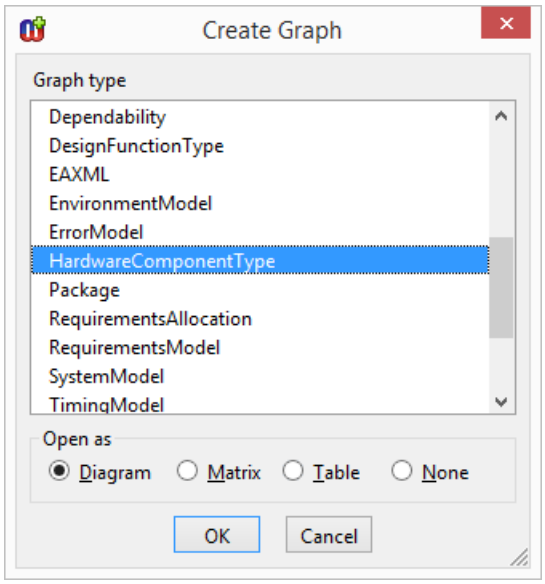


Figure 3-1. Choosing the graph type to be created

You are next asked to enter the details of the system for which the Hardware Architecture model will be created. These details are entered in property fields as shown in Figure 3-2. The dialogs contain four types of property fields:

- Short string fields, longer text fields or lists. All these have a white background color. Values for these strings and text fields can be inserted by just activating the field and then typing the value or selecting the required list value from the pull-down list (like "Checking" in the '*Show report*' field in Figure 3-2).
- For the collections user creates and modifies the values by using the right mouse popup menu. Existing item can be opened by double-clicking the item, which will open the selected item in own property dialog.

- Boolean or radio button values can be activated by clicking the chosen item after the property name, see '*Show allocated prototypes*' in Figure 3-2.
- Grey property fields ('*Type*' in Figure 3-2) have a link to the other object ("MySystem: HardwareComponentType" in the sample Figure 3-2). The details of these objects are specified in a separate property dialog. You may create a new object by double-clicking the grey property field. After double-clicking you may select if you want to create a new object or reuse an existing one. If you have entered a value earlier for this property field, double-clicking the property value opens the corresponding property dialog.

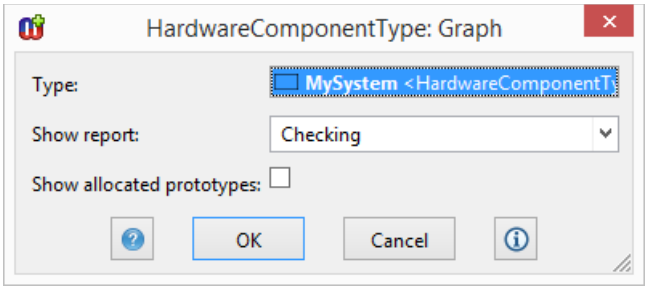


Figure 3-2. Entering the name for the hardware architecture

Next insert a new HardwareComponentType by double-clicking the grey '*Type*' field. Among the possible kind of types, choose '*HardwareComponentType*'. Enter new property values like in Figure 3-3. When editing the large text field you can also open a text editor by choosing **Editor…** from the field's pop-up menu. Text editor is an internal editor or your favorite external editor depending on your settings (see MetaEdit+ User's guide for more details).

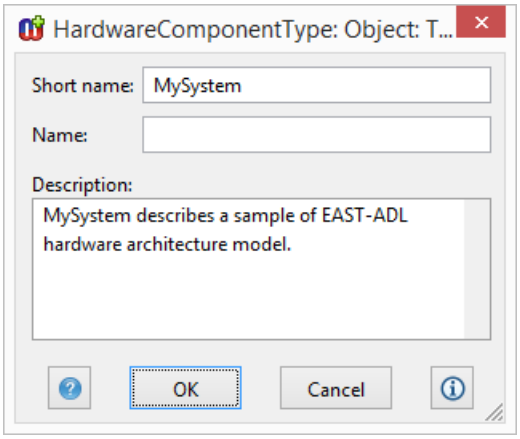


Figure 3-3. Entering the name and description for the hardware component type

Press **OK** to close both dialogs. Now MetaEdit+ opens an empty Diagram Editor for specifying the hardware architecture (as in Figure 3-5).

At the top of the Diagram Editor window you will find the menu bar. Below the menu bar is an action toolbar, and below that another toolbar that contains the object and relationship types for the modeling language in use. In EAST-ADL case the toolbar should look different, depending on the chosen modeling language. The main part of the window is reserved for creating hardware architecture model. There is a sidebar on the left with a tree view of all object types and instances in the graph, and below of it you can find a property sheet showing the properties of the active element (as in Figure 3-4).

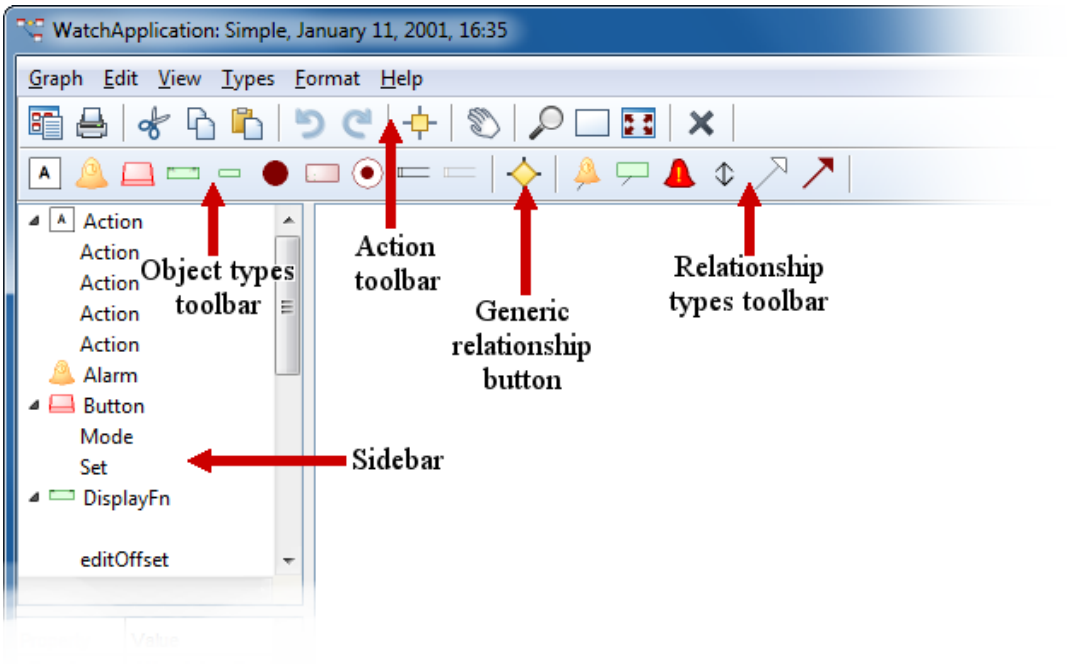


Figure 3-4. The Diagram Editor toolbars

At the bottom of the window there is a LiveCheck pane, which is used for reporting the useful information during the modeling time, like model checking, hardwarePortConnector connections or unconnected pins found from the model. User may select the suitable information by setting the '*Show report*' value on the property sheet for the model. By clicking the underlined elements in the LiveCheck pane you may trace back to the corresponding element in the diagram. Under the LiveCheck pane there is a status bar shows information about the selected element on the canvas, its subgraphs and controls for diagram grid and zoom.

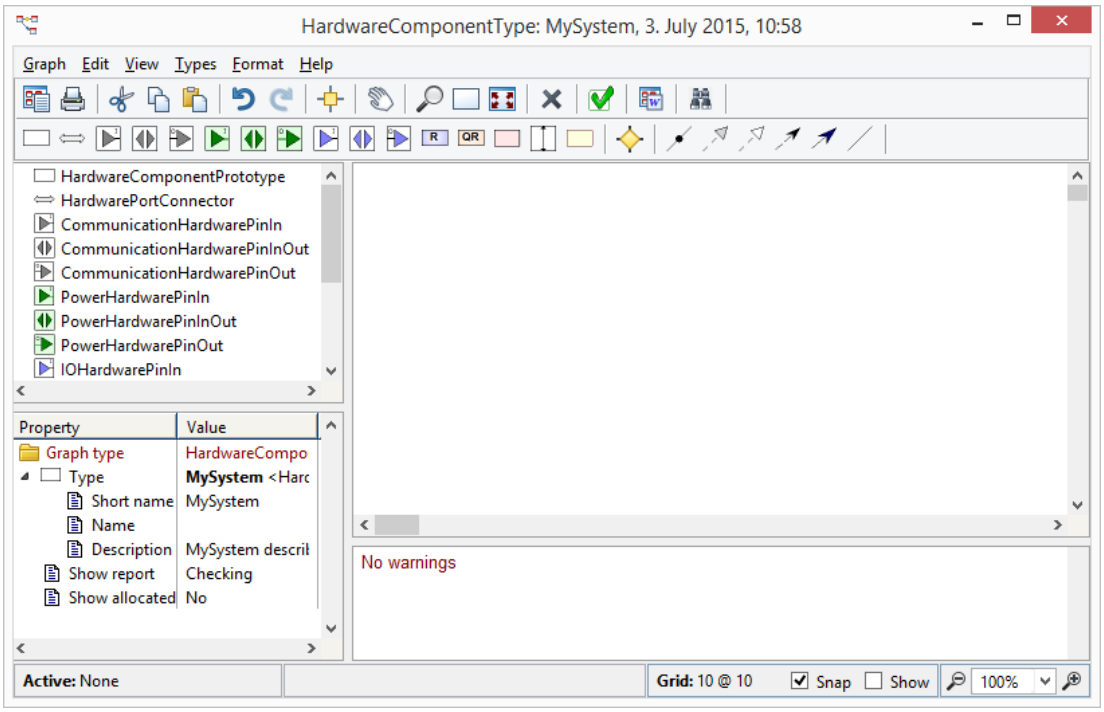


Figure 3-5. The Diagram Editor for hardware architecture modeling

3.2 ABOUT THE HARDWARE ARCHITECTURE LANGUAGE

The modeling concepts of the language are shown in the editor's toolbar (the objects are on the left side and relationships on the right side). They both can also be seen from Types menu. Figure 3-6 shows the contents of the Types menu: The modeling objects and relationships of EAST-ADL hardware architecture with their notational symbols with labels.

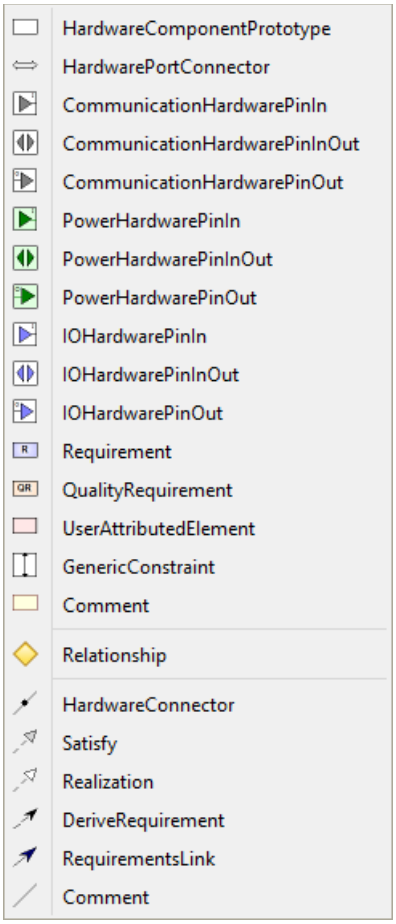


Figure 3-6. Modeling concepts for hardware architecture modeling

The first set of concepts includes the modeling objects - most importantly the HardwareComponentPrototype and HardwarePortConnector. A more detailed description of these concepts can be found from Appendix A.

Remaining objects deal with the various pins forming the boundary interfaces of the system. In EAST-ADL there are 'Communication', 'Power' and 'IO' pins. Types list contains also the 'Requirements, 'UserAttributedElement', 'GenericConstraint' and 'Comment'. 'Comment' is an extra object, which can be used to attach visual descriptions or comments to the models.

Hardware architecture allows six kinds of relationships among the objects, most importantly the 'HardwareConnector', 'Satisfy' and 'Realization' relationships. Requirements are connected with other elements via the Requirements relationships and the 'Comment' is used to link a Comment object to any of the other model elements.

3.3 ADDING OBJECTS TO THE DIAGRAM

New objects, like prototypes, can be created in several ways but here we look just one way: choosing the specific object type from the toolbar. In the Diagram Editor, click the **HardwareComponentPrototype** button on the toolbar (see Figure 3-7) and then click the left mouse button in the canvas.

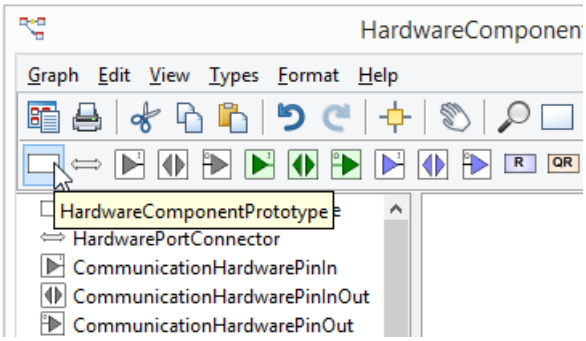


Figure 3-7. Choosing object type from the toolbar

Next you can enter the information about the prototype in the dialog that opens (see Figure 3-8). In EAST-ADL, model elements must have a short name. This mandatory field needs to follow naming rules that enable XML-based model exchange (see Chapter 6). The optional name field allows entering descriptive name of the element without formatting restrictions. Enter the short name for the new HardwareComponentPrototype (e.g. '*LeftWiper'*), optionally a name and a description and press **OK**.

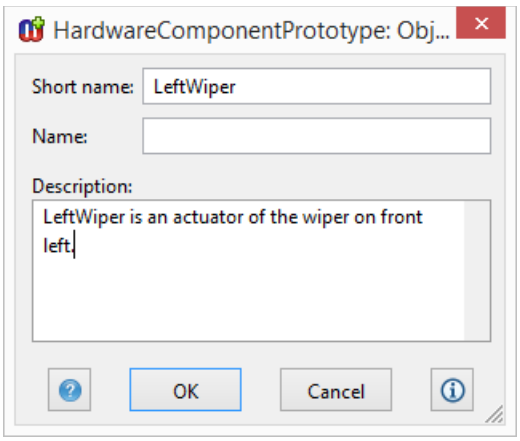


Figure 3-8. Entering information about an object

LeftWiper HardwareComponentPrototype is now shown in the drawing area of the Diagram Editor. If the optional name was given it will be shown inside the symbol. Double-clicking the prototype symbol will open the property dialog again for editing.

If you want to create several prototypes you can keep CTRL key pressed when choosing the type from the toolbar. This makes selection sticky. To remove the sticky selected item, click the selected item in the toolbar again.

Now create a few more other kind of prototypes in the same way. Moving the symbols around and resizing those works the same way as in any drawing tool. Adding all other object types works in the similar manner: first selecting the object type from the toolbar, then placing the object on the canvas and then giving the details in the property dialog and accepting the changes by OK.

After you have drawn a few more objects, your diagram in the Diagram Editor could look something like this:

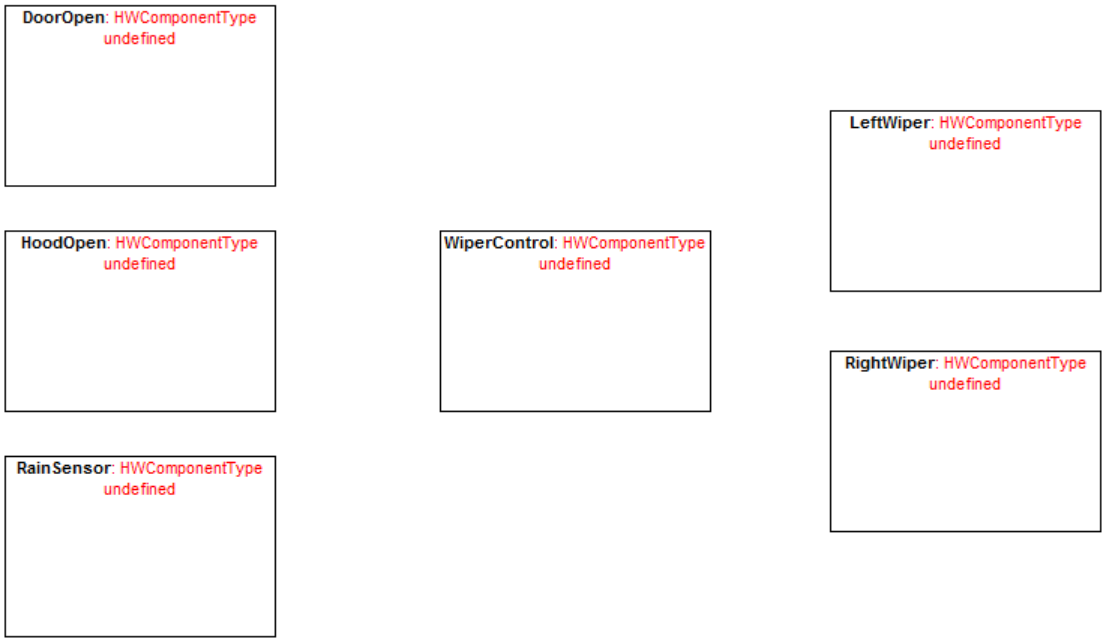


Figure 3-9. The Diagram Editor with objects

The diagram shows three sensors on the left (*'DoorOpen'*, *'HoodOpen'* and *'RainSensor'*), one controlling node in the middle (*'WiperControl'*) and two actuators (*'LeftWiper'* and *'RightWiper'*) on the right.

3.4 DEFINING TYPES FOR THE PROTOTYPES AND PINS

In EAST-ADL each prototype must have a corresponding type definition. If type definition is not given all prototypes look similar, just white rectangles with the name on the top (see Figure 3-9). Undefined types are also reported in LiveCheck pane under the drawing area and model checking reports (see Section 4.1 for details).

When creating hardware architecture you may create prototypes without considering yet their type definitions. You may also start by defining the types immediately or - as in most cases - start using already existing types.

Since we don't have any types defined yet, let's next define types for the wipers. To define the type you must link a subgraph (including the type definition) for each prototype.

To create a type definition for the prototype: First select the prototype (*'LeftWiper'*) and next choose '**Open Subgraph**' from its pop-up menu (or Ctrl + double click the prototype). This opens Subgraphs dialog to select some existing types or create a new one. Now let's create a new type definition by selecting the item '**HardwareComponentType | Create new decomposition…**' from the list (see Figure 3-10) and pressing **OK**.

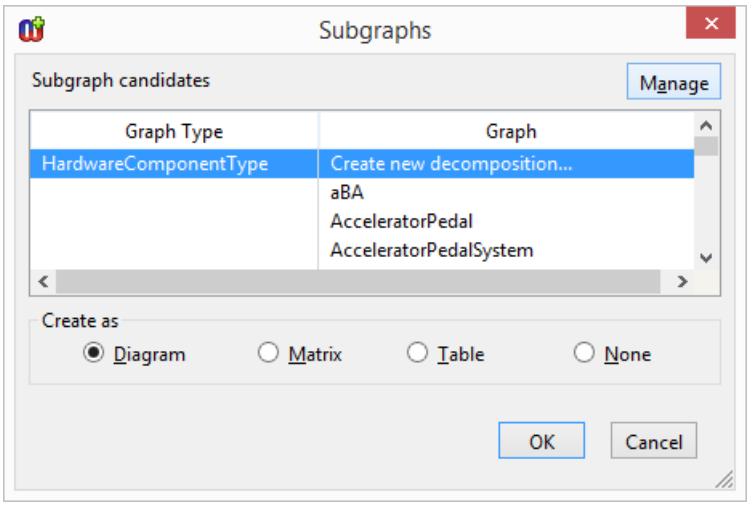


Figure 3-10. Creating a new subgraph

Next enter the details for the **type called** '*Wiper*'. For the each hardware component type, first select the corresponding type for the HardwareComponentType property field value: for Actuators → ActuatorType, for Nodes → NodeType etc. Here select ActuatorType (Figure 3-11), give Wiper details and press OK as in Figure 3-12.

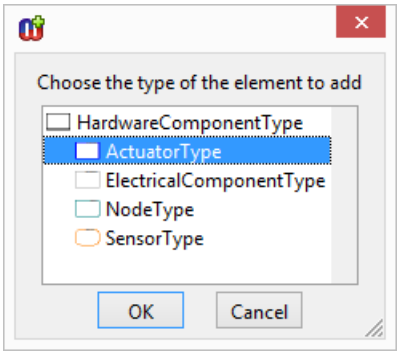


Figure 3-11. Selecting a type

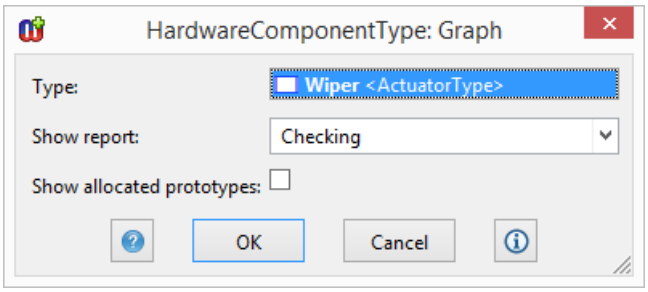


Figure 3-12. Wiper actuator details

After closing the property dialogs, an empty '*Wiper*' subgraph will open. Each Type definition (i.e. subgraph) may contain pins and other prototypes following the type-prototype hierarchy of EAST-ADL.

Let's define next the Pins for that '*Wiper*' type. Select the **CommunicationHardwarePinIn** from the toolbar (third from the left) and then click the empty place on the canvas. Enter the value '*WiperInput'* as a short name for this Pin, like in the Figure 3-13. All the pins defined in the subgraph will be available on the perimeter line of corresponding prototype and allow creating the connections between the prototypes via the pins.

You may also reuse the existing property value by using the '**Share Property…**' functionality for each property field pop-up. Please notice that changes in the shared property values will occur in all places where the same string has been used. All shared properties are presented with red color for string or text. If you want to delete the property sharing link, select '**Remove Sharing**' from the property field's popup.

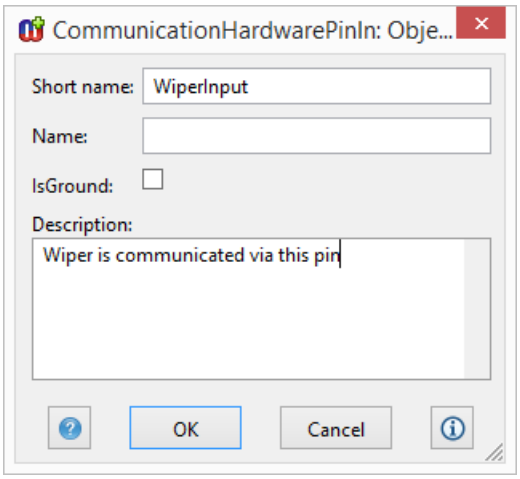


Figure 3-13. Pin's properties

After you have created a '*WiperInput*' Pin the subgraph for the Wiper type looks like Figure 3-14.

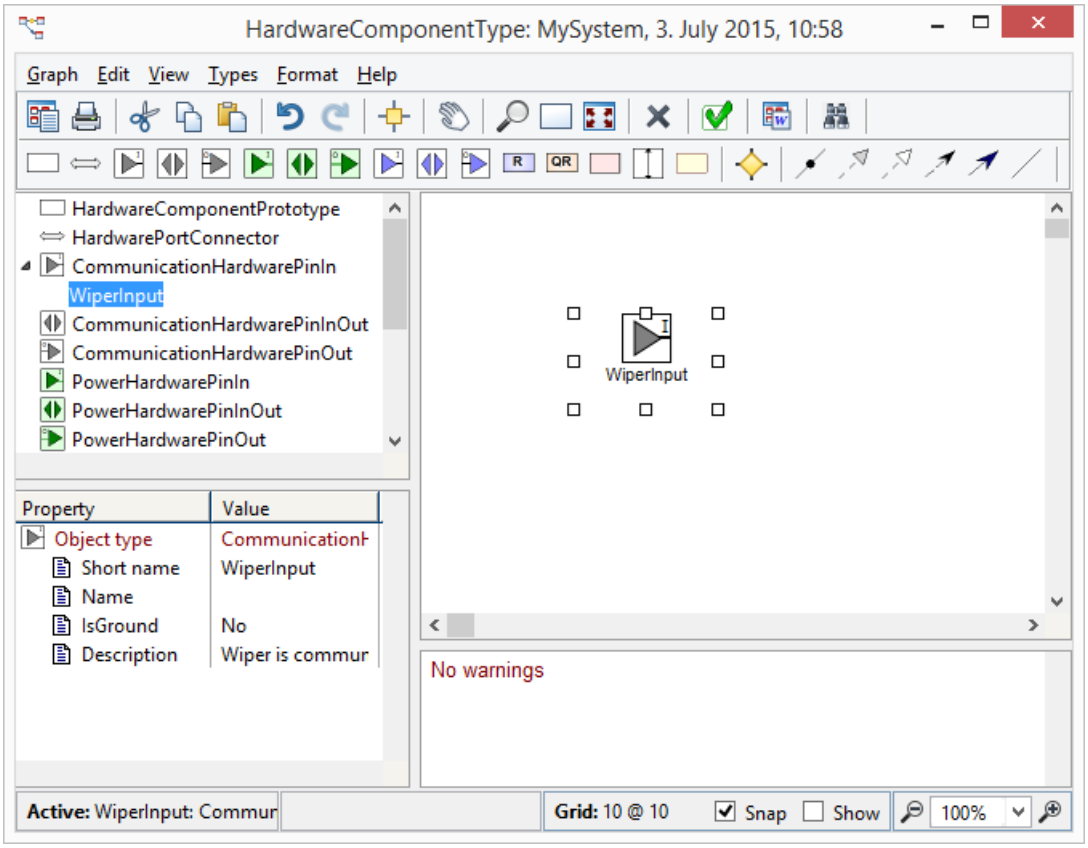


Figure 3-14. WiperInput Pin definition for Wiper actuator type

You may now close this type definition of Wiper and view the previously created '*MySystem*' hardware component type. The '*LeftWiper*' should now contain the type definition and also show the defined pin ('*WiperInput*') inside the prototype symbol on the left of the perimeter

line (see Figure 3-15). Every prototype shows the pins available on the perimeter line - incoming pins on the left side and outgoing on the right side. The same pin type may be connected multiple times.

For improved readability each pin type has own symbol: an arrow pointing the direction of the connection and color presenting the type of the pin; grey for the communication, blue for the input/output and green for the power.

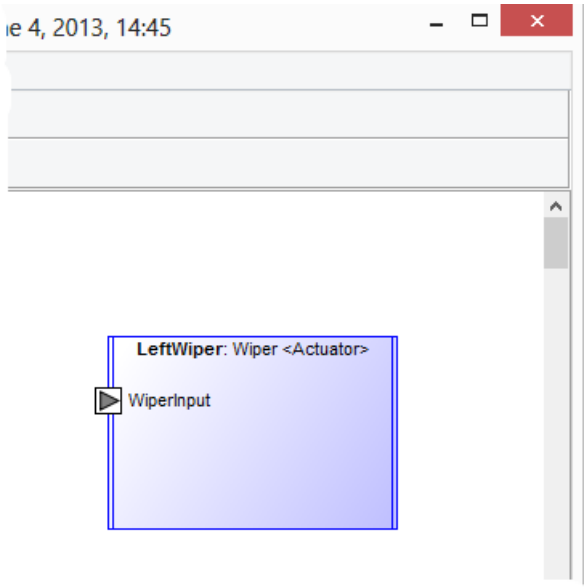


Figure 3-15. WiperInput Pin for LeftWiper

Type definitions can be reused among several prototypes. In our example the '*RightWiper*' prototype shares the same '*Wiper*' definition. To use the same type, select '**Open Subgraph'** for the '*RightWiper*' prototype. From the selection list (Figure 3-16) choose the existing **Wiper** and click **OK**.

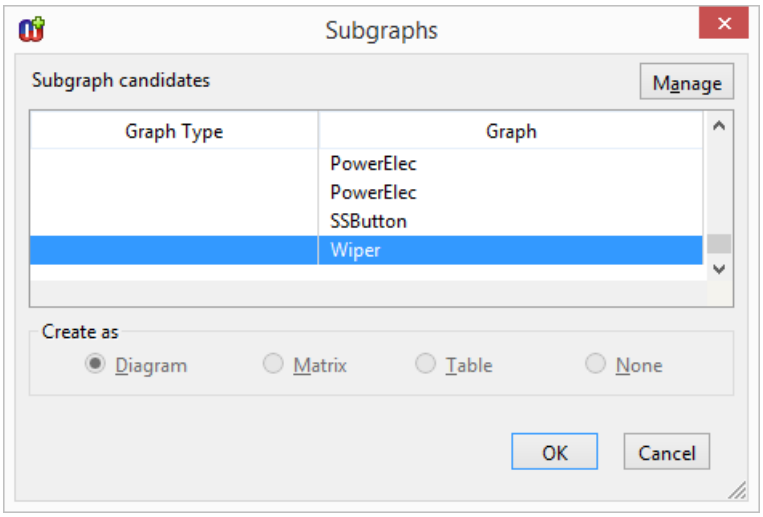


Figure 3-16. Reusing the existing Wiper definition

If you refresh (F5) the '*MySystem*' hardware architecture diagr, you will see that now '*RightWiper*' has similar type definition like the '*LeftWiper*', see Figure 3-17.

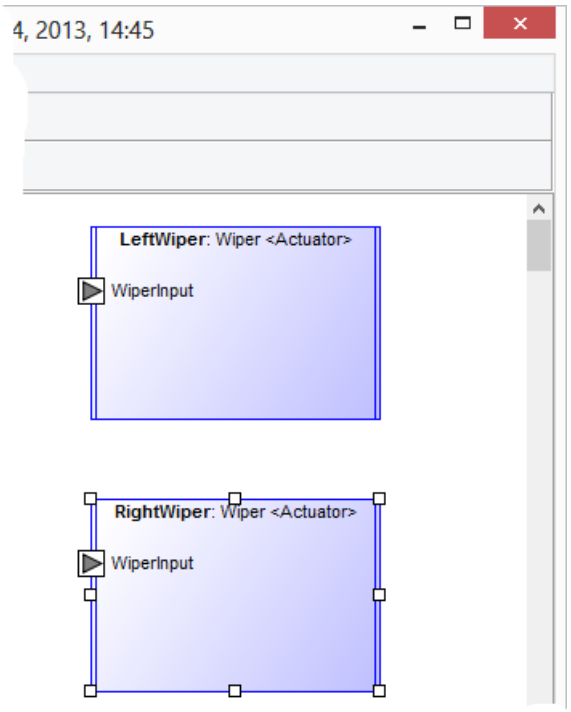


Figure 3-17. Wiper definition reused for both wipers

After creating all the type definitions and defining their pin definitions, '*MySystem*' should look like Figure 3-18:

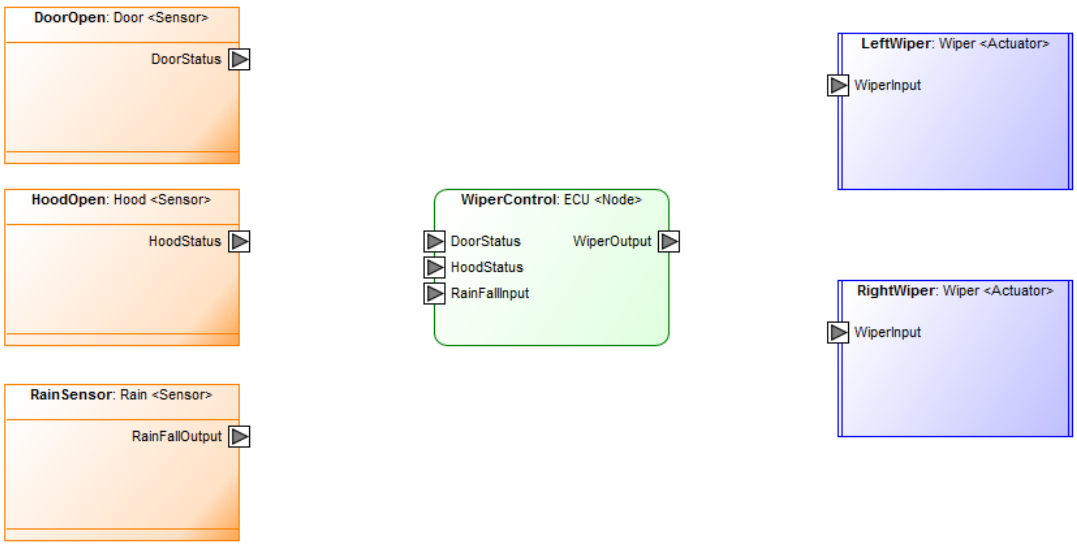


Figure 3-18. Types defined for the prototypes

3.5 ADDING RELATIONSHIPS TO THE DIAGRAM

The next things we want to add to our diagram are the hardware connectors between the prototypes. Let's look the simplest way to create a communication relationship between '*WiperControl:ECU'* <Node> and '*LeftWiper:Wiper'* <Actuator>.

→ *Please note that taking the window focus out of the Diagram Editor during the relationship creation will cancel the operation.*

First, press the **Relationship** button from the toolbar as illustrated in Figure 3-19. This selection triggers the creation of any kind of relationship.

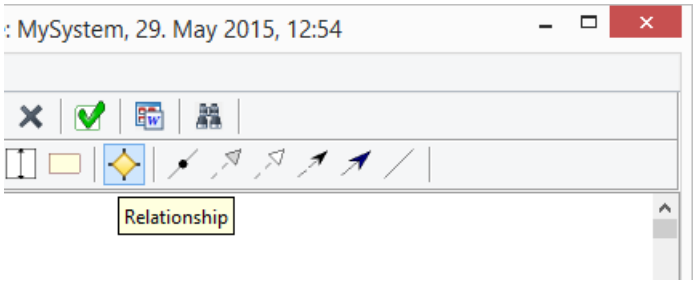


Figure 3-19. Selecting any kind of relationship to be added

Move the cursor over the '*WiperControl:ECU'* object and select its highlighted outgoing pin ('*WiperOutput'* symbol on the perimeter line) (step A in the next Figure 3-20) and click the left mouse button. Next move the cursor over '*LeftWiper:Wiper'* and select its highlighted incoming '*WiperInput*' pin, (step B) with the left mouse click.

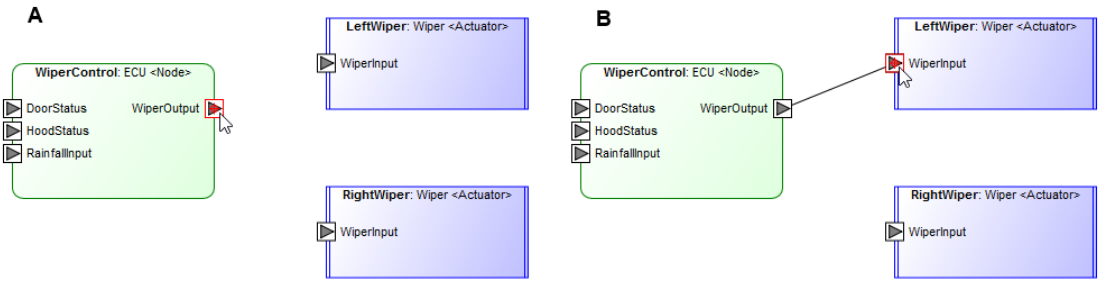


Figure 3-20. Selecting objects to be connected with a relationship

A new property dialog opens for specifying the properties for the hardware connection. The first active tab shows the properties of the hardware connection and two remaining tabs present the pins of the connection. All the pin properties were already given in the subgraph so other tabs are empty (and shown in italics). Let's enter the hardware connection details and press '**All OK**' to add the communication connection, see Figure 3-21.

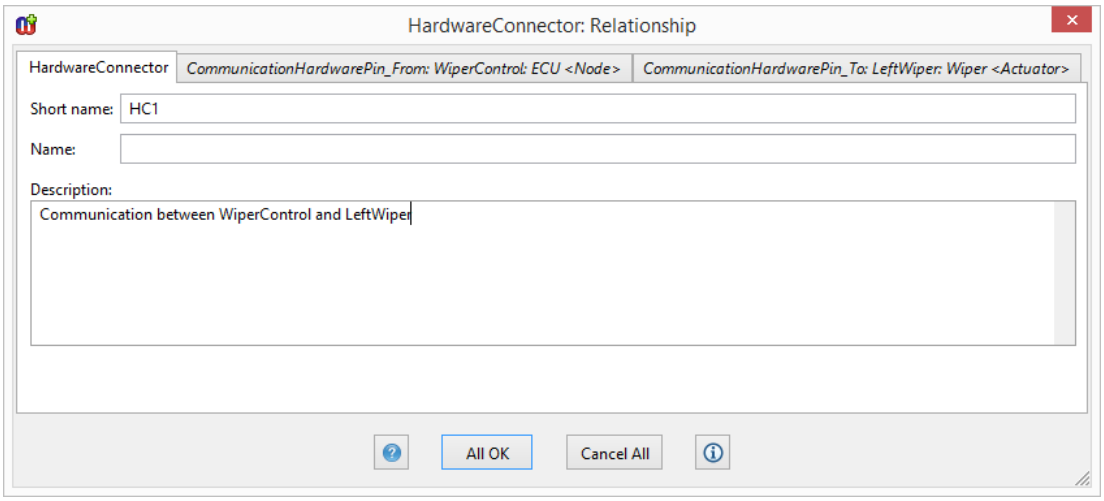


Figure 3-21. Dialog for entering properties for hardware connection

MetaEdit+ follows the EAST-ADL language and knows automatically that there may be only '*Communication*' (HardwareConnection) between these two communication pins. It also knows the direction of the connection: from 'Out' port to 'In' port.

You may continue by specifying other connections between the prototypes. Please notice that some connections may be defined between pins and some may connect directly the prototypes together, like '*Realization*' relationship.

An alternative way to create a new relationship is choosing the exact kind of relationship type from the toolbar. You may add relationships also by selecting '**Connect…**' from the prototype's pop-up menu (e.g. '*WiperControl:ECU'*) and after that, left click the source outgoing pin and next the target incoming pin. For alternative ways to add connections please see MetaEdit+ User's Guide. After you have drawn the rest of the relationships, '*MySystem*' should look something like in Figure 3-22:

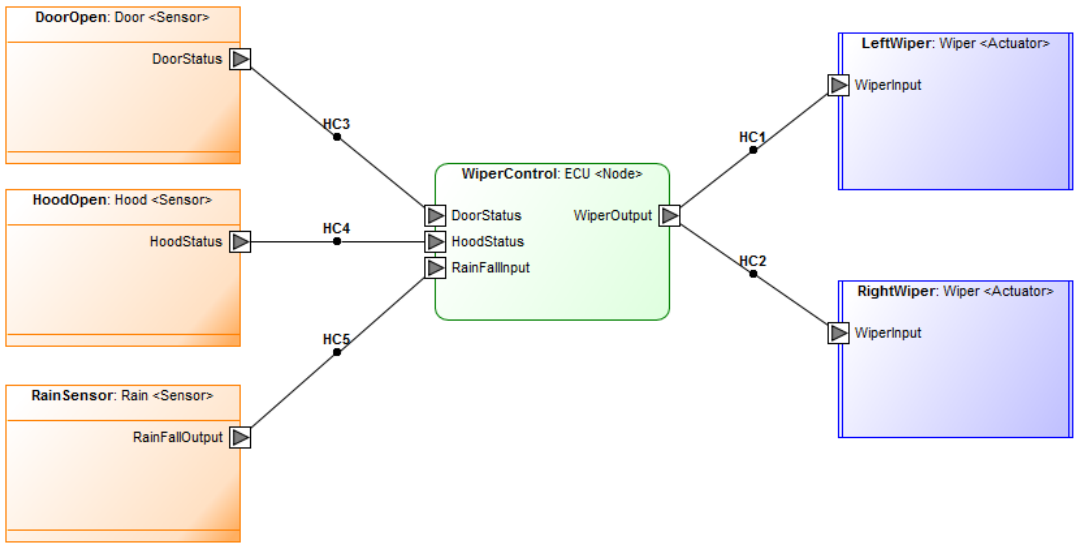


Figure 3-22. MySystem with HardwareConnections

If you want to create additional breakpoints for the relationship line during the relationship creation you may click in the empty canvas space between the objects. Later you may add the breakpoints by just moving the line.

If your connections are utilizing the HardwarePortConnector, you need to first add HardwarePortConnector object type to your model. Next enter the details of it and then attach the chosen HardwareConnectors to the collection field. You may add existing items from collection's popup menu via **Add Existing…** functionality, see Figure 3-23.

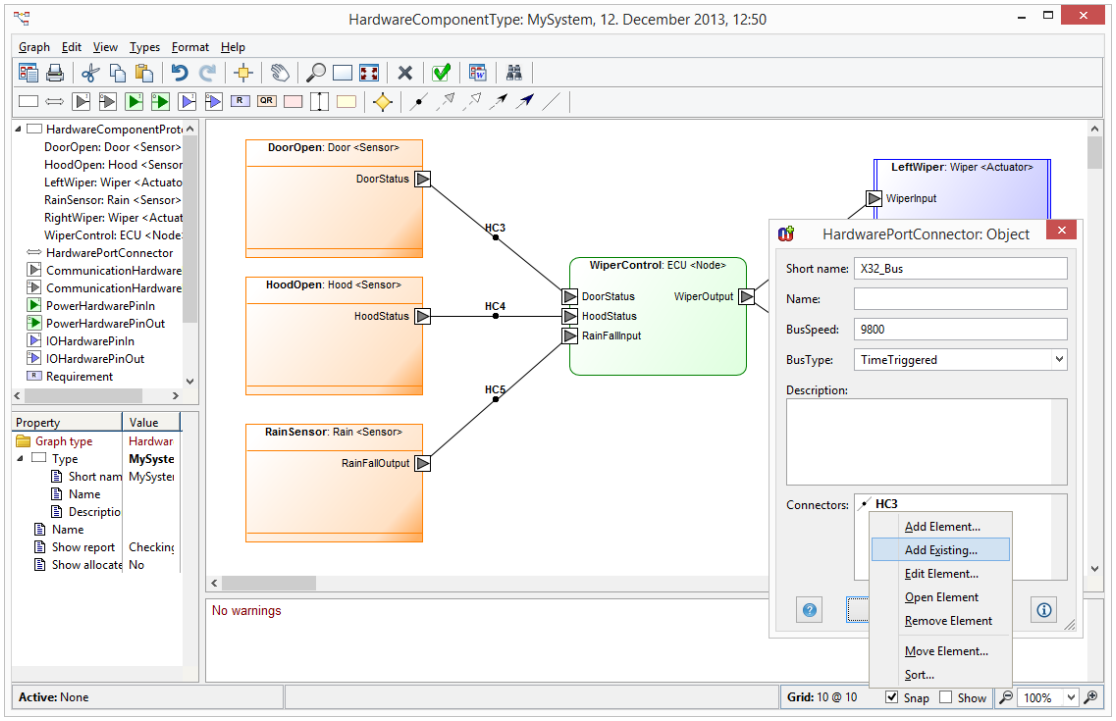


Figure 3-23. HardwarePortConnector information

Connections which utilize the HardwarePortConnector are visualized with a wider background color for the relationships; see "HC3" and "HC4" in the Figure 3-24 illustrating this.

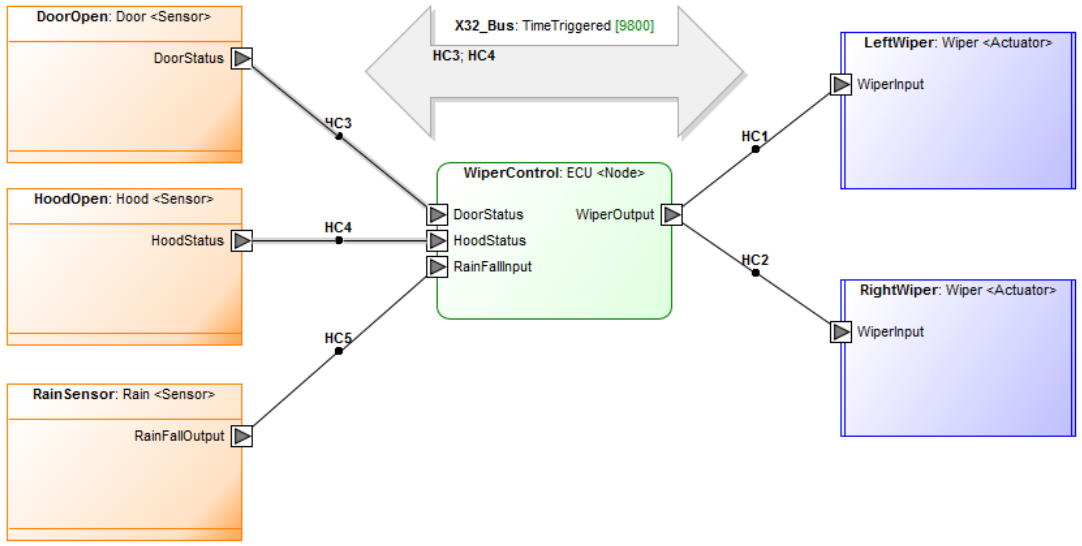


Figure 3-24. The finished hardware architecture diagram

Now we should have a more complete design and the model checking and error annotation should not show any warnings - so syntactically the hardware architecture is correct, complete and consistent. Please note that some of the warnings shown may relate to the XML exporting issues, i.e. unnamed actuators, sensors or hardware connectors.

3.6 SAVING YOUR WORK

As we have now defined the first part of the hardware architecture, it is a good time to save our work. Go back to the MetaEdit+ launcher. There you can find **Commit** and **Abandon** buttons (as shown in Figure 3-25).

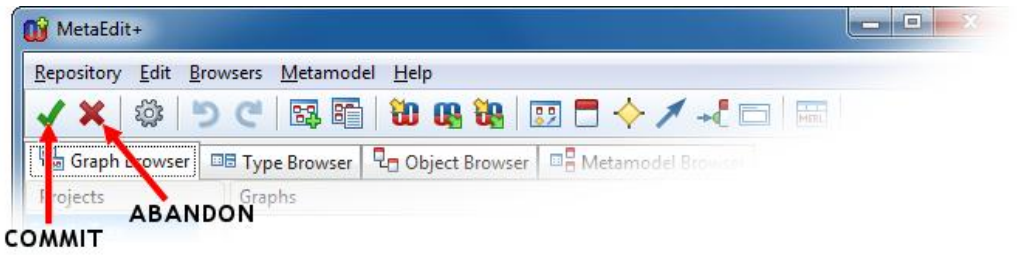


Figure 3-25. Commit and Abandon buttons in the MetaEdit+ launcher

Pressing the **Commit** button saves the all changes made in this transaction into the repository, while pressing **Abandon** restores the state saved by the previous commit.

3.7 EXITING METAEDIT+

You have now completed the first tasks in developing hardware design architectures in EAST-ADL modeling language. If you want to quit this tutorial at this time, you can exit MetaEdit+ by selecting **Repository | Exit…** from the main Launcher menu. If you have uncommitted changes, MetaEdit will show the dialog in Figure 3-26. Choose **Commit and Exit** or **Abandon and Exit** depending on whether you want to save the latest changes.

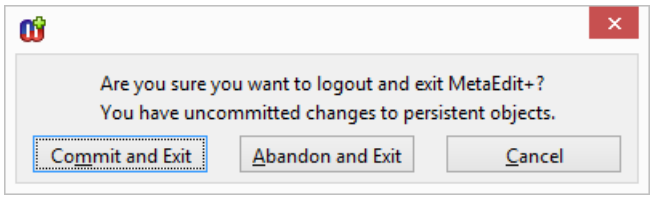


Figure 3-26. Logout with uncommitted changes

However, if you are interested in seeing the other modeling languages of EAST-ADL, start MetaEdit+ again and continue with the next chapter. If not, skip straight to the conclusion in Chapter 8.

4 Advanced modeling functions

We have now successfully created a small design of the hardware architecture. Let us next apply this modeling language in more detail, and get familiar with some of the more advanced features provided by MetaEdit+.

4.1 MODEL CHECKING

MetaEdit+ checks the rules of EAST-ADL language when creating, editing and removing the model elements. Some of the rules are also checked with model checking reports as well as annotating errors directly in models.

There are several ways to find and report the correctness, completeness and consistency of the model already during the modeling time. Some of the checks are automatically followed already during the modeling time, like entering empty or illegal values to the elements or creating illegal kind of connections between the model elements.

Some of the checking are annotated directly in the model and will be shown after the data has been given. As a default the bottom pane of the diagram editor will show these checking reports. There is also possibility to change that setting if you open the Graph properties by double-clicking the property sheet at the bottom left of the model or select **Graph | Properties….** In Hardware architecture models you may change the **Show report** value to report '*Checking*' or list the '*Unconnected pins*'.

You may also run the model checking by pressing **Check** button from the toolbar. The result of the model checking is shown in a separate report output window. By clicking the underlined elements listed in the report traces you back to the element in the diagram so you can edit the model elements and correct the model.

4.2 MODEL HIERARCHIES

EAST-ADL uses a type-prototype hierarchy to specify the internal structure of the hardware architecture. For example, *MySystem* (HardwareComponentType) may contain a number of prototypes (Sensors, Power supplies, etc.) and their connections. There is no limit to the depth of the hierarchy.

You may create a subgraph containing the pin definitions first and later attach these subgraphs to the prototype via **Open Subgraph** functionality or Ctrl + double click the element similarly like in Section 3.4. You may also have all types specified in separate package first, which is one graph type similar to the HardwareComponentType, and then create just prototypes using the already defined types. This way all the models, and all the developers, use the same commonly and only once defined types.

4.3 GENERATING DOCUMENTS FROM THE HARDWARE ARCHITECTURES

In many cases it would be very desirable to be able to produce various outputs of the information stored in a diagram, e.g. as web pages, allocation reports, data for simulators, Simulink models, software configuration, etc. These can be produced with MetaEdit+'s generator system. Depending on the modeling languages used, different generators can be applied. As an example, let's generate Word document and HTML web page from the already created Hardware Architecture.

4.3.1 Word document generation

Documentation can be generated from Hardware Architecture models by selecting the **Doc** generator icon from the toolbar. This produces documents in RTF format and creates finally Word documents using a given template. The generated document includes the hierarchy of the models - starting from the diagram the generator was executed.

→ *The document generation applies macros in the Word to create the styles, headings etc. You may need to change your security settings and allow running the macros when opening the generated document in Word. If macros are not accessed, the resulted file is stored and opened only as .rtf file. (You can edit your security settings by clicking the "Macro Security" in the Developer ribbon)*

→ *After you make any changes in the security, please run the generator again. If macros are accessed correctly, the resulted document should be stored in .doc format and have all the pictures, tables & hyperlinks as well as table-of-contents included*

→ ***Please notice: after you change the security settings in Word, then changed setting will remain to all other Word sessions too!***

4.3.2 HTML generation

Another way to launch the generator is select **Graph | Generate…** from the menu (or press the **Generate** button in the Diagram Editor's action toolbar). A list of the available pre-defined reports will appear. Choose **Export graph to HTML** from the list (Figure 4-1) and press **OK**.

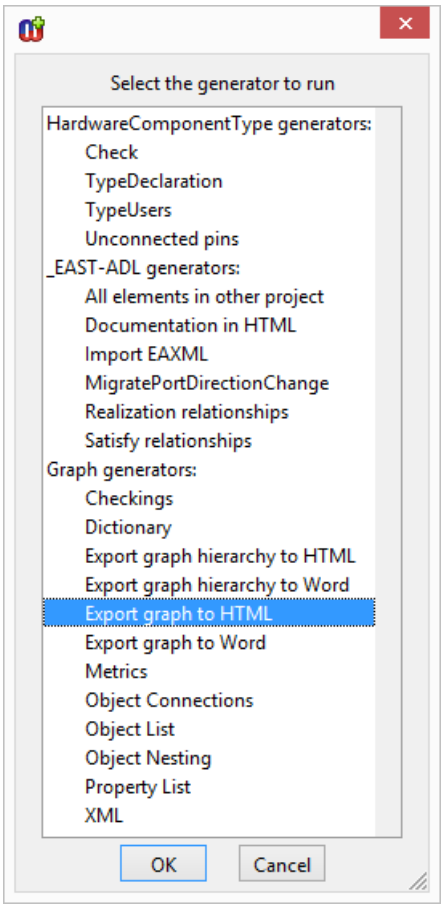


Figure 4-1. Selecting generators

MetaEdit+ produces the HTML output, reports the success of the generation and opens the generated web page in your default browser (Figure 4-2).

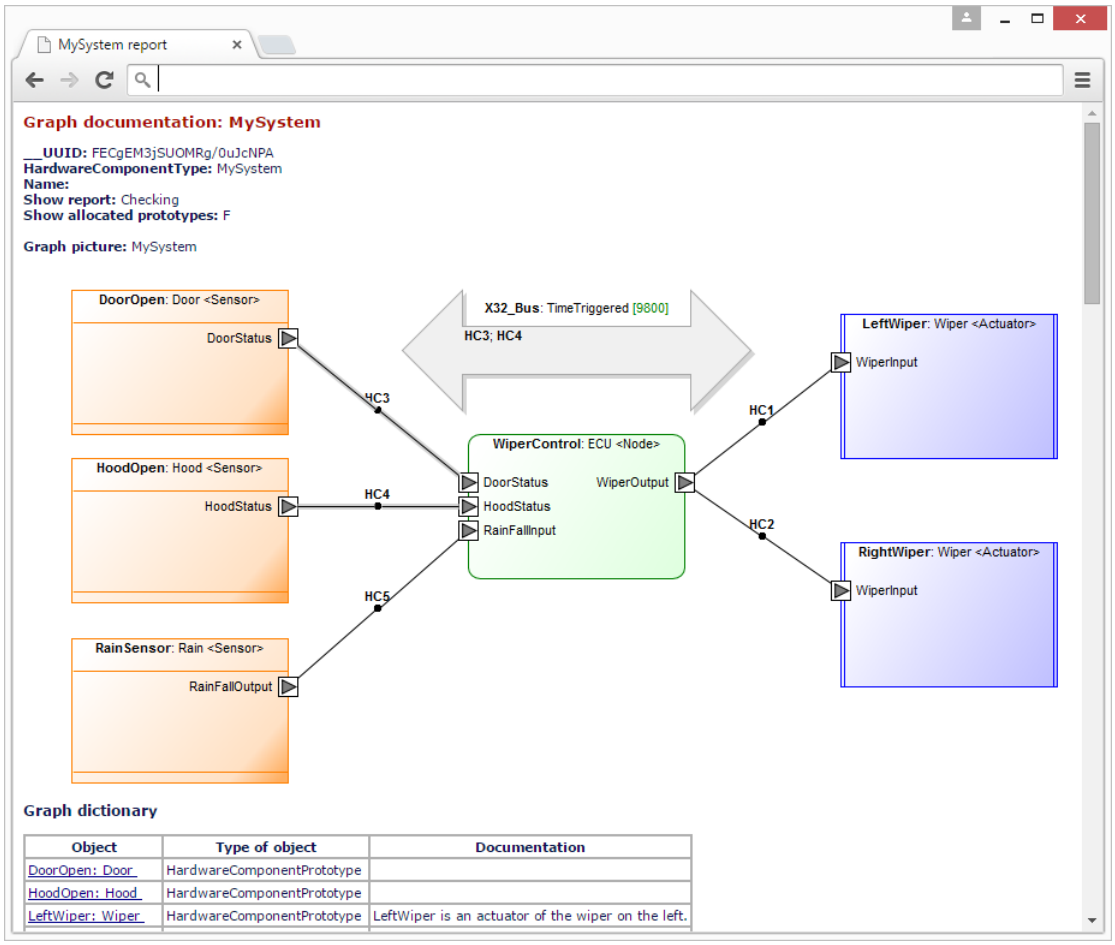


Figure 4-2. The results of the HTML generation

4.3.3 Listing unconnected ports

You may also produce a list of the unconnected ports in this model by running the generator **Graph | Generate… | Unconnected pins**. Generated list presents all those HardwareComponentPrototypes and their pins, which have not been connected. You may trace back to the original model element just by double-clicking the underlined item in the output.

4.3.4 Type usage

EAST-ADL follows type-prototype pattern: Each FunctionType may contain a number of prototypes and each prototype is based on some FunctionType. This hierarchy is shown in Browsers, like in Graph Browser and prototypes also show type name in their identifiers.

Since it is also useful to find out all prototypes using the given type, a report ‘TypeUsers’ provide a list of all prototypes based on the current type.

Report ‘TypeDeclarations’ provides a reference to the package hierarchy in which the Type is been defined. In EAST-ADL a concept ‘EAType’ is used to define all function types.

4.4 OTHER MODELING FUNCTIONS

MetaEdit+ also provides several other functions that this EAST-ADL tutorial does not describe. The functionality provided by MetaEdit+ to work effectively with models includes:

- Reusing model elements between designs (using selection tool via **Add Existing…**, and **Copy** and **Paste Special…** functions).
- Tracing among model elements (**Info…** function available from the pop-up menu of each model element).
- Refactoring models with **Replace…** function.
- Organizing and handling models into hierarchies, projects, and in multi-user environment for several engineers in a team working together with the models at the same time.

The Diagram Editor also supports editing features such as auto layout, views, different representations, zooming, grid, scaling, exporting to different formats. For a more detailed description of these Diagram Editor functions please see the MetaEdit+ User's Guide.

5 Languages of EAST-ADL

We have now successfully created and documented the first version of the hardware architecture. It is, however, just a partial description of whole automotive electric system. EAST-ADL provides modeling support for other system aspects too, like for vehicle features, requirements, functions, errors, dependability and for several types of allocation. In the following we look some of these other modeling languages of EAST-ADL.

5.1 OPENING OTHER PROJECTS

The EAST-ADL repository includes examples of other kind of EAST-ADL models. Your models may be stored also in other projects or in the same EAST-ADL Examples project. To inspect existing projects, open the pop-up menu in the Main Launcher's Projects list on the left (see Figure 5-1) and choose **Open…** Then select the project and press **OK**.

To see the models in the browser you must first select the project as active in the project list. If you select several projects you will then see models from both of these projects. The default project selection on the bottom left corner shows that all new models will be stored to the '*EAST-ADL Examples*' project.

→ *Note that any changes made in the models stored in the repository will remain in the same project also after the changes. For more details of using different projects and reusing data between them please see MetaEdit+ User's Guide.*

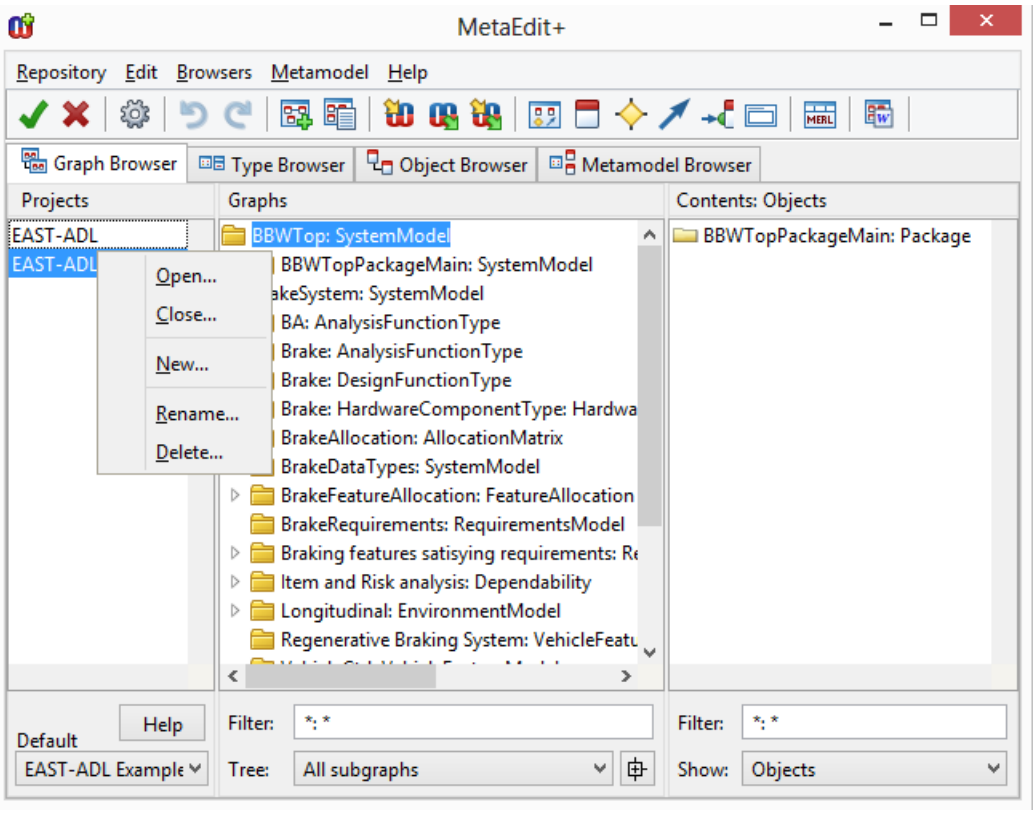


Figure 5-1. Several projects opened in the Main launcher

5.2 PACKAGES

EAST-ADL allows organizing the model elements into packages. A package may include other packages (as shown in Figure 5-2) creating a package hierarchy or other model elements. Engineers are free to decide how many packages and how deep package hierarchies they want to use. In the example below the package '*BrakeSystem*' contains seven packages and a system model.

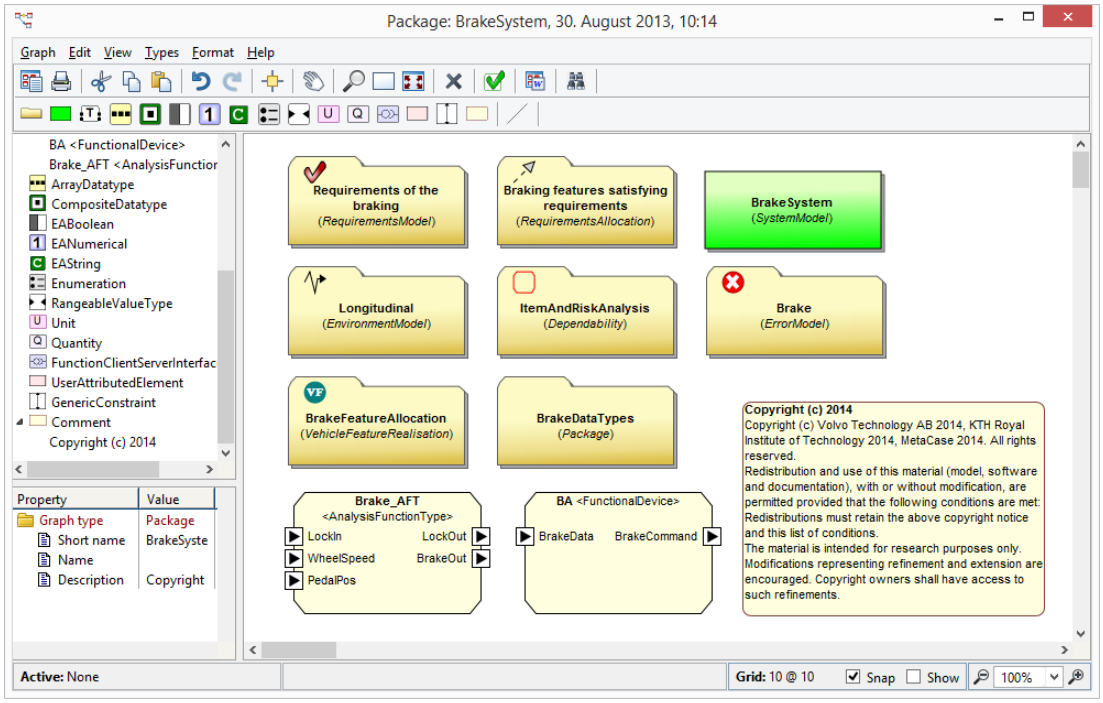


Figure 5-2. Package Diagram

Each package refers to different kind of model elements. To create a package and its hierarchy, add Package element to the canvas and create a subgraph for it. Model editing operations are the same as for hardware architecture described in Section 3.4.

In our example, there is one package containing all the data types of the brake system. Similarly there is a package for requirements of the brake system, a package for the environment, a package for feature allocation, a package for requirements allocation, a package for dependability and a package for error models. Finally, there is a system model containing the feature models, functional architectures and hardware architecture of the brake system. This last part, hardware architecture, we already specified in Chapter 3.

In real systems, the number of models and model hierarchies grow and MetaEdit+ provides a number of browsers to manage large specifications. All browsers enable navigating and accessing model elements using their respective view. For example, Object Browser in Figure 5-3 shows the same packages as the package diagram in Figure 5-2.

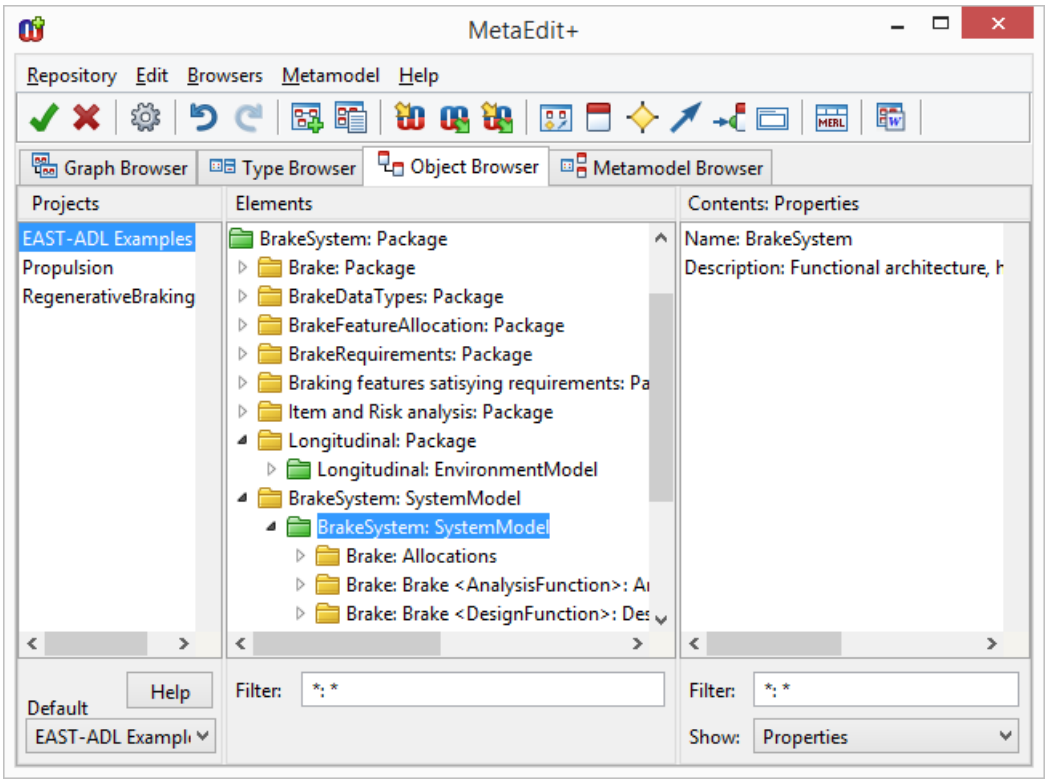


Figure 5-3. Package Hierarchy in Object Browser.

Browsers allow navigating all model elements, like in the Figure 5-3 a System Model is opened and its content in terms of Allocations, AnalysisFunction, and DesignFunction is shown.

Object Browser allows tracing among model elements (via **Info…**) and apply filters to reduce the large hierarchies by their model data and by the type of model data. For example, filter '*MySys* : Hard**' would show in the browser only the hardware architecture we already created along with the possible hierarchy it belongs. Other browsers, namely Graph Browser and Type Browser allow accessing and navigating in models using different views. For more details, see MetaEdit+ User's Guides.

5.3 SYSTEM MODEL

System model collects the feature models, architectures and their allocations for a given system. A sample of System model is illustrated in the Figure 5-4.

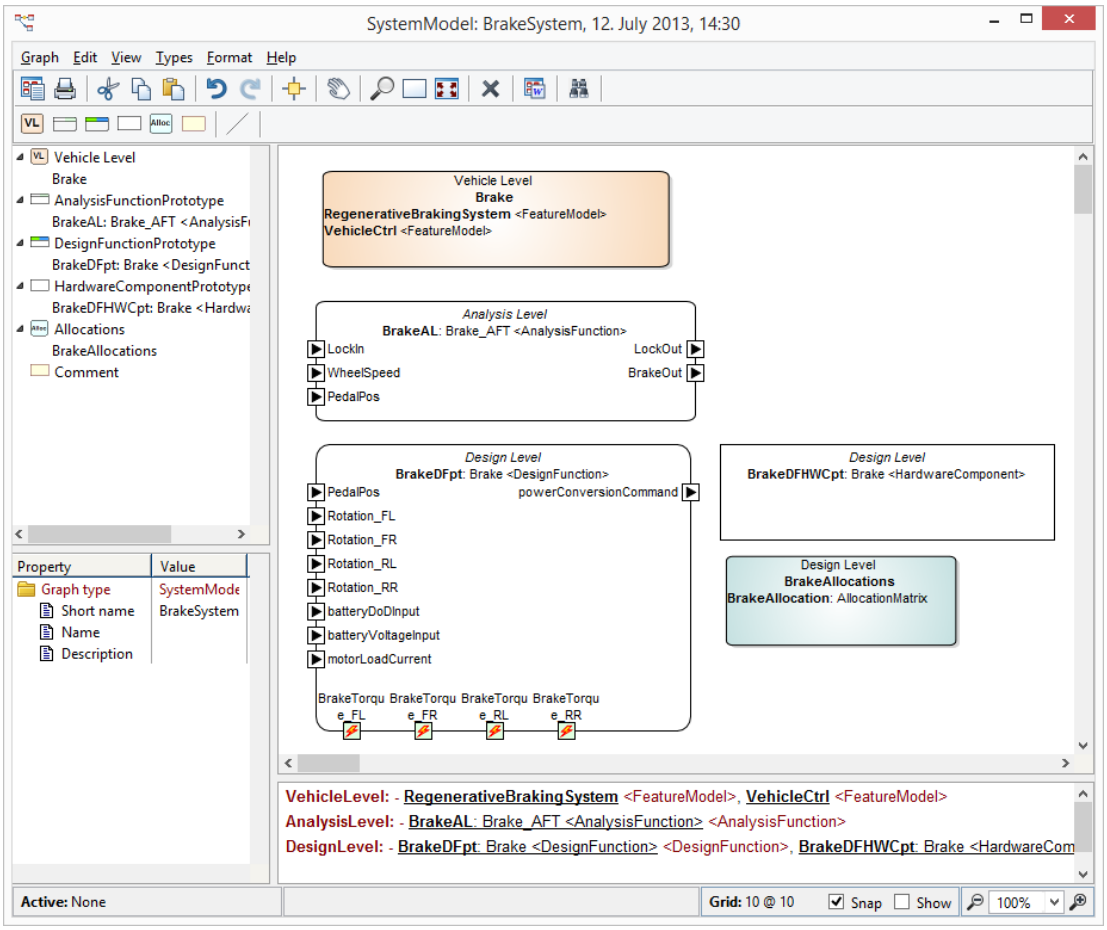


Figure 5-4. SystemModel

Elements in the system model, organized into vehicle level, analysis level and design level, are described in subgraphs, thus system model is creating a hierarchical structure similar to packages.

A vehicle level contains a number of feature models (Section 5.4), analysis level contains the main analysis function (Section 5.5), design level contains the design function (Section 5.6), hardware component (Sections 3.2 and 5.8), and allocations (Section 5.7) among the prototypes of design function and hardware component.

5.4 VEHICLE FEATURE MODELS

Feature models define the commonalities and variabilities of the product variants within the product line. They may be created from different perspectives, such as from the technical point of view or from the end-customer perspective. A sample of feature model is illustrated in Figure 5-5.

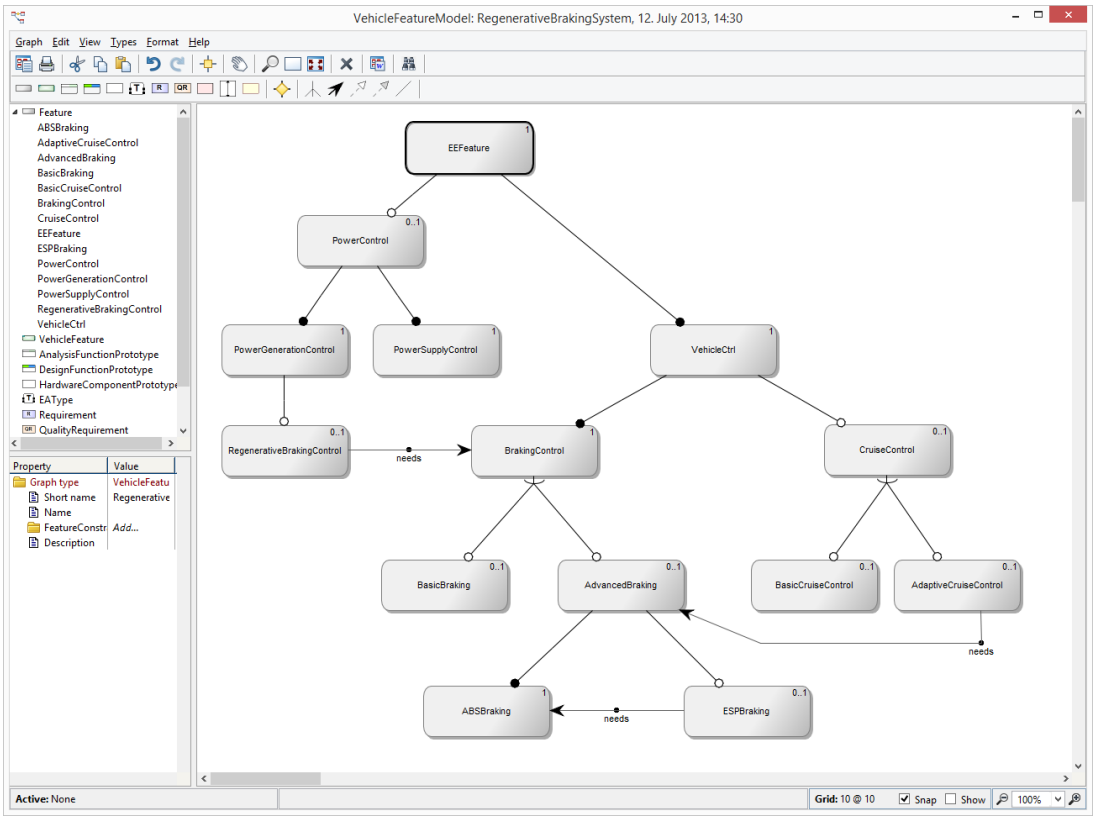


Figure 5-5. Example of vehicle feature model

5.4.1 Feature modeling concepts

The description of the modeling concepts is available from the **Help | Graph Type** in the Diagram Editor. Similarly to hardware architecture modeling they are visible in the toolbar and can also be seen from Types menu. Figure 5-6 shows the contents of the Types menu: The modeling objects and relationships with their notational symbols.

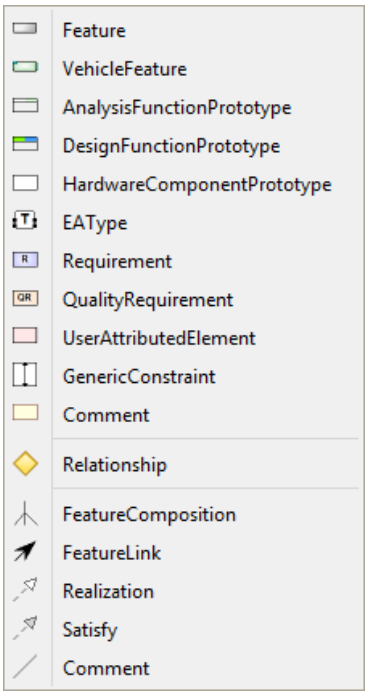


Figure 5-6. Modeling concepts for vehicle feature modeling

Main concepts for feature modeling are '*Feature*' and '*VehicleFeature*' as well as connections between these using '*FeatureComposition*' and '*FeatureLink*' relationships. Most important part is the characteristics of features describing their relationships as well as the actual and required binding times.

The vehicle feature models may also include function types and prototypes as well as requirements to map them to the feature models. '*Realization*' and '*Satisfy*' relationships can be applied with these concepts. Please see **Help | Graph Type** for the details of the feature modeling concepts.

5.4.2 Feature model checking and documentation

Most of the rules of feature modeling are supported directly in the language definition (aka in the metamodel). Few remarks on the rules could be mentioned: EAST-ADL supports multiple root features and MetaEdit+ shows the root features with thick border: all the features that do not have child link role are visualized as root features. Every feature may be only at one child role.

Feature model may also include requirements and functions specifying how they satisfy or realize the features. This information can be later traced and reported with different reports, like Item realization or how different features realize the requirements.

Similarly to specifying hardware architecture, a number of predefined checking and documentation reports are available from **Graph | Generate…** menu. You may also produce Word document by pressing the **Doc** button in the toolbar of Diagram Editor.

You may also trace all the Features and their connections in opened models by running the generator 'Trace all features'. Output will show all the Features and VehicleFeatures and their connections. This report, like others in MetaEdit+, allows you to trace to the original element by double-clicking the underlined elements.

5.5 FUNCTIONAL ANALYSIS ARCHITECTURE

Functional analysis architecture specifies a solution without concerns about the implementation restrictions. The analysis model is thus a logical representation of the system to be developed. It does not distinguish if functionality is enabled via hardware or software or how the communication among the functions is implemented. A sample of functional analysis architecture is described in Figure 5-7.

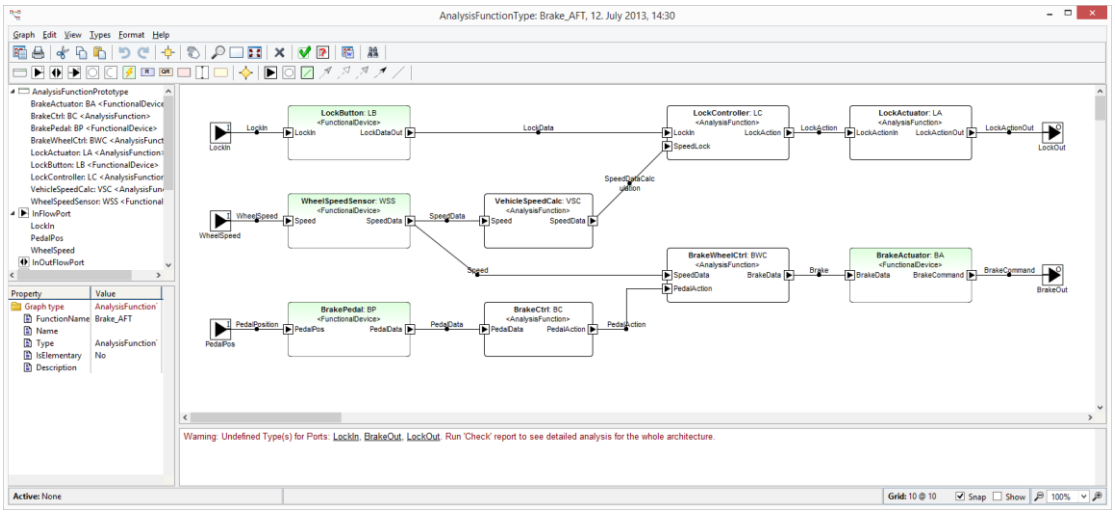


Figure 5-7. Example of functional analysis architecture

5.5.1 Functional architecture modeling concepts

Figure 5-8 shows the contents of the Types menu: The modeling objects and relationships with their notational symbols. A more detailed description of the modeling concepts is available from the **Help | Graph Type** in the Diagram Editor.

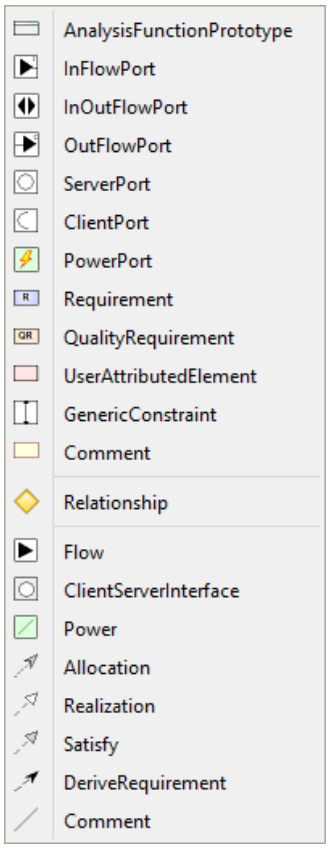


Figure 5-8. Modeling concepts for functional analysis architecture

Similarly to hardware architecture, the functional architecture is organized into type-prototype hierarchy. We can inspect the graph hierarchy similarly already done with the Hardware Architecture: by simply Ctrl + double-clicking the function prototype specified in the diagram. Alternatively choose from the pop-up menu of the element **Open Subgraph**.

5.5.2 Model checking

As all models in MetaEdit+, functional analysis architecture is checked based on the rules of the metamodel at modeling time. In addition, some of the rules are implemented as separate model checking available from the toolbar of the Editor as well as showing warnings by annotating models. These work similarly to the ports of functional architecture as the checking of pin types in hardware architecture models. You may also see the list of unconnected ports by running the Unconnected ports report from the toolbar.

5.5.3 Type users

While EAST-ADL does not specify the link from type to the prototypes it is typing, MetaEdit+ provides 'TypeUsers' report for that. This is useful when type is edited and we want to know all the prototypes the change in type influences to.

5.5.4 Tracing the FunctionPrototypes

You may also trace all the FunctionPrototypes and their connections by running the generator 'Trace all functions'. Output will result all the FunctionPrototypes and their connections and you may trace to the original element by double-clicking the underlined elements.

5.5.5 Creating error models

For error analysis, model-to-model transformation report is available: 'Produce ErrorModel' creates initial dependability and error model based on the selected analysis architecture.

5.6 FUNCTIONAL DESIGN ARCHITECTURE

Functional design architecture describes the concrete functional definition. In the design level the analysis level architecture is abstracted into software and hardware. The design level, however, is not mapped to any specific software architecture platform yet, although with generators can be mapped to them (see Section 5.7 for AUTOSAR mapping). A sample of functional design architecture is described in Figure 5-9.

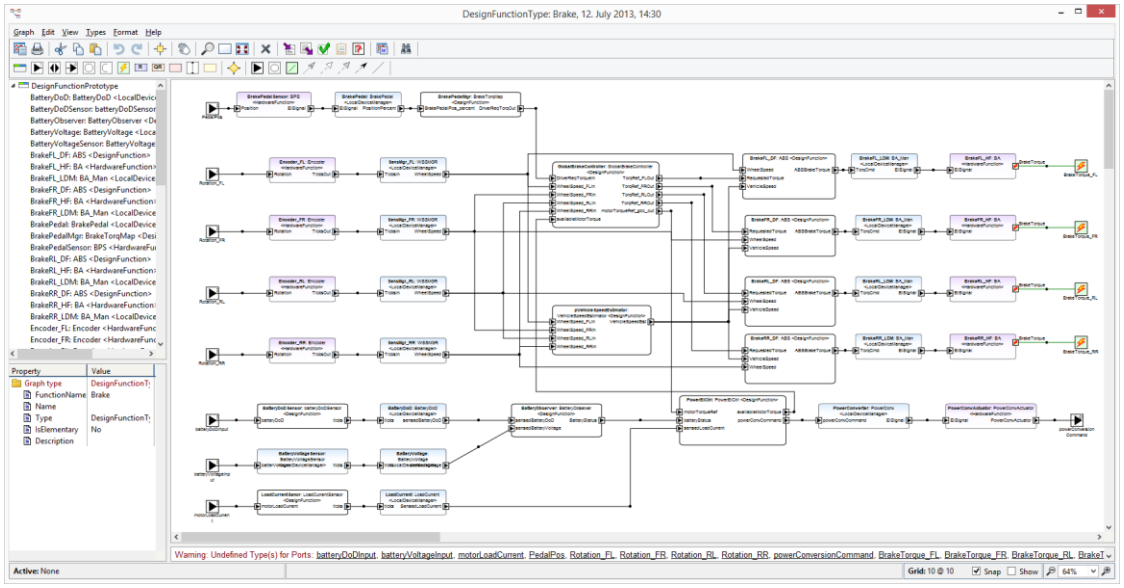


Figure 5-9. An example of functional design architecture

5.6.1 Modeling concepts for functional design architecture

The modeling concepts of the functional design architecture are largely the same as for the functional analysis architecture, only the '*AnalysisFunctionPrototype*' is replaced by '*DesignFunctionPrototype*'. Thus the modeling concepts are almost the same as illustrated in Figure 5-8. The notational conventions for prototypes vary depending on its type (been defined in the subgraph).

5.6.2 Model checking

Functional design can be checked using the model checking reports and showing errors as model annotations just like in hardware architecture and analysis function types. You may test these by running checking reports and using annotation with the sample models included in the '*EAST-ADL Examples*' project.

5.6.3 Generating Simulink files

Simulink files can be generated from functional design architecture by selecting the **.MDL** generator icon from the toolbar. If you have Simulink installed the produced models will be opened there. This generator and all the others can be defined and modified using the generator system of MetaEdit+ Workbench, which can be accessed from **Graph | Edit Generators…**.

5.6.4 Checking consistency with existing Simulink files

If Simulink models are already available, been generated or made without considering the planned architecture, 'CheckWithSimulink' generator provides consistency checking. By pressing the icon 'CheckWithSimulink' in the toolbar the architecture design is verified against an existing implementation to report any differences. This offers good support for comparing the planned architecture with the actual implementations provided by various suppliers. The comparison checks the architecture-level interfaces of individual modules, like ports, their prototypes and hierarchies. By default the Simulink models are expected to be stored to the default directory (/reports/) but can be changed in the Generator script.

5.6.5 Importing Simulink files

In case Simulink models are already available they can be used to create initial architecture models. Report named as 'Import Simulink File Hierarchy' selects a root .mdl file containing a system, extracts the relevant parts like model references for type-prototype hierarchy and ports, and creates the architecture model. While reverse engineering like this may not lead to the ideal architecture design, companies often find it an appropriate solution to lower the effort needed to modernize development practices.

5.6.6 Type users

While EAST-ADL does not specify the link from type to the prototypes it is typing, MetaEdit+ provides 'TypeUsers' report for that. This is useful when type is edited and we want to know all the prototypes the change influences.

While Graph Browsers shows the type-prototype hierarchy, report 'TypeDeclaration' provides a direct reference to the package in which the type is been defined.

5.6.7 Tracing the FunctionPrototypes

You may also trace all the FunctionPrototypes and their connections by running the generator 'Trace all functions'. Output will result all the FunctionPrototypes and their connections and you may trace to the original element by double-clicking the underlined elements.

5.6.8 Creating error models

For error analysis, model-to-model transformation report is available: 'ProduceErrorModel' creates initial dependability and error model based on the selected design function.

5.7 FUNCTION AND HARDWARECOMPONENT ALLOCATION

AllocationMatrix specifies how design functions are allocated to hardware components. This is done by joining the prototypes and function connectors of functional design architecture to prototypes and PortConnectors of hardware architecture.

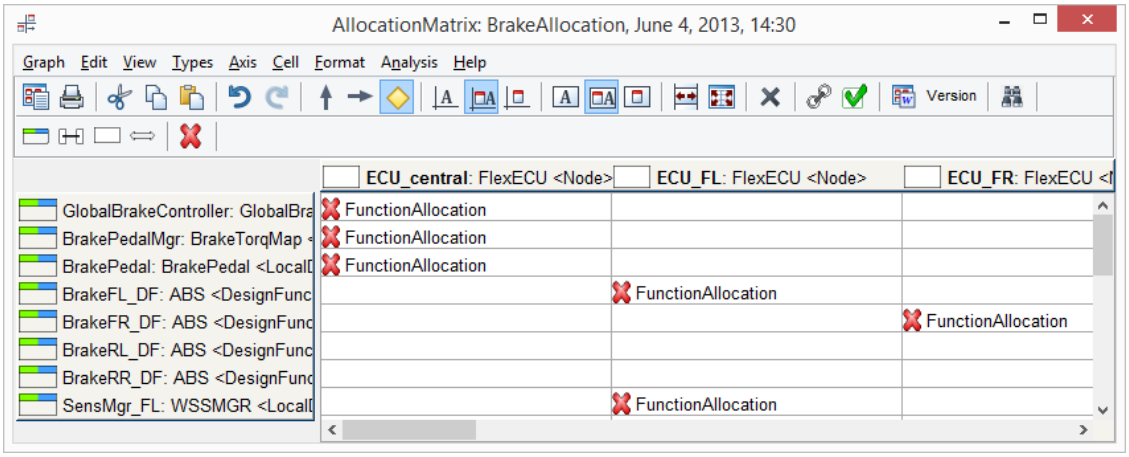


Figure 5-10. Allocation matrix

For this matrix a number reports can be produced, like reporting the current status of allocations or producing allocation for the software architecture, like for AUTOSAR. These reports are available from toolbar or from Graph | Generate … menu.

5.8 HARDWARE DESIGN ARCHITECTURE

Hardware design architecture specifies the resource platform with nodes, sensors, actuators, electrical components and HardwarePortConnectors. It also shows the physical topology of electrical elements and their connectors.

Chapters 3 and 4 already described the language for specifying hardware architectures and we describe here one of the more advanced capabilities that utilizes the previously defined allocation: By choosing 'Show allocation' in the HardwareComponentType (selected in the bottom left in the editor) the hardware architecture is annotated by showing the allocated design functions within the symbol of hardware component it has been allocated to.

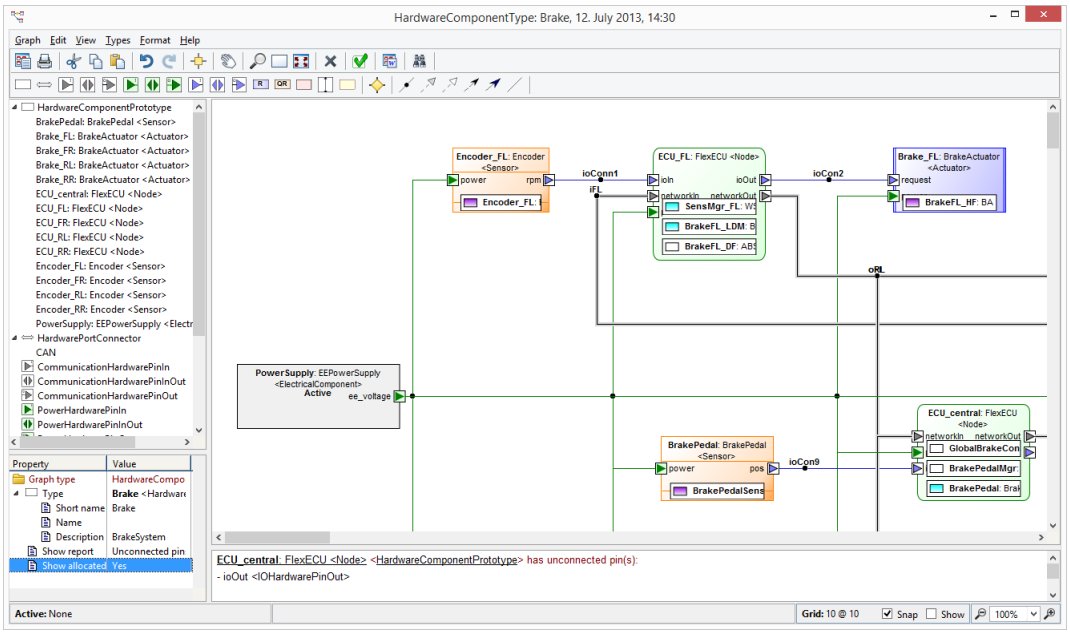


Figure 5-11. Visualizing allocation made in hardware architecture design

5.9 OTHER EAST-ADL EXTENSIONS

In addition to the core sublanguages of EAST-ADL there are also a number of extensions for specifying:

- data types
- requirements and use cases
- dependability
- timing modeling
- error modeling
- environment
- verification&validation
- variation
- behavior

You may inspect all these extensions via the sample models being included in the repository or create your own models.

6 Exporting and importing models in EAXML format

An exchange of EAST-ADL model among different tools is possible via XML file exchange. The XML file must follow the EAXML schema, current version being EAST-ADL_V2.1.12.xsd. EAXML import and export covers the nominal architecture modeling described in previous sections.

6.1.1 EAXML export

EAXML file can be generated from EAXML graph. This graph can contain any number of packages creating a package hierarchy with Package Diagrams. To export EAXML, create the package hierarchy and press **Export to EAXML** generator icon in the toolbar or from the list of generators. As a default the output in EAXML format is generated to your *...\My Documents\MetaEdit+ 5.1\reports* subfolder.

The file can be imported to other EAST-ADL tool following the same EAST-ADL version and schema. If you want to test and validate the generated file (`*.eaxml`), save also the latest EAST-ADL schema (EAST-ADL_V2.1.12.xsd) to the same folder where the generated `.eaxml` file exists.

→ *EAXML export includes all the elements within the package hierarchy. If packages refer to elements outside the package hierarchy, e.g. reusable types, then EAXML export includes a reference path to them but not their content.*

6.1.2 EAXML import

With MetaEdit+'s native formats (`*.mxm` (XML format) or `*.mec` (binary file format)) MetaEdit+ is able to export and import all EAST-ADL models to any other MetaEdit+ tool – including not only the conceptual model data but also the model representation as well as metamodels (for details see Chapter 7).

MetaEdit+ is able to import EAXML files – given that they are first converted to the format of MXM. To enable automated EAXML import a XSLT processing script has been developed. Figure 6-1 illustrates the process to import EAXML file into MetaEdit+.

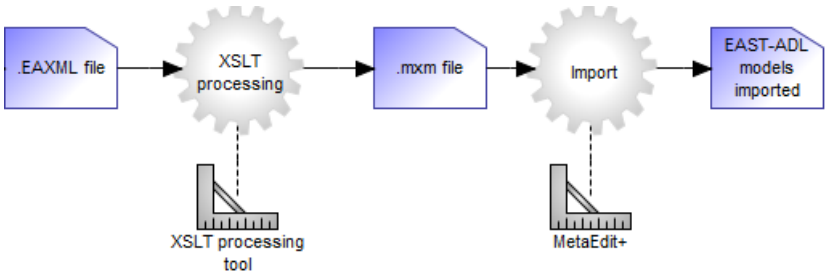


Figure 6-1. EAXML processing

XSLT transformation script (EAST_ADL_import.xslt) is delivered together with the EAST-ADL repository. Place the XSLT script to the '…\MetaEdit+ 5.1\reports' subfolder.

To run the transformation you also need to have a XSLT 2.0 processing tool installed, e.g. Oxygen, AltovaXML, Saxon, Kernow or similar. The XSLT script ('*EAST_ADL_import.xslt)*' translates the `.EAXML` file into the `.mxm` file format to be imported into MetaEdit+.

In the next example scenario, it's assumed that Saxonica's Saxon XSLT transformation tool is installed into the default directory ('*C:\Program Files\Saxonica\...*'). In case you use some different transformation tool, edit the generator by opening Generator Editor for the generator named 'Import EAXML'. This generator can be found from _EAST-ADL graph: Open Generator Editor, select Generator | Change Graph Type… and choose '_EAST-ADL'. Next in Generator Editor select the **Import EAXML** generator from the top-left widget. See Figure 6-2 for the command line that may need update for your transformation tool.

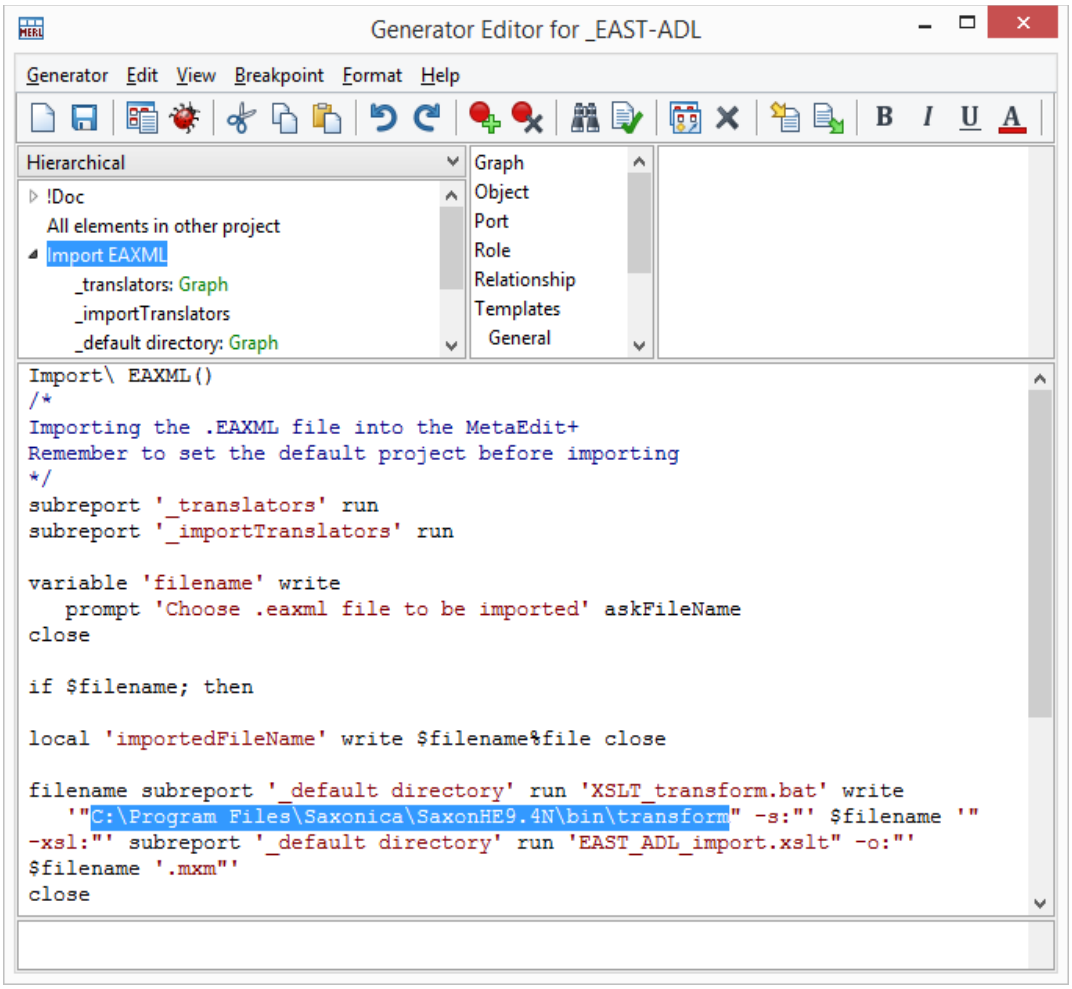


Figure 6-2. Setting the XSLT transformation tool

You can import EAST-ADL models (in eaxml format) by using the following instructions:

1. Set the default project in MetaEdit+ main launcher. Default project is the project, where all the new or imported models will be stored. Note that actually the storing is made only if you press **Commit** after having done the import.
2. Run a generator **Import EAXML** from any graph and select `EAXML` file to be imported.

Imported EAXML models are shown in browsers similarly to other EAST-ADL models. By double-clicking the imported elements you can see their properties. When opening a graph the first time select **Create new Diagram** as a representation format.

7 Language updates

EAST-ADL is an evolving language with multiple language extensions. MetaEdit+ tool supports language evolution and extension allowing you to use models made with earlier versions of the language too. If you updated to a newer version of the language, also your models and future modeling operations will follow the latest language version.

With MetaEdit+ Workbench you can extend EAST-ADL, or any other modeling language, to meet your needs. You can thus freely modify the concepts and related rules, notation as well as generators for producing model checking, documentation, metrics or code. You may also modify tool functionality by changing the dialogs, toolbars, or even access the models via API.

7.1 IMPORTING NEW EAST-ADL VERSION

MetaCase will also continue to improve the EAST-ADL language based on feedback from users. The language updates that extend the current EAST-ADL version will be made available as .met files. These files store the language definition, including concepts, rules, notation and generators, into a file that can be imported to MetaEdit+ repository.

Please note: If you have made own modifications to the existing EAST-ADL language definition and import the new language version coming from MetaCase, your modifications to the EAST-ADL language will be lost. Your models will still remain, but you are not anymore able to create new ones based on your modified language version. Contact info@metacase.com to solve that!

You can import these language extensions via .met file by using the following instructions:

1) Save the updated language definition provided as binary `.met` and optional `.mxs` files to your hard disc
2) Take a backup copy of your current EAST-ADL repository
3) Start MetaEdit+ and login to the EAST-ADL repository and open the 'EAST-ADL' project
4) Press **Import** button in the Main launcher
5) Select the `.met` file from your hard disc and press **Open**
6) When MetaEdit+ asks project for the new types, select 'EAST-ADL' and click OK
7) After the import is finished, a dialog opens saying: "Import finished." Click **OK**.
8) Test the models and generators, and if everything looks OK you may **Commit**. If there are errors or problems, select **Abandon** so the imported metamodel changes won't be saved.
9) If library symbols require update, you need to import also the `.mxs` file coming along with the .met file. Just follow the same procedure from the step 4 onwards, but in the phase 5 select the `.mxs` file to be imported.

If everything was OK and you committed the transaction, .met file is not needed anymore. More details on importing can be found the System Administrator's Guide: section 2.4.1 "Importing types and models".

7.2 READING NEW UPDATED GENERATORS FROM FILES

Typically EAST-ADL updates in the metamodel come with updated generators too. Sometimes you may want to read only updated generator or import totally new generator (e.g. for AUTOSAR, HIPHOPS, document generation etc.) without the metamodel. Generators are always defined specific to a given language (graph type in MetaEdit+ terms), because the generator uses the data structures of the graph type in its definition. So before importing any generator, please check first the correct language type for it.

Reading the new generator scripts can be done by selecting the Generator Editor's menu **Generator | Read from File...**. This opens a dialog from where the user selects the generator definition file(s) to be imported. When reading from a file, you are normally warned if the generator would have the same name as an existing one. Also, the generator is not actually saved, but is left in the text area, ready to be saved explicitly or via a prompt when you select another generator definition or close the Generator Editor. If you selected several generators to be imported at the same time each generator is saved automatically without further interaction with user.

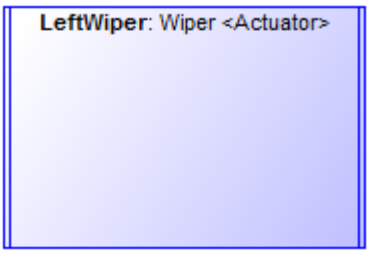
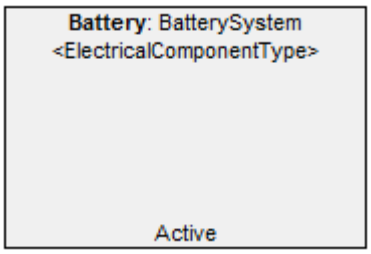
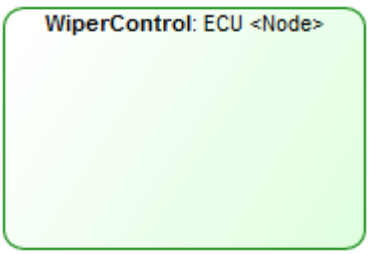
8 Conclusion

We have now walked through a set of examples on using automotive architecture design language EAST-ADL. To learn more about EAST-ADL you are most welcome to create further models, inspect the other sample models available in 'EAST-ADL Examples' project. You will also find more information about the EAST-ADL language from http://www.metacase.com/solution/east-adl.html.

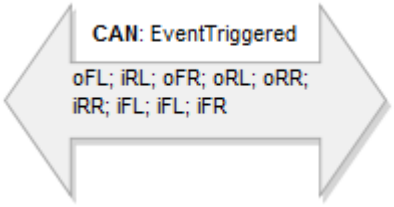
To learn more about MetaEdit+ you are encouraged to study MetaEdit+ User's Guide that comes with the installation of MetaEdit+. The installation package includes also other manuals for system administration (especially for multi-user use) and for language creation using MetaEdit+ Workbench. Our web pages at http://www.metacase.com provide further information: there you will find user references, sample languages, discussion forums, downloadable white papers, and FAQs. If you have any questions about MetaEdit+, do not hesitate to contact us!

9 Appendix A: Hardware architecture modeling language

The concepts for hardware architecture modeling include:

Language concepts	Representation of the concept
The Actuator is the element that represents electrical actuators, such as valves, motors, lamps, brake units, etc.	
PowerSupply denotes a power source that may be active (e.g., a battery) or passive (main relay).	
The Node element represents an ECU, i.e. an Electronic Control Unit, and an allocation target of FunctionPrototypes. Node represents the computer nodes of the embedded electrical/electronic system.	
Sensor represents a hardware entity for digital or analog sensor elements.	

HardwarePortConnector represents a logical connection that carries data from any sender to all receivers.



Hardware connectors represent wires that electrically connect the hardware components through its ports. Color of the line depends on the type of the hardware connector (Communication, IO or Power)

IOHardwarePin represents an electrical connection point for digital or analog I/O. Blue triangle and the letter indicate the direction of the flow. Relationships between these pins are shown with solid blue line.

The CommunicationHardwarePin represents the hardware connection point of a communication bus. Grey triangles and the letter indicate the direction of the flow. Relationships between these pins are shown with solid black line.

PowerHardwarePin represents a pin that is primarily intended for power supply, either providing or consuming energy. Green triangle and the letter indicate the direction of the energy. Relationships between these pins are shown with thick green color.

Comment object helps to document and develop the language further. You may connect the Comment to any other object and describe your question or idea, or apply the comment to highlight a particular aspect.