Telelogic Tau 3.1

UML Tutorial

Terms of use

Trademarks

How to contact Customer Support

The **Help Desk** is the single point of contact for all support related communication regarding the Telelogic Tau products:

- Telelogic Help Desk at our offices.
- Distributor Help Desk if you have received the Telelogic products from a local distributor.

Contact the Help Desk if you:

- **Need to obtain a license key or other licensing materials.**
- Have questions, suggestions, or problems related to the installation, setup or performance of the Telelogic Tau software.

  When contacting Help Desk, try to provide the following information:

  – Product version
  – Operating system
  – Module or tool
  – Third-party compiler, if applicable (for example Borland C++ 5.0, Sun cc, or gcc)

You can find address to the Telelogic Help Desk at the Telelogic home page: http://support.telelogic.com/en

Table of Contents

Introduction

Purpose of this tutorial

The purpose of this tutorial is to make you familiar with Tau and the UML language. This tutorial primarily addresses persons with no or little experience of Tau, but with knowledge of the basic concepts of UML and state machines.

You will design a coffee machine, starting with the initial modeling and ending with a simulation of your design.

This tutorial provides step-by-step instructions on how to start up a workspace and create a project from scratch. You will produce a UML specification. You will step-by-step complete the design to the state where you can simulate it in the Model Verifier. The instructions in this tutorial should be complete to let you perform all steps. More information on the various work procedures can be found in the Tau On-line help.

In the margin of the textual description you will sometimes find pictures of toolbar buttons. The purpose of this is to make it easier for you to identify the button that are referred to in the text. The button pictures will in most cases only be present the first time you are instructed to use a button. These buttons belongs to various toolbars which may be activated or deactivated by the user, therefore it can happen that a button referred to is not visible in the current set of toolbars. The toolbar must then first be activated. This can be done in the Customize dialog which is opened from the Tools menu.

When you are instructed to insert symbols in diagrams make sure that the diagram is active by clicking in it. In a recently opened diagram it is not always possible to directly click on a toolbar to insert symbols, first click anywhere in the diagram then select the desired symbol from the toolbar.

The Coffee Machine

Behavior of the coffee machine

The coffee machine in the example provides the user with a cup of coffee or a cup of hot water (for tea), given that the customer has inserted the required amount of money. The coffee machine can handle coins of the values 5 and 10, where 5 is the price for a cup of tea while a cup of coffee costs 10. The following holds:

- If a coin with the value of 10 is inserted and the Coffee button is pressed, the customer receives a cup of coffee.
- If a coin with the value of 10 is inserted and the Tea button is pressed, the customer receives a cup of hot water plus change.
- If a coin with the value of 5 is inserted and the Coffee button is pressed, the money is returned.
- If a coin with the value of 5 is inserted and the Tea button is pressed, the customer receives a cup of hot water.

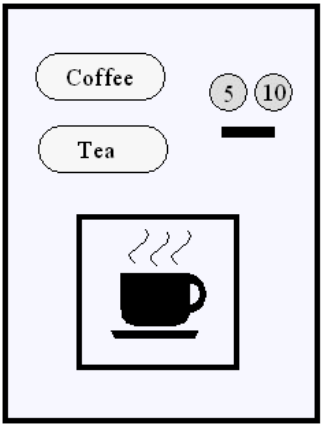

Figure 1: The coffee machine

Model and Diagrams

General

A diagram is a view of a UML model showing a set of elements and their relations. In this tutorial you will use the capabilities of the tool to work both directly in the model and through diagrams. You will work with different diagrams where each of these diagrams present a certain view of the model. When a new entity, for example a class, is drawn in a diagram, it becomes part of the model. To present different views, the same class may be drawn again in the same diagram or in a different diagram. The information stored in the model is the sum of all descriptions made of an entity.

Deleting entities

The distinction between the model and the diagrams must be kept in mind when designing in Tau. As stated above, an entity, for example a class, is included in the model as soon as it is added to a diagram. **Deleting a class from a diagram does not mean that the class is removed from the model!** The reason for this is that the same class could be used in other diagrams. It is however possible to delete an entity from the model. By right-clicking an entity there will appear a context-sensitive shortcut menu. From this menu **Delete from Model** can be clicked instead of **Delete**.

Diagram element creation toolbar

The Diagram element toolbar is only active when the diagram is used. Click in the diagram to activate the toolbar. Buttons in the **Diagram element creation** toolbar are normally handled so that you click on the toolbar button, then click the diagram to position the entity controlled by the button.

Toolbar quick-buttons are in sometimes context-sensitive, so the result may depend on selections in the current diagram and the relation between the selection and the button entity. It is often possible to access a shortcut menu (right-click) to get assistance from the model semantics. An example of this is when you draw messages in sequence diagrams, you click on the Message Line button, click on the sender lifeline and then right-click on the receiver lifeline and the shortcut menu will have a drop-down box with all signals in the current scope.

User Interface

General

The Tau user interface main areas:

1. the Workspace window
2. the Desktop
3. the Output window

In this tutorial you will sometimes have other areas active, for example watch windows during testing. It is also possible to drag an area to alter its position within the window or even to position it outside the main window of the user interface.

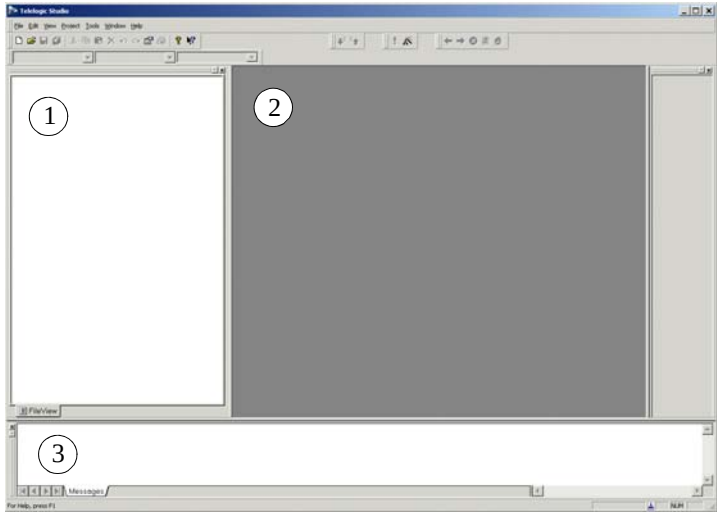

Figure 2: The user interface

The Workspace window

The Workspace window presents the entities contained in a model. Different views are available for displaying different kinds of information. The **File View** shows all elements that are represented as files. The **Model View** con-

tains all UML elements. This is the view to select when adding diagrams or working directly in the model. There is furthermore a view called **Instances** which is only active when simulating your system with the Model Verifier.

The Desktop

The Desktop is the area where diagrams and documents appear when opened. This is where you work with your model as it is viewed from different diagrams.

The Shortcut bar

The Shortcut bar, which is optional, gives you the possibility to put any frequently used toolbar in a special area, or to create shortcuts to your own scripts.

The Output window

The Output window is used for logging events and displaying errors and warnings.

The Shortcut menu

By right-clicking an entity you will get a shortcut menu. This menu will frequently contain context-sensitive commands.

Workspace

General

A workspace is your personal working area, where you can work in separate projects but also include files that are not related to a specific project. You can define more than one workspace, but you can only work in one workspace at a time. You cannot share a workspace with other users. The information contained in a workspace is stored in a text file with the extension *.ttw.

Creating a workspace

You will now create a workspace for the development of the coffee machine example. Do the following:

1. Start Tau.
2. On the **File** menu, click **New...** The dialog that appears contains four tabs: File, Project, Template and Workspace.
3. Select the **Workspace** tab.
4. Name the workspace “Tutorial” and select the location where the workspace file (Tutorial.ttw) will be stored. Click **OK**. Your new workspace appears in the **File View** of the Workspace window.

The next step is to add a project.

Projects

General

Using projects is a way of grouping the contents within your workspace. You can for example let your workspace “Tutorial” contain several different examples, one being this coffee machine, using one project for each example. Diagrams and documents can be moved between projects. A project is not individual and can therefore be shared between users. The information contained in a project is stored in a text file with the extension *.ttp.

Creating a project

You will now create a project for the coffee machine example. Do the following:

1. On the **File** menu, click **New...** Select the **Project** tab.
2. A dialog with a number of choices for creating various types of applications appear. Select **UML for Model Verification**.
3. Name the project “CMdesign” and select the location where the project files will be stored. Select **Add to current workspace** and click **OK**.
4. A second dialog appears, suggesting a name for the file representing your model and a location for this file. Make sure that **Project with one file and one package** is selected and click **Next**.
5. A third dialog appears, displaying the name of the file representing the project and the name of the file representing the model. Click **Finish**. Your project appears in the Workspace window.

Note

A plus sign (+) to the left of an icon in the Workspace window indicates that the icon is collapsed, i.e. more information can be displayed. To expand the structure for this icon, click the plus sign. An entire substructure can be expanded by selecting a collapsed icon in the Workspace window and pressing the multiplication key (*) on your numeric key pad. To collapse a substructure click on the minus (-) sign for its root icon.

6. Expand the tree structures. The **File View** displays the created files and the **Model View** displays an empty package. In the Model View you will also find information from the internal structures of the Tau representation of UML. This information is found in the packages called **Library** and **Predefined**.

Use Case Diagrams

General

Use case diagrams describe the relationships between use cases and actors for a system. In a use case diagram it is possible to group use cases with a subject frame.

Creating use case diagrams

You will now create a use case diagram for the use cases for the coffee machine example. These cases were described earlier, see "Behavior of the coffee machine" on page 2.

1. Make sure that the Workspace window displays the **Model View**.
2. Select your package **CMdesign** in the Model View. Right-click and from the shortcut menu point to **New** and then click **Collaboration**. Name the collaboration **UserDrinks**.

Note

*Renaming of model entities is done by selecting the element and pressing F2, **or** by selecting the element and clicking once on the name. This will open the name string for editing.*

You will use a collaboration in the model to encapsulate the use case diagram, and also later use cases represented as sequence diagrams.

3. Select the collaboration. Right-click and from the shortcut menu point to **New** and then click **Use case diagram** to insert a use case diagram.

An icon for the use case diagram appears in the **Model View**. The diagram is now open on the Desktop. Click in the diagram to make it active.

The available symbols are now high-lighted in a toolbar. When resting the mouse pointer on a symbol a tooltip appears, indicating the name of the symbol.

4. A symbol is inserted in a diagram by clicking the quick-button representing the symbol, then clicking the diagram on the Desktop. Place an **actor** in the diagram. Give the actor the class **Customer** (for this the class name shall be preceded by a colon according to the syntax). It is possible to give the actor a name, but it is not necessary.

The class Customer will not be bound at this stage in the design. This can be observed in two ways. First the type name is underlined with a red wave line (See Figure 3 on page 9) indicating a name binding error, secondly there will be some error messages in the Output window (Autocheck tab).

5. Click on the Use case symbol and place it in the diagram. Name the use case **MakeCoffee**. Select the use case and create an association line to the actor. To do this drag the leftmost (**Association line**) of the three line handles from the use case symbol to the actor. See Figure 3 on page 9.

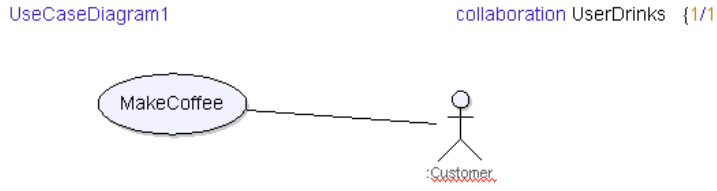

Figure 3: Use case and actor

6. Save your work.

You have now completed a use case diagram containing one of the possible use cases for this system. You will add some more use cases to this diagram later when working with sequence diagrams.

Class Diagrams

Class diagram

Class diagrams describe the types of objects that a system consists of, and the relationships between them. They also show attributes and operators of the classes.

Creating class diagrams

You will now use class diagrams for modeling the coffee machine example.

1. Make sure that the Workspace window displays the **Model View**.
2. Select your package **CMdesign** in the Model View. Right-click and from the shortcut menu point to **New** and then click **Class diagram**.
3. An icon for the class diagram appears in the **Model View**. The diagram is now open on the Desktop. Name the diagram “DomainModel“.
4.  Use the **Class symbol** quick-button to add two classes, click on the button, then click in the diagram to position the class. Name the classes **CoffeeMachine** and **Customer**. These classes represents the coffee machine and a prospective user, see Figure 4 on page 10.

Figure 4: The classes CoffeeMachine and Customer

Note

When class customer is created any errors concerning the actor type in the autocheck tab are resolved.

Active or passive classes

A class can be either active or passive. An active class contains behavior, while a passive class merely contains definitions. An example of a passive class is a data type. Your class CoffeeMachine will contain behavior, and must therefore be set to active. Do the following:

1. Right-click the CoffeeMachine class symbol.
2. On the shortcut menu that appears, click **Properties**. The properties dialog opens and displays the properties for class CoffeeMachine. In **Filter** select Class.
3. Select the **Active** check box. You may leave the dialog open.

Note

The properties dialog can be opened by pressing ALT + Enter.

An active class is displayed in the diagram with double vertical border lines, see Figure 5 on page 11.

Figure 5: Active and external classes

Informal class

The class Customer is an entity which is not a part of the coffee machine design itself. You will give it the stereotype **informal**. By applying the informal stereotype no code will be generated for this class when you will simulate your model. Do the following:

4. Right-click class Customer. On the shortcut menu that appears, click **Stereotypes...** .
5. Select the **TTDPredefinedStereotypes:informal** check box. The stereotype is activated as soon as the dialog is closed.
6. Right-click the class symbol for Customer and select **Active**.

Decomposing a class

You will now refine your model by adding two new classes representing the controller part and the hardware part of the coffee machine. **Controller** will contain the logic for the system, while **Hardware** will simulate the hardware behavior.

When adding these classes, different methods will be used. You will add the classes directly to the model and then use some modeling features of the editor. Do the following:

1. In the Model View of the Workspace window, locate package **CMdesign**. Right-click package **CMdesign**. On the shortcut menu, point to **New** and click **Class**. A class icon appears in the Model View.
2. Name the class **Controller**.
3. In the Model View, select package **CMdesign**.
4. Right-click package **CMdesign**. On the shortcut menu, point to **New** and click **Class**. A class icon appears in the Model View.
5. Name the class **Hardware**.
6. Select the class Hardware in the Model View. Right-click class Hardware. On the shortcut menu, change the properties to make Hardware to an active class. Do the same for class Controller.

Note

Observe that you do not need to go to the properties dialog to set a class to active. The shortcut menu contains some context-sensitive choices and one of these is controlling whether a class is active or not.

Component diagram

Component diagrams are closely related to and can be considered a subtype of class diagrams. Component diagrams describe the identifiable and possibly replaceable components of a system. Just like a class diagram the component diagram can also show ports and interfaces of the classes.

Creating component diagrams

You will now use a component diagram for further modeling of the coffee machine example.

1. Make sure that the Workspace window displays the **Model View**.
2. Right-click class **Controller**. On the shortcut menu, point to **Create Presentation**. In the New Symbol tab of the dialog click on the table row with **New ComponentDiagram**.
3. An icon for the diagram appears in the **Model View** and the diagram is opens on the Desktop. Name the diagram "**ControlComponents**".
4. Drag and drop the Hardware class icon from the Model View into the open component diagram (**ControlComponents**) on the Desktop.
5. Save your work.

Figure 6: Components for CoffeeMachine

Your component diagram should now look as depicted in Figure 6 on page 13.

Signals

General

UML has a specific class symbol for defining signals. This symbol is one of the predefined stereotypes, indicated by the `<<signal>>` heading. All signals used in the UML model must be defined. A signal defined on a certain level can be seen and used by all entities on lower levels.

Defining signals

You will now define all signals needed to implement the behavior of the coffee machine. You will use a class diagram to draw the signal definitions in. Do the following:

1. Select your package **CMdesign** in the Model View. Add a class diagram by right-clicking and from the shortcut menu point to **New** and then click **Class diagram**. The diagram appears in the Model View of the Workspace window.
2. Name the diagram “Signals”. The diagram is now open on the Desktop.
3. From the toolbar, select a Signal symbol and place it in the diagram. Define the signals (one Signal symbol per signal) from the table below.

Note

Press and hold CTRL when placing a symbol is a shortcut to keep the selection in the toolbar. This makes it possible to position multiple symbols of the same sort without going back and forth between the diagram and the toolbar.

To Customer	From Customer	To Hardware	From Hardware
CupOfCoffee	Coffee	FillWater	WaterOK
CupOfWater	Tea	FillCoffee	CoffeeOK
ReturnChange	Coin (Integer)	HeatWater	Warm

4. Signal **Coin** will carry one signal data of type integer. This signal data will hold the value of the inserted coin(s). Add the type **Integer** in the middle compartment of the symbol representing signal **Coin**, see Figure 7 on page 15.
5. Save your work.

Note

A coloring of an entity, for example on a signal name or a type, indicates that Tau has found the matching definition. This is called name resolution and appears on all levels when designing in Tau. Integer is one of the pre-defined types in UML. If no definition is present the name will be underlined with red.

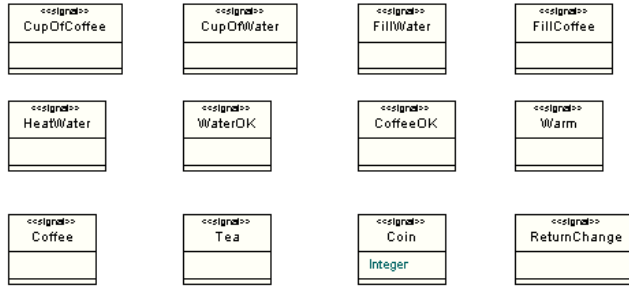

Figure 7: Signal definitions

Defining interfaces

You will now create interfaces containing the signals for the interaction with the environment. Do the following:

1. Open the class diagram **Signals**.
2. From the toolbar, use the **Interface symbol** and define the following two interfaces:
   - FromUser
   - ToUser
3. In the Model View drag the signals to the interfaces.
   - Signals **Coin**, **Coffee** and **Tea** should go to the interface **FromUser**.
   - Signals **ReturnChange**, **CupOfCoffee** and **CupOfWater** should go to the interface **ToUser**.
4. Save your work.

Note

Each signal will be a feature of the interface and the text line will be treated as a presentation element. In addition to being editable like text it will be possible to delete the entire line (presentation element) if you position the cursor first on the line and press backspace.

To view the signals in the interface symbols you can right-click and on the shortcut menu point to Show all operations. This will also let you modify signal parameters.

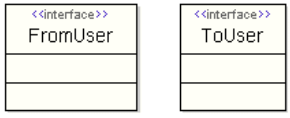

Figure 8: Interface definitions

Interface relations

The interfaces are now ready to be used. You will now create an **Association** between the two interfaces and a **Dependency** from ToUser to Customer.

1. Select **ToUser** interface symbol, drag the Association line handle from **ToUser** to **FromUser**. This will implicate that wherever one of the interfaces is used, the other will also be available.
2. Remove the navigability of the association. Right-click on the line close to its connection to the interface, select Target and in the sub-menu deselect **Navigable**. Right-click on the line close to the connection to the other interface, select Source and in the sub-menu deselect **Navigable**.
3. Go to the class diagram DomainModel. Use the **Previous** button on the Navigation toolbar, or double-click the diagram icon in the Model View.
4. Drag the interface **ToUser** from the Model View into the class diagram DomainModel. Select Customer class symbol, drag the Dependency line handle to ToUser interface symbol. This will indicate that Customer has a dependency ToUser. The class Customer is not to be implemented so this will not make any impact on the generated code.

Ports

All signals going to or from an instance of a class pass through a port. Separate ports can be used for each entity with which the part communicates. You will now add ports to classes in your model to enable communication, starting with ports for external communication. Do the following:

1. Go to the diagram DomainModel. Select class CoffeeMachine. Hold down SHIFT and in the toolbar select the **Port symbol**. A port appears on the border. This port represents the communication to and from the class. Name the port **P1**.

2. Make sure that the port P1 is selected. Add a realized interface to P1 by clicking the toolbar button. Name the interface **FromUser**.
3. With P1 selected press ALT + Enter to open the properties dialog. Make sure that Port is selected in Filter field. Note that the interface name is inserted in the **Realizes** field. Add the interface name **ToUser** in the **Requires** field.

4. Go to the diagram DomainModel again. Select the port P1 and click the tool bar button for the **Required** interface. Observe that the name is automatically inserted on the interface symbol.

Note

It is not necessary to add the required interface, because of the association between the interfaces. This is only done to improve readability.

5. Drag and drop the Controller class icon from the Model View into the open class diagram (DomainModel) on the Desktop.
6. Drag and drop the Hardware class icon from the Model View into the open class diagram (DomainModel) on the Desktop.
7. Select the class Controller, add a port “P2”. Add required and realized interfaces to port P2 in the same way as for P1. Observe that the signals must go in the same direction.
8. Save your work

Your diagrams should now contain ports and interfaces as shown in Figure 9 on page 18. Note that the diagrams are rearranged with respect to earlier pictures and that only a subset of the symbols are shown.

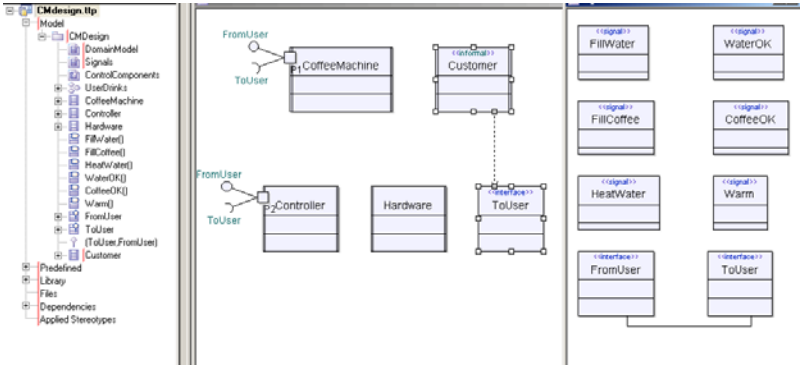

Figure 9: Interfaces and ports

Sequence Diagrams

General

A sequence diagram describes the behavior of use cases. A sequence diagram is commonly built up with instances of active classes in your system and messages (signal instances) between them. Sequence diagrams can be drawn manually or be generated during simulation from a Model Verifier trace.

Creating sequence diagrams

You will now create a sequence diagram for the use case MakeCoffee. Make sure that the Workspace window displays the Model View.

1. Locate the collaboration **UserDrinks** in the Model View. Right-click the use case **MakeCoffee**, from the shortcut menu point to **New** and click **Sequence diagram**.

Note

The actor type is now bound to class Customer. In the use case diagram the class name is no longer underlined with red, but is now colored green. This is an example of the work done in the background analyzing your model changes and updating your presentation elements to keep your model consistent at all times.

2. The diagram is now open and an icon for the sequence diagram appears inside the collaboration in the Model View. The sequence diagram is contained in an **Interaction** which can contain one or more diagrams describing the use case.
3. Locate the class **Customer** in the Model View and drag it to the sequence diagram. The class appears as a lifeline.
4. Drag the class **Controller** to the sequence diagram.
5. Drag the class **Hardware** to the sequence diagram.

In the following instructions you will use three different ways of creating messages in the sequence diagram. For each of these note especially the binding of signal names from your model. The **Message line** button in the element toolbar is the common starting point for all three ways.

6. Draw a message from Customer to Controller. Drag the signal **Coin** onto the message line. The signal name appears with parameter list containing the types of the parameters, in this case one integer parameter. Replace the parameter type information with the integer value 10.

7. From the element toolbar select the **Message line** button. Click on Customer and then right-click when placing the receive event on Controller. From the shortcut menu point to **Reference existing** and click **Coffee**.

Note

Reference existing can be used for classes with ports and interfaces that defines a set realized of signals.

8. Draw a message from Controller to Hardware. Type the signal name **FillWater** onto the message line.

Note

CTRL + Space is a shortcut command that performs name completion. If you start typing a name, this will complete the name or present a list with entities allowed in the scope.

9. Continue to draw the sequence depicted in Figure 10 on page 20
10. Save your work.

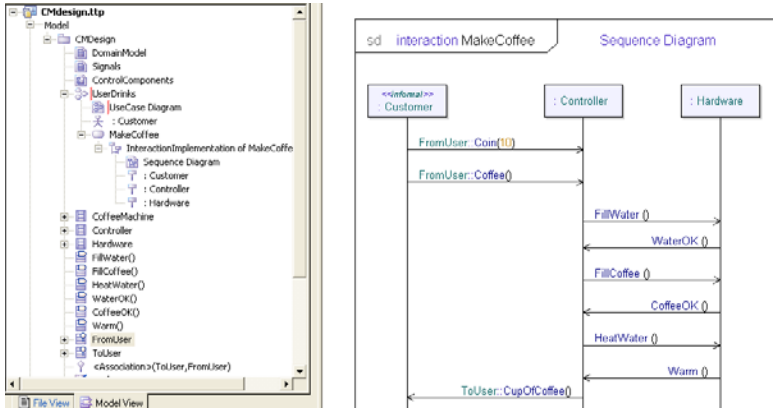

Figure 10: Sequence diagram for the use case MakeCoffee

Use cases and subject frame

1. Open the use case diagram in the collaboration **UserDrinks**. Create a new use case and name it **MakeTea**.
2. Create a new use case and name it **TeaMaking**.
3. Draw a dependency line from MakeTea to TeaMaking. Create an association line to the actor from MakeTea.

MakeTea should be one of the main use cases describing a complete scenario for the system to produce a cup of hot water. The use case TeaMaking is a sub-scenario describing filling of the cup and heating the water.

4. Create a **subject symbol** around the use cases. After selecting the symbol in the toolbar, drag to enclose the use case symbols with the subject symbol. Name the frame and identify the frame to belong to class CoffeeMachine. See Figure 11 on page 21.
5. Save your work.

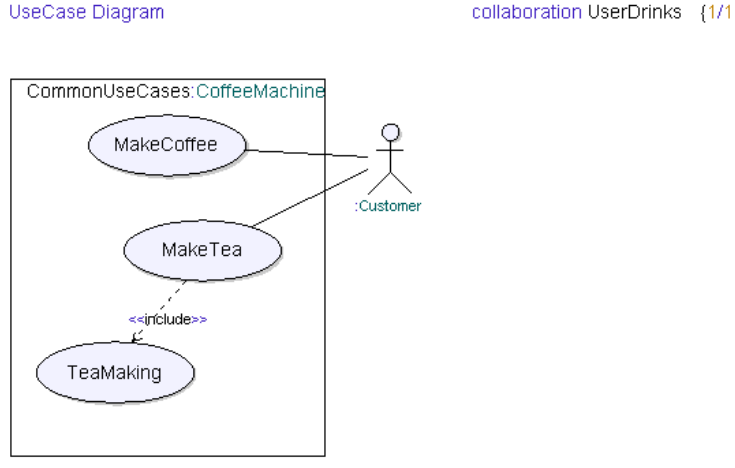

Figure 11: Use cases in a subject frame

The subject frame is optional but it encapsulates your use cases to belong to one particular class in your model allowing you to draw complex use case diagrams with common actors to one or more systems.

Sequence diagrams with references

1. Locate the collaboration **UserDrinks** in the Model View. Right-click the use case **MakeTea** and from the shortcut menu point to **New** and click **Sequence diagram**.

   The diagram is now open and an icon for the sequence diagram appears inside the collaboration in the Model View.

2. Locate the class **Customer** in the Model View and drag it to the sequence diagram. The class appears as a lifeline. Drag the class **CoffeeMachine** to the sequence diagram.

3. Draw a message from Customer to CoffeeMachine. Type the signal name **Coin** with parameter **5** onto the message line. Draw a new message from Customer to CoffeeMachine. Type the signal name **Tea** onto the message line.

4. Place a **reference** symbol below the signals. Name the reference symbol **TeaMaking**. See Figure 12 on page 22.

5. Save your work.

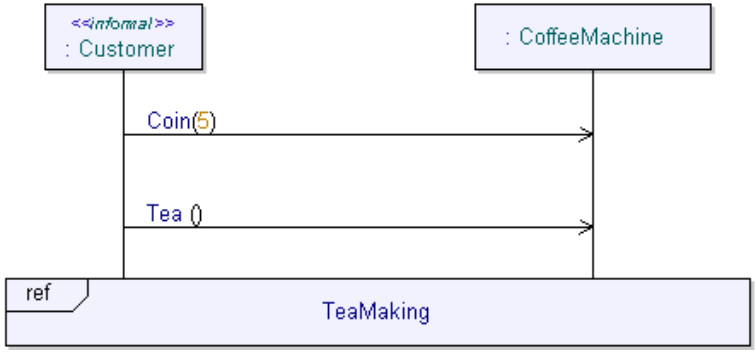

Figure 12: Sequence diagram with reference

You can use reference symbols as build blocks to create longer and more complex sequence diagrams from a common set of simple sequences.

Model Navigator

As a model grows to be larger and more complex, the work of navigating the model may take more and more time and can take an increasing amount of time to master. The Model Navigator is designed to alleviate many navigation tasks by dynamically determining a set of navigation modes applicable to some element. It may be trivial navigations, such as determining the scope or the different diagrammatic views of a model element, or it can be non-trivial navigations, such as finding all classes derived from a given ancestor class or all attributes of a given type.

You will now use the Model Navigator to browse through your model, using the signal FillWater as your starting point. You will navigate from the definition of the signal, up through its scope without losing contact with the signal, and try to find a diagram where it is used.

1. Select the root package CMdesign and then press CTRL + F to open the **Find** dialog. Type FillWater in the **Named** field. This will locate the signal FillWater in your model and present a listing in the Output window's Search result tab.

Note

The Find dialog only finds definitions, not the usage of them. To locate where an entity is used, right-click the entity in the Model View and from the shortcut menu click List References.

2. Look in the output window, tab **Search Result**. Right-click FillWater and then from the shortcut menu point to Locate (double-click will also do locate).

   The signal diagram is now open.

3. Now hold down CTRL and then point and click on the name FillWater inside the icon for signal FillWater in the diagram. The cursor changes to a hand and if the entity (signal FillWater) had not been selected in the Model View before it is now.
4. You can try this by holding down CTRL and click on some of the other signals in the diagram. Observe how the selection in the Model View also changes. If you hold down CTRL and point to something with multiple presentations (for example a signal that belongs to an interface) the Model Navigator will open.
5. Right-click the signal FillWater in the Model View, then point to **Model Navigator** from the shortcut menu.

The Navigate tab in the Output window becomes activated. Let us try to follow a path originating from the signal FillWater through the model.

6. Click the tab **References** and click the item with **Role**: InstanceOf (Location: MakeCoffee). This is the definition of the information found on a Message in a sequence diagram. The InstanceOf node shows where the signal is used.

Note

Each time you select a new definition in the Model Navigator your view is from the new definition. To go back you can use the Recent tab or right-click any tab.

7. Now go to the Shortcuts tab and select Scope, which is the Interaction node for the use case containing the sequence diagram.
8. The Interaction node has a Diagrams tab, allowing you to look at the diagram defined in this node. In this case there is only one diagram which becomes the current diagram on your desktop area when you click on the diagram item.

You have now taken a small tour in your system following the model bindings.

State Machine Diagrams

General

A state machine diagram describes the behavior of an active class. A state machine has one or more possible states and a change of state is triggered by a signal reception. In UML, the state machine concept has been extended with data handling, meaning that signal data and other variables can be declared and handled. Two different notations are supported: **transition-oriented syntax** and **state-oriented syntax**. Both syntaxes can be used for describing the behavior of a state machine, but the state-oriented syntax is more suitable for getting an overview of a large design. The transition-oriented syntax is suitable for detailed design.

State machine diagram for class Hardware

State-oriented syntax

When using state-oriented syntax, have in mind the following:

- When connecting states, select the first state. Two handles appear. Grab the handle represented by a filled circle. Drag the line to the second state and click the symbol.
- To change the shape of the connecting line, click it. Square-shaped handles appear. Drag the handles to form the requested shape.
- When drawing a line from a state back to the state itself, grab the handle represented by a filled circle and drag the line away from the state. Click to change directions, drag the line back to the state and click the symbol.
- The following notation is used on transition lines:
  <input>/ <transition statement>;
  where <transition statement> can be for example: ^ **<signal name>**. Signals without parameters should be followed by an empty set of parenthesis. There can be multiple statements in a transition, they are then separated by semicolon.

Composite state

It is possible to have composite states in a UML state machine. This is a way of defining a sub-state with a state machine of its own.

A composite state can be used for example when a set of actions tend to always lead back to one identifiable state. This can be identified in the state machine for **Hardware**. When producing tea or coffee the signal exchange between Controller and Hardware is very similar but there is one signal exchange more when the user requires a cup of coffee. This signal exchange can then be put in a composite state.

Starting with the state-oriented syntax, you will now describe the behavior of class Hardware in a state machine. Do the following:

1. Add a state machine to class Hardware. Right-click and from the shortcut menu point to **New** and then click **State machine diagram**. Name the state machine “HandlingWater”. The state machine is contained in a **state machine implementation** in turn contained in a state machine, by default named to **initialize**.
2. Implement the behavior as depicted in Figure 13 on page 26.
3. Save your work.

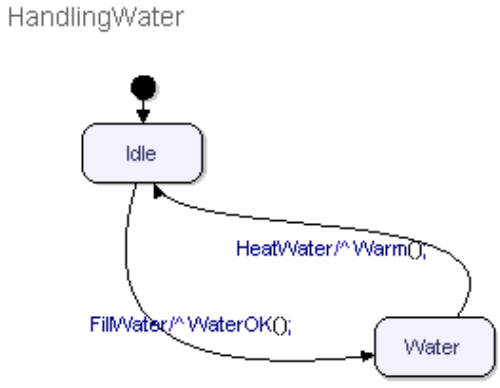

Figure 13: State machine diagram HandlingWater

Note

Name resolution, visualized as coloring of entities, appears whenever the matching definition of an entity is found. If the definition is included in the model, navigation is possible. To navigate to the definition, hold down CTRL and click the name.

UML is case-sensitive. In order for name resolution to be performed the case must match.

Transition-oriented syntax

When using transition-oriented syntax, have in mind the following:

- To add a symbol, click the corresponding quick-button in the toolbar, then click the desktop.
- To connect two symbols, select the first symbol. Two handles appear below the symbol. Grab the handle represented by a square and drag a line to the second symbol. Click the symbol.
- Autoflow: Select a symbol in the flow and hold down SHIFT, then click on one of the symbols in the toolbar. The new symbol is automatically connected to the selected symbol.

You will now use the transition-oriented syntax to describe the behavior in the composite state.

1. Open the state machine “HandlingWater” in class Hardware.
2. Double-click on state **Water**. The Model Navigator opens. Go to tab New diagram, point to **State machine diagram**. A new diagram opens in the desktop.
3. Draw the sub-state definition as depicted in Figure 14 on page 28. The history state implies a return to the state WaitFill.
4. Save your work.

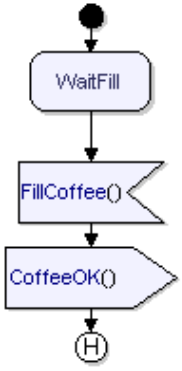

Figure 14: Composite state

State machine diagram for class controller

You will now describe the behavior of class Controller in a state machine, using the transition-oriented syntax. To add a state machine, do the following:

1. In the Model View of the Workspace window, select the icon representing class Controller and create a new state machine diagram for Controller.
2. Rename the state machine diagram to “NewOrder” to indicate that this diagram will describe the behavior when a new coffee or tea order is received.
3. Implement the behavior as depicted in Figure 15 on page 29.

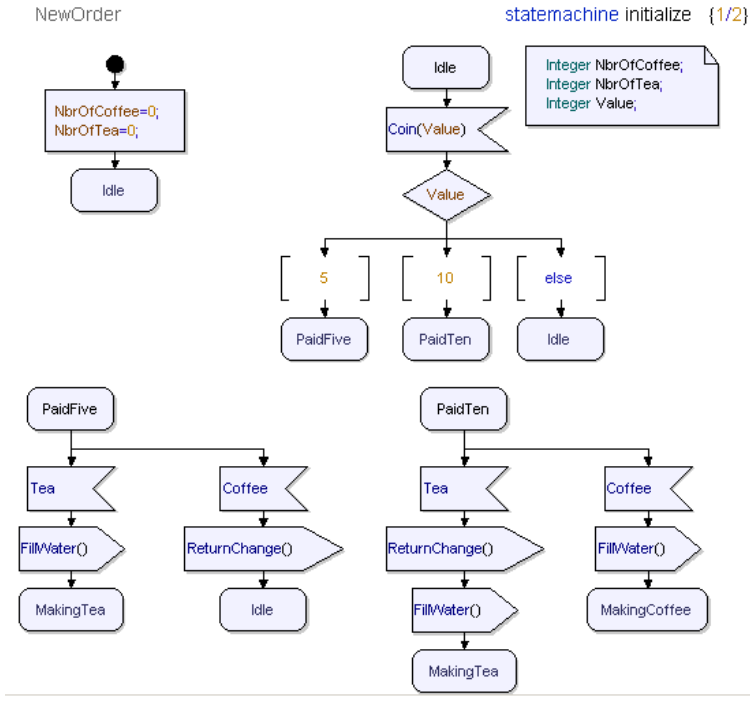

Figure 15: State machine diagram NewOrder

The state machine describes the reception of a coin, the reception of a coffee or tea order and the start of the hardware communication. Two variables are used for counting the number of cups of coffee or tea that have been served. Another variable is needed for holding the value of the coin.

4. To get more drawing space, you can now add a new state machine for the rest of the behavior. In the Model View, select the icon representing the **state machine implementation** and create another state machine for Controller.
5. Name the new state machine “MakingBeverage“.
6. Implement the rest of the behavior according to Figure 16 on page 30.
7. Save your work.

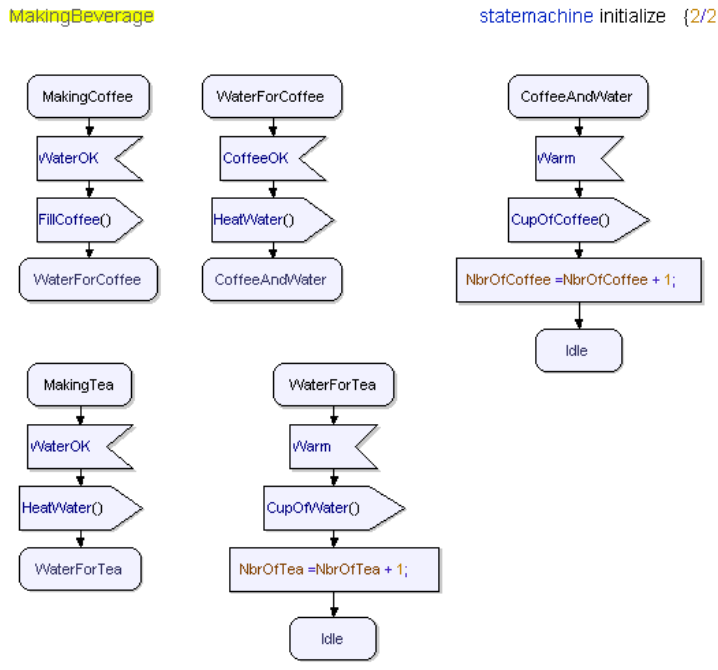

Figure 16: State machine diagram MakingBeverage

The state machine diagrams are now complete.

Composite Structure Diagrams

General

The next step is to add a composite structure diagram. A composite structure diagram displays instances of active classes, and the communication between them. This is the diagram for showing how your objects from your model should be instantiated and built together to form a system. The instances are called **parts**. A part communicates with other parts or with the environment through **ports**. Ports are connected through **interfaces** or **connectors**.

Creating a composite structure diagram

You will now instantiate your classes and describe the communication between the customer, the controller and the hardware in a composite structure diagram. To add a composite structure diagram, do the following:

1. In the Model View, select the icon representing class CoffeeMachine.
2. Right-click on class CoffeeMachine and select **New** and then **Composite structure diagram**.
3. The composite structure diagram appears in the Model View. Name the diagram “Communication“.

Parts

A part is an instantiation of an active class. Add the parts by doing the following:

1. Make sure that the diagram Communication is open on your desktop. From the toolbar, select the symbol representing a **part** and place it in the diagram.
2. Click inside the symbol to activate the text area. Name the instance **Ctrl:Controller**, where Ctrl is the name of the part and Controller is the name of the class it instantiates.
3. Add a part representing the instantiation of class Hardware. Name the part **Hw:Hardware**.

Note

It is also possible to drag-and-drop the active classes to the composite structure diagram.

Ports

You will now add some more ports to your model to enable communication, starting with the port representing the customer. Do the following:

1. Select the part Ctrl. Hold down SHIFT and in the toolbar, select the **Port symbol**. A port appears on the border. Name the port “P3”. This port represent the communication with the Hw part.
2. Select the Hw part. Add one port and name it “P4”. This port represents the communication with the Ctrl part.
3. Add required and realized signals for the ports P3 and P4. Observe that the signals in the ports must correspond to the direction of the connection line.

   The following signals can be received by Ctrl:

   – CoffeeOK, WaterOK, Warm

   The following signals can be received by Hw:

   – HeatWater, FillWater, FillCoffee

Connectors

A connector is a signal path which can be bidirectional or unidirectional. Connectors connect the ports in the composite structure diagram. You will now add a connector to your model. Do the following:

1. Add a connector between ports P3 and P4. This is done by selecting one of the ports, then dragging the handle to the other. Three text fields appear. Name the connector “CtrlbiHw“.
2. Right-click on the connector line and select **Show all signals** from the shortcut menu. For each of the connectors the signals corresponding to the Realizes and Requires properties will be filled in.

Your composite structure diagram should now look as depicted in Figure 17 on page 33.

3. Save your work.

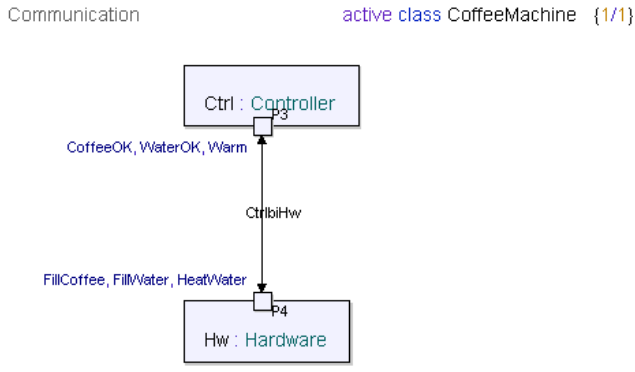

Figure 17: The composite structure diagram

Your composite structure diagram is now complete. Ports and parts have been added to the model as you have edited the diagram.

Ports and interfaces

You will now add your ports to the classes in the component diagram.

1. Open your component diagram (ControlComponents).

Figure 18: The component diagram with ports and interfaces

2. Drag and drop port P3 to Controller.
3. Drag and drop port P4 to Hardware.

The interfaces and signal lists will be shown as in Figure 18 on page 33.

Relations

General

The next step is to add relations between your classes. Relations between classes in a model are best illustrated in the class diagram.

Associations

An **association** represents a relationship between objects. Each association has two roles, represented by the directions of the associations. You have earlier created an association between the interfaces ToUser and FromUser in the model.

Compositions

Controller and Hardware “live and die” with class CoffeeMachine, which means that their relation to class CoffeeMachine is a **composition**.

1. In the diagram DomainModel, select the class CoffeeMachine. Three handles appear below the symbol.
2. Grab the handle represented by a square (Association/Aggregation/Composition line), drag the line to class Controller and right-click to the class symbol. Select **Reference Existing**. The drop-down box should contain the Ctrl part.

Note

It is possible to change a composition into an association or an aggregation from the shortcut menu, by right-clicking on the line close to the connection to CoffeeMachine.

3. Add a **composition** in the same way between class CoffeeMachine and class Hardware.
4. Save your work.

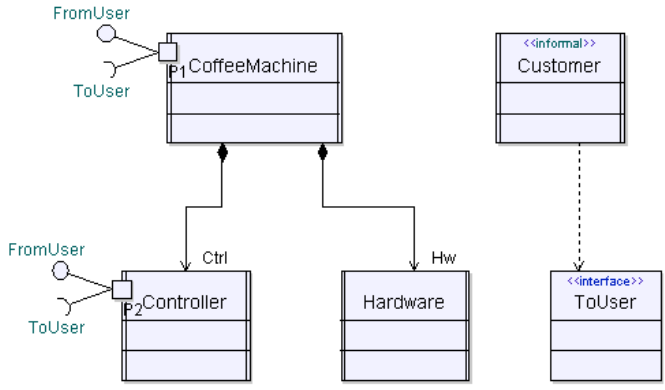

Figure 19: Compositions in the domain model

Your class diagram should now look as depicted in Figure 19 on page 35.

Check

General

Tau has built-in check functionality, with the purpose of finding errors as soon as possible. Two types of checks are supported, **Autocheck** and **Check**.

Autocheck

The Autocheck is constantly running, checking the scope which you are currently working in. The scope is determined from the diagram that is open on the Desktop. All errors found are displayed in the Output window. Since the check is constantly running, an error message disappears as soon as the error is corrected.

Check

A check is initiated by you by clicking one of the check quick-buttons, **Check All** or **Check selection**. **Check All** checks the entire model, while **Check selection** only checks the entity selected in the Model View, and its subtree. All errors found are displayed in the Output window. The errors remain in the list until a new check is initiated.

To check your model, do the following:

1. Click the **Check All** quick-button.
2. If error messages appear in the Output window, try to understand and correct them.
   - Some types of error messages can be double-clicked, causing the problem area to appear on the Desktop.
   - Another way to do this is to right-click the error message and in the shortcut menu that appears, point to **Locate**.

   To verify a correction, click the **Check All** quick-button again.

Build Artifact

General

A build artifact is representing your model or a subset of it. A build artifact is an element in the UML model, with properties dedicated to the build process. When working with a build artifact a number of files, representing a source code kernel, is used to translate a UML model into an executable program. Different kernels exist for applications and debugging.

Model Verifier set-up

The build artifact requires a proper set-up of code generator add-ins for the Model Verifier. This is primarily controlled when you created your project file, but can be changed from the Tools menu, selecting Customize. Open this dialog and in the **Add-ins** tab check that the module **Model Verifier** is active.

Creating a build artifact

You will now create a build artifact representing your model. The build artifact will be executed later.

1. Right-click **CMdesign** package and from the shortcut menu point to **Model Verifier** and click **New Artifact**. An icon for the build artifact appears in the Model View. Name the build artifact “artifact_cm“.
2. Set a root for the Build Artifact. This decides which part of the model that you want the Build Artifact to represent. The Build Artifact will represent the entity you set as root and its sub-tree. Right-click on artifact_cm in the Model view. From the shortcut menu choose **“Select Build Root...**”. Set the root to class CoffeeMachine. Click **OK**.
3. Right-click the build artifact icon in the Model View and select **Build (Model Verifier)** then point to **Launch**.

Note

*In order to build with the Model Verifier, you must have a supported C compiler installed. If you are using a GNU C Compiler (GCC) you must adjust your artifact for this. Right-click **artifact_cm**, point to **Properties**. In the **Filter** menu choose the stereotype 'ModelVerifier'. Set **Target Kind** to* `Win32 - gcc`.

4. The result of the build is displayed in the Output window. When the build is completed a set of status messages is displayed, ending with a count of error and warning messages.

Model Verifier

General

Verifying your model gives you the opportunity to test the behavior of your design. The responses to received signals can be evaluated. Three different types of tracing are available: textual trace, graphical UML trace and sequence diagram trace.

Coffee machine verifying

You will now test the behavior of the coffee machine. To start a Model Verifier session, do the following:

1. Right-click your build artifact, artifact_cm, in the Model View and select **Build (Model Verifier)** then point to **Launch**. A new tab, **Instances**, opens in the Workspace window.
2. Expand the information in the Instances tab by selecting **CMdesign.ttp** and pressing the multiplication key (*) on your numeric key pad. A number of different parameters belonging to Ctrl, Hw and the environment (env) are displayed.

Creating a message matrix

A list of possible signals, a message matrix, can be created to simplify the debugging. This is opened from the **View** menu, **Model Verifier Windows**, then select **Messages**.

To test all use cases, four different signals from the environment are needed:

- Signal Coin, carrying value 10
  This signal represents inserting a coin with the value of 10.
- Signal Coin, carrying value 5
  This signal represents inserting a coin with the value of 5.
- Signal Coffee
  This signal represents pressing the Coffee button.
- Signal Tea
  This signal represents pressing the Tea button.

To add signal Coin carrying value 10 to the message matrix, do the following:

1. Double-click the upper left field of the message matrix, containing '...'. The first row is activated.
2. In column **Sender**, click **unspecified**. A drop-down box appears. Click **env[1].**
3. In column **Signal**, click **unspecified**. A drop-down box appears. Click "**::CMdesign::Coin**".
4. In column **Channel**, keep the current value, **unspecified**.
5. In column **Receiver**, click **unspecified**. A drop-down box appears. Click **CoffeeMachine.Ctrl[1].**
6. In column **Parameter**, click **unspecified**. In the field that appears, type the value **10**. Press **Enter** on your keyboard. The signal is included in the message matrix.
7. Add the other possible signals according to Figure 20 on page 40.

	Sender	Signal	Channel	Receiver	Parameters
1	env[1]	::CMdesign::Coin	unspecified	CoffeeMachine.Ctrl[1]	10
2	env[1]	::CMdesign::Coin	unspecified	CoffeeMachine.Ctrl[1]	5
3	env[1]	::CMdesign::Coffee	unspecified	CoffeeMachine.Ctrl[1]	unspecified
4	env[1]	::CMdesign::Tea	unspecified	CoffeeMachine.Ctrl[1]	unspecified
...					

Figure 20: Message matrix

Note

To send a signal in the message matrix, double-click the first field.

Watch parameters

The current states for env, Ctrl and Hw are displayed. At start up, all states are set to start, referring to the Start symbol. This means that the state machines have not yet reached their first states. It is possible to select which of the other parameters should be watched during the debugging.

1. In the instances view, right-click on the parameters below. In the shortcut menu that appears, point to **Watch**.

   Ctrl:

   – NbrOfCoffee

   – NbrOfTea

   – Queue

   Hw:

   – Queue

   env:

   – Queue

Tracing

Different methods can be used for tracing the debugging. Per default, a textual trace is started. You can see the textual trace in the lower-right corner of the user interface. Per default, execution tracking is also executed. This means that the UML symbol currently executed is highlighted in the state machine on the Desktop. As an option, a sequence diagram trace can be used. To start the debugging, do the following:

1. During the first use case you will only use a sequence diagram trace. Turn off the execution tracking by clicking the **Show Next Statement** quick-button.

2. Start a trace by clicking the **Tracing in sequence diagram** quick-button. A sequence diagram appears on the Desktop. The sequence diagram contains instances representing the environment, Ctrl and Hw.

3. Click the **Next Transition** quick-button twice. Ctrl and Hw both reach their Idle states.

Verifying use cases

It is now time to insert a coin in the coffee machine. The first use case you will debug is the case where coins to the value of 10 are inserted, and the user selects Coffee. A cup of coffee should be returned.

1. In the message matrix, double-click the left most field of the first line, representing signal Coin, value 10 (or right-click and point to **Send**). In the sequence diagram trace, the sending of signal Coin is displayed. The signal is placed in the input queue of Ctrl. This can be seen in the **Queue** parameter for Ctrl.
2. Click the **Next Transition** quick-button. Signal Coin is now consumed by Ctrl. The queue for Ctrl should now be empty. Note that the state for Ctrl is changed to **PaidTen**.
3. Select coffee by sending in the third signal in the message matrix. The sequence diagram displays the signal sending.
4. Click the **Next Transition** quick-button. Signal Coffee is consumed and signal **FillWater** is sent from Ctrl to Hw.
5. Click the **Next Transition** quick-button the number of times required for signal **CupOfCoffee** to appear, indicating that the use case is completed. Note that the Ctrl parameter **NbrOfCoffee** is increased to 1.

Both Ctrl and Hw have now completed their tasks and are back in state idle, ready to handle a new order.

6. Now that you are familiar with the sequence diagram trace, turn the execution tracking back on by clicking the **Show Next Statement** quick-button. The executed UML symbol will now be high-lighted.

7. You will now save this test to possibly replay it later. Press the **Replaying mode** button. This will put the Model Verifier in a Replay mode letting you step through your test from the start of the session.

8. Press the **Restart** button. Then press Next-transition repeatedly and observe the same test be performed once again.

When the test is complete most buttons in the Model verifier tool bar are dimmed. Observe that the test is done with the change in trace that you have selected. You may interrupt a replay test scenario between transitions for example by sending other signals into the system. Until you deactivate the replay mode it will try to recover execution from where it was interrupted.

9. You will now save this test scenario. This is done from the File menu, select Save Scenario. Save this test as **Coffee10.ttdscn**. If you make changes to your system you can replay this scenario the next time you execute a Model Verifier.

Note

You can switch between diagrams on the Desktop with Ctrl + Tab.

10. In the same way as above, execute the rest of the use cases described in "Behavior of the coffee machine" on page 2. Before each scenario you should start the sequence diagram trace, and at the end of each scenario you should stop the trace. This will produce a new use case for each scenario. The use cases generated from a trace will be visible in the Model View in a separate package called **DebugTrace**.

Note

The starting states for the state machines will only be shown in the traces when you have performed a complete Restart of the Model Verifier session.

11. When all scenarios are completed you can stop the Model verifier session, press the **Stop Model Verifier** quick-button.

Hint

The package DebugTrace will by default not be saved in your current model file. If you want to keep these diagrams you should save them into a new file or move them into your current .u2 file. Use the shortcut sub-menu Model View Filters and select Show Files, then drag-and-drop the package (or a selection of your choice) into the .u2 file found in the Files section.

Referenced sequence diagram

You will now use the sequence diagram traces from the Model Verifier session to create the referenced sequence diagram TeaMaking.

1. Locate a sequence diagram from the traced sequences where the drink selection is Tea. Double-click on "interaction" (or anywhere in the diagram heading). This will locate the use case in package DebugTrace in the Model View.
2. Drag the Interaction node of this use case to the use case for TeaMaking.
3. Edit the sequence diagram to look like Figure 21 on page 44. Drag the appropriate classes from the Model View of the CMdesign package to the lifeline headings. Remove the signals Coin and Tea. (These are specified in the scenario MakeTea, see Figure 12 on page 22.) Check that all states and signals have bound to the model.

4. Save your work

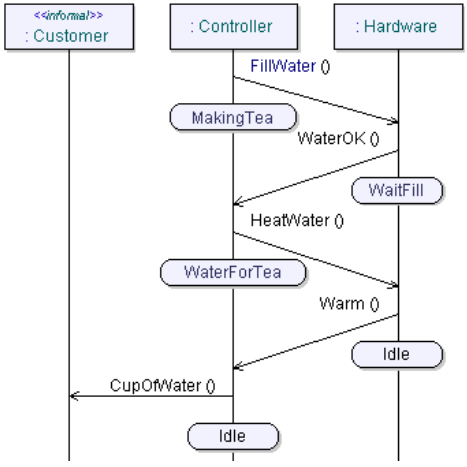

Figure 21: Sequence diagram TeaMaking

Iterations and additions

Purpose

This next section cover some additions to the model, showing some more features of UML. You will also get a repetition of some of the editors that you have worked with along the tutorial. This will also give a brief insight in how Tau supports an iterative work process.

Some of the descriptions of what to do will in this section be shorter and not as detailed as before. This will especially be so when it concerns activities similar to those earlier described.

Timer

Introduce a timer into your coffee machine. The timer should expire after 10 time units and it is to be set in conjunction with the activity of heating the water.

1. Go to class Hardware in the Model View. Add a timer called **Heater** to the class. Right-click class Hardware and from the shortcut menu point to **New** and click **Timer**.
2. Go to the state machine for Hardware. The timer signal should be received in such a way that it triggers the sending of signal Warm. Replace the signal **HeatWater** with **Heater** in the transition from state Water to state Idle.

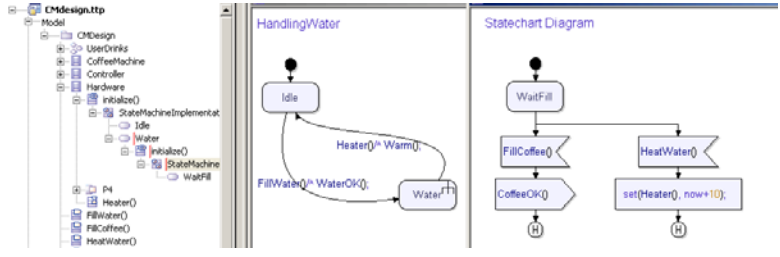

Figure 22: Timer declaration and use

3. Go to the state diagram for composite state Water. Add the signal HeatWater in parallel to the signal receipt of FillCoffee. This transition should contain the **set** statement of timer Heater, "`set(Heater(),now+10);`". See Figure 22 on page 45.
4. Save your work.
5. You should now build and execute with the Model Verifier to test that the timer works as expected. Trace the scenario in a sequence diagram. This will show timer representation in sequence diagrams. Use your saved scenario **Coffee10.ttdscn** to run this test. When the test reaches the timer event it can no longer proceed as the system has changed. Stop the replay of the test and run the rest of the scenario manually.

6. When all scenarios are completed you can stop the Model verifier session, press the **Stop Model Verifier** quick-button.

Structured data

Introduce a class representing a set of structured data into your coffee machine. The contents of this data model will be milk and sugar to be put into the coffee. This will require several steps to be performed as it will affect receiving and sending of signals as well as some internal behavior. It will also result in a more complex scenario when executing with the Model Verifier.

1. Open the class diagram **Signals**. Add a class in the diagram and name the class **Additives**.
2. Add two attributes to the class, **Milk** of type **Boolean** and **Sugar** of type **Integer**. Declare the attributes to be public.

Hint

To set the visibility to public type a '+' sign in front of the attribute name in the interface (class) symbol.

Note

*Next you will add a parameter of type Additives to the signal **Coffee** and to the signal **CupOfCoffee**. To be able to edit parameters for signals in interfaces you must first make the signals visible in the interface symbol.*

3. Select the signal **Coffee** in the Model View, drag it to **FromUser** interface symbol in the class diagram. Type in the parameter and declare it as a part.
4. Select the signal **CupOfCoffee** in the Model View, drag it to **ToUser** interface symbol in the class diagram. Type in the parameter and declare it as a part. See Figure 23 on page 47.

5. Open the state diagram **NewOrder** for class Controller. Declare a variable called **Add** of type Additives. Declare the variable as a part. See Figure 24 on page 48.

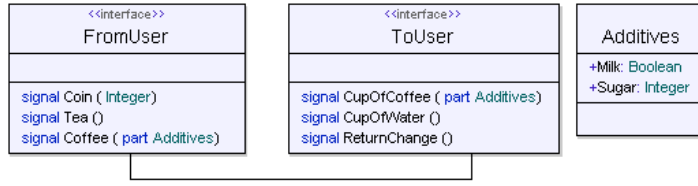

Figure 23: Class Additives

Note

Declaring a parameter or variable as part affects the way it will be handled during code generation and thus it will affect how to send in values through the Model verifier interface. By default a class will be represented by a pointer to a structured data model, part variables will be possible to reference by value.

6. Declare a variable called **NoAdd** of type Additives. Do **not** declare the variable as a part.
7. Use the variable Add to receive request for milk and sugar whenever the signal coffee is received, a possible solution is shown in Figure 24 on page 48.
8. Use the value of the variable Add as parameter when sending CupOfCoffee to the environment.
9. Save your work.

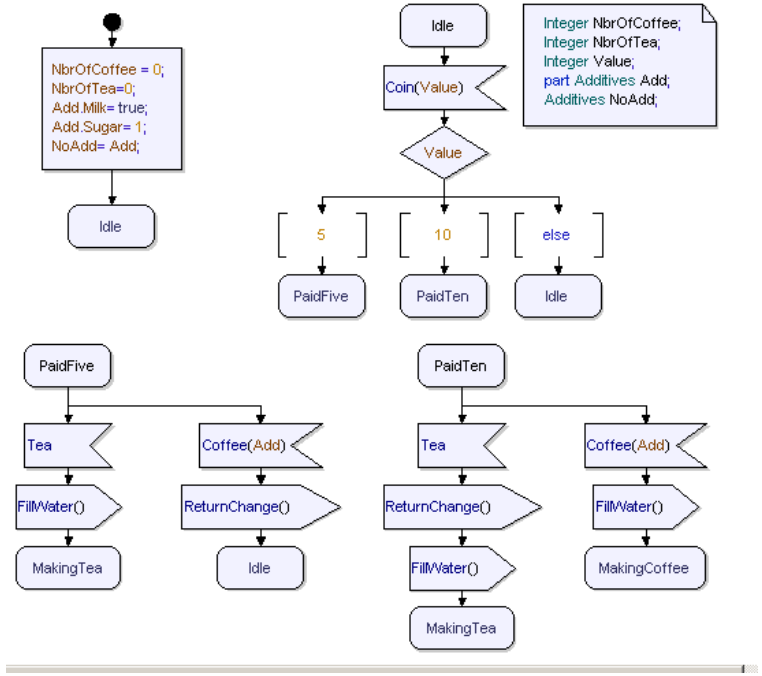

Figure 24: State machine for Controller with Additives

10. You should now build and execute with the Model Verifier to test that the new signal parameters work as expected. Use the possibilities to **Watch** attributes in the Model Verifier, note for example how Add and NoAdd are presented.
11. You will also have to update the sequence diagrams to allow for the new parameter. See Figure 25 on page 49. Observe the type qualifier before the parameter. You will have to supply the complete class with the type qualifier in the parameter field of the Messages window (`Additives (.Milk=true, Sugar=3.)`).
    It is possible to copy and paste the class with parameters from a **Watch** window by using **Deep Copy** on the shortcut menu and then paste in the **Parameters** field with CTRL + V.

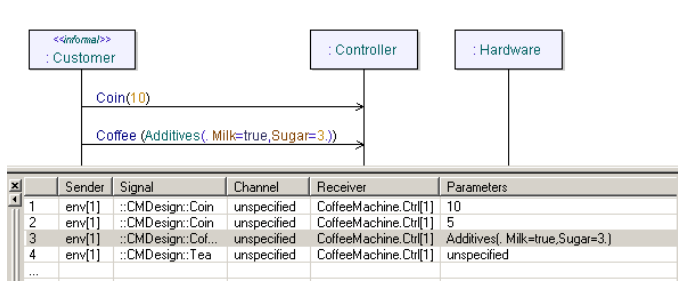

Figure 25: Sequence diagram and Messages window with parameter of type Additives

Conclusions

Model

The diagrams in UML are views of your model. Whenever possible the model will work for you, creating definitions and instantly binding these to elements as they are created. To your assistance you have the Model Navigator which allows you to browse the model bindings of your choice.

Editors

You should now feel familiar with working with the editors in Tau. The example you have built is simple to its functionality but contains examples of a large part of the possible symbols in UML. The tutorial has in some situations shown different ways of drawing similar constructions. Which to use in a given situation depends on the characteristics of the problem but is also many times a personal preference. Important to understand is the underlying model support to speed up your work and make it more precise.

Test

The test activities described within this tutorial model are limited to a set of validations that normal scenarios execute properly. In short that the system does what was intended.

Workflow

The workflow in this tutorial is intended to be aligned with a possible workflow in a larger software project. The scope of the tutorial is focused on demonstrating the tool rather than imposing a methodology. The order of the activities described should however fit well with the design phase of a larger software project. It is in this case important to remember that the analysis is already done and even if you have edited the complete model from scratch all the design decisions was made in advance. You can find some more information on workflow methodology in the chapter Description of Workflow in the online help of Tau.

What's next?

You have now completed the tutorial and are ready to start working on your own with UML and Tau. If you would like to have more information on model driven work it is recommended that you study the chapter "Working with Models" in the online help.