Cadena Tutorial

Adam Childs

Jesse Greenwald

Jason Weber

Cadena Tutorial

Adam Childs
Jesse Greenwald
Jason Weber
David Haverkamp
Gary Daugherty

Table of Contents

List of Figures

List of Examples

Foreword

This tutorial offers an example-based introduction to the Cadena Development Environment. Before you walk through this tutorial, be sure to read the entire Cadena README file and follow all of the installation instructions.

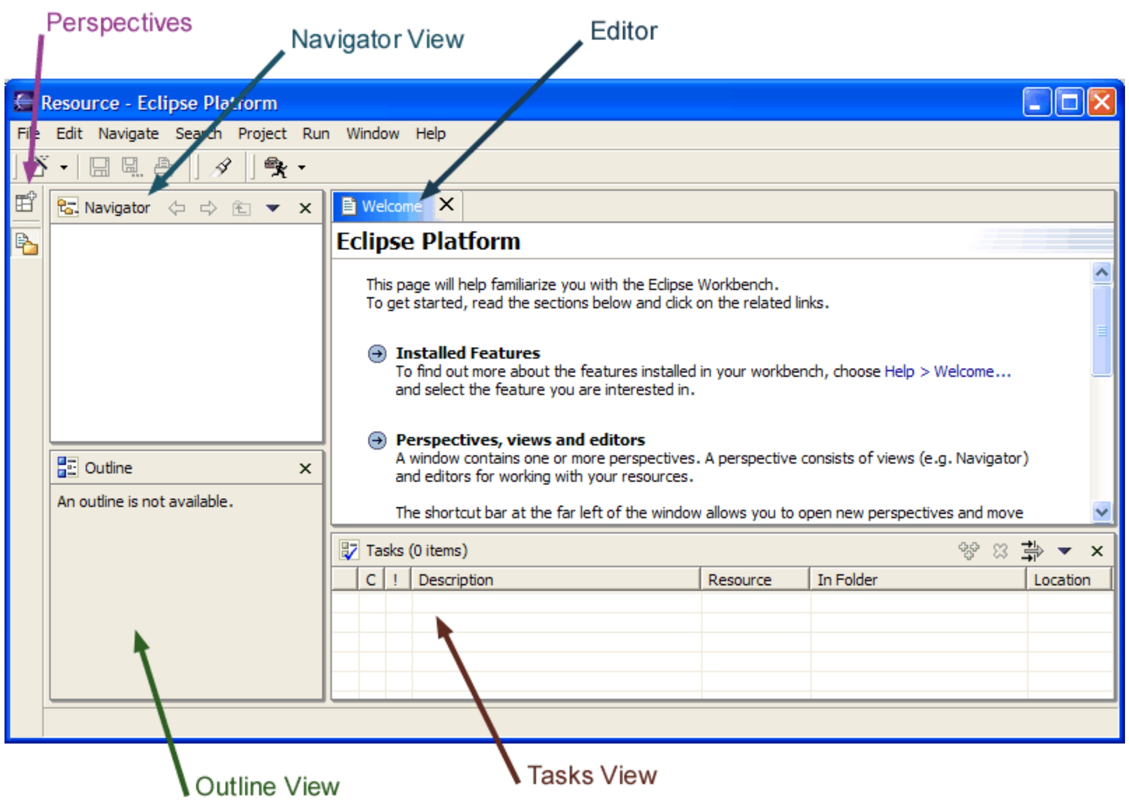
If you have any questions regarding Cadena or find any errors in this document, please contact `<cadena@cis.ksu.edu>`.

Chapter 1. Getting Started

Introduction to Eclipse

Eclipse is a Java-based open-source IDE. The Eclipse user interface consists of views and editors grouped together as perspectives. In the figure below, icons representing perspectives are displayed on the far left. The Resource perspective (which will be used throughout this tutorial) is currently selected.

Figure 1.1. Eclipse main window


Perspectives
Navigator View
Editor
Resource - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Navigator
Welcome
Outline
Tasks (0 items)
C ! Description Resource In Folder Location
Outline View
Tasks View

Eclipse Platform

This page will help familiarize you with the Eclipse Workbench.
To get started, read the sections below and click on the related links.

Installed Features
To find out more about the features installed in your workbench, choose Help > Welcome...
and select the feature you are interested in.

Perspectives, views and editors
A window contains one or more perspectives. A perspective consists of views (e.g. Navigator)
and editors for working with your resources.

The shortcut bar at the far left of the window allows you to open new perspectives and move

An outline is not available.

The figure above also shows several views and an editor that will be used in this tutorial. The Navigator view is used for convenient navigation of project files and folders. The Outline view is linked with the editor and displays the hierarchal structure of the edited file. In this example, the editor is also supported by the Tasks view. The Tasks view displays errors and warnings during development.

Perspectives can be switched by selecting a perspective icon from the bar on the left, or from the menu bar by selecting: Window->Open Perspective...

Views can be switched from the menu bar by selecting: Window->Show View...

Note

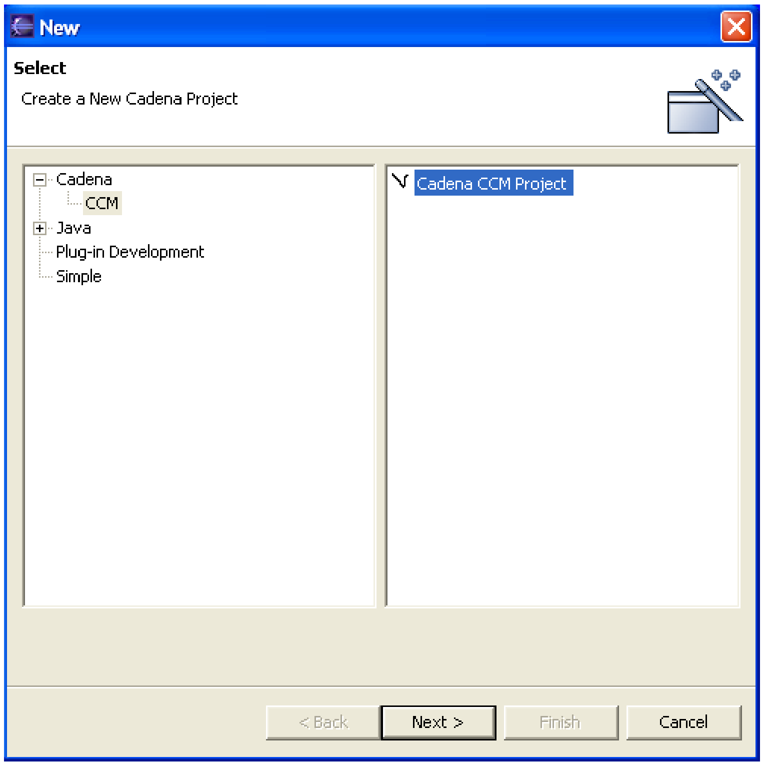
For a complete tutorial on Eclipse only (no reference to the Cadena plugin), take a look at their online documentation [http://www.eclipse.org/documentation/main.html].

Creating a new Cadena project

To create a new Cadena project:

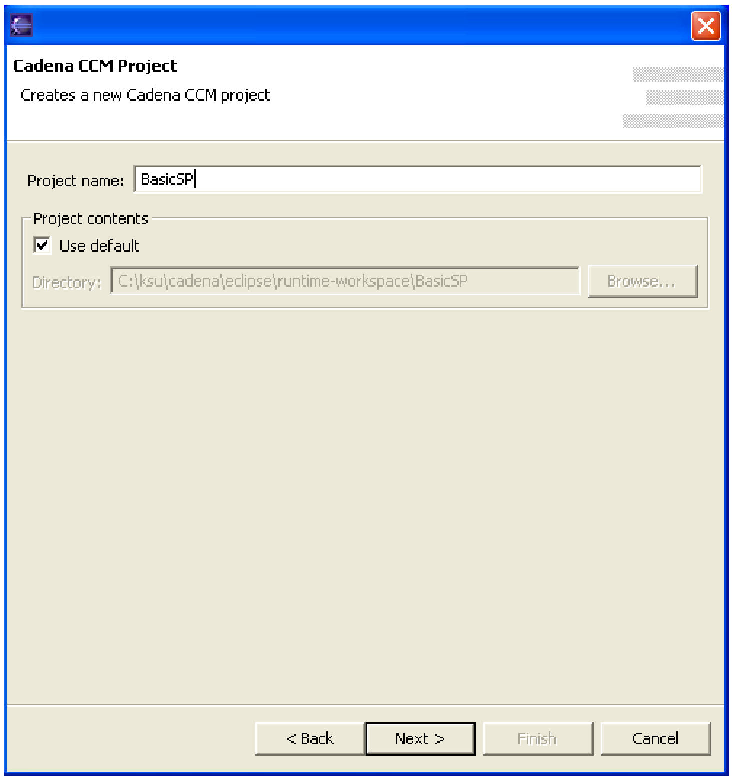
- Right click within the Navigator view or click on the File menu.
- Select New->Other... from the popup menu.

Figure 1.2. Creating a new project



- Expand Cadena, click on CCM, and select Cadena CCM Project (as seen above).
- Select "Next".

Figure 1.3. Naming a new project



Cadena CCM Project

Creates a new Cadena CCM project

Project name: BasicSP

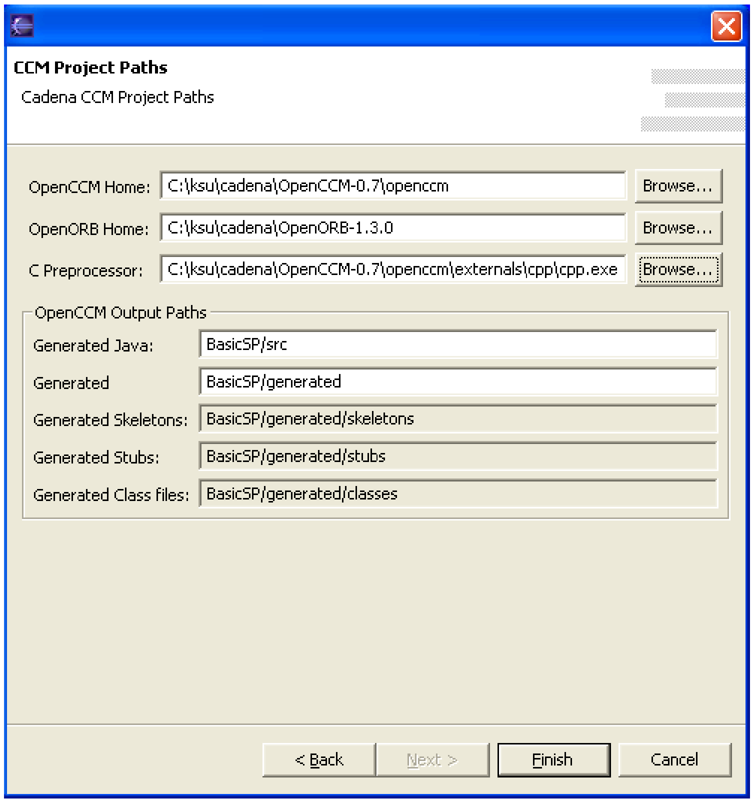
Project contents

☑ Use default

Directory: C:\ksu\cadena\eclipse\runtime-workspace\BasicSP

- Name the new project "BasicSP".
- Select "Next".

Figure 1.4. Configuring a new project


CCM Project Paths

Cadena CCM Project Paths

OpenCCM Home: C:\ksu\cadena\OpenCCM-0.7\openccm

OpenORB Home: C:\ksu\cadena\OpenORB-1.3.0

C Preprocessor: C:\ksu\cadena\OpenCCM-0.7\openccm\externals\cpp\cpp.exe

OpenCCM Output Paths

Generated Java: BasicSP/src

Generated: BasicSP/generated

Generated Skeletons: BasicSP/generated/skeletons

Generated Stubs: BasicSP/generated/stubs

Generated Class files: BasicSP/generated/classes

- For the CCM Project paths, select "Browse..." and select the directory corresponding to the above path structure. Use the path of the OpenCCM installed during the Cadena installation.

  The OpenCCM path should be:

  ```
  [Cadena install]\OpenCCM-0.7\openccm
  ```

  The OpenORB Home path should be:

  ```
  [Cadena install]\OpenORB-1.3.0
  ```

  The C++ Compiler path should be:

  ```
  [Cadena install]\cadena\OpenCCM-0.7\openccm\externals\cpp\cpp.exe
  ```

  Note

  The OpenCCM output paths are initialized according to the project name. These paths are passed to OpenCCM during code generation. Leave these paths as they are for now.

- Select Finish to create the new project.

If the BasicSP project has been created, it should appear in the Navigator view.

Importing Files

Running the BasicSP example requires the following files to be added to the project:

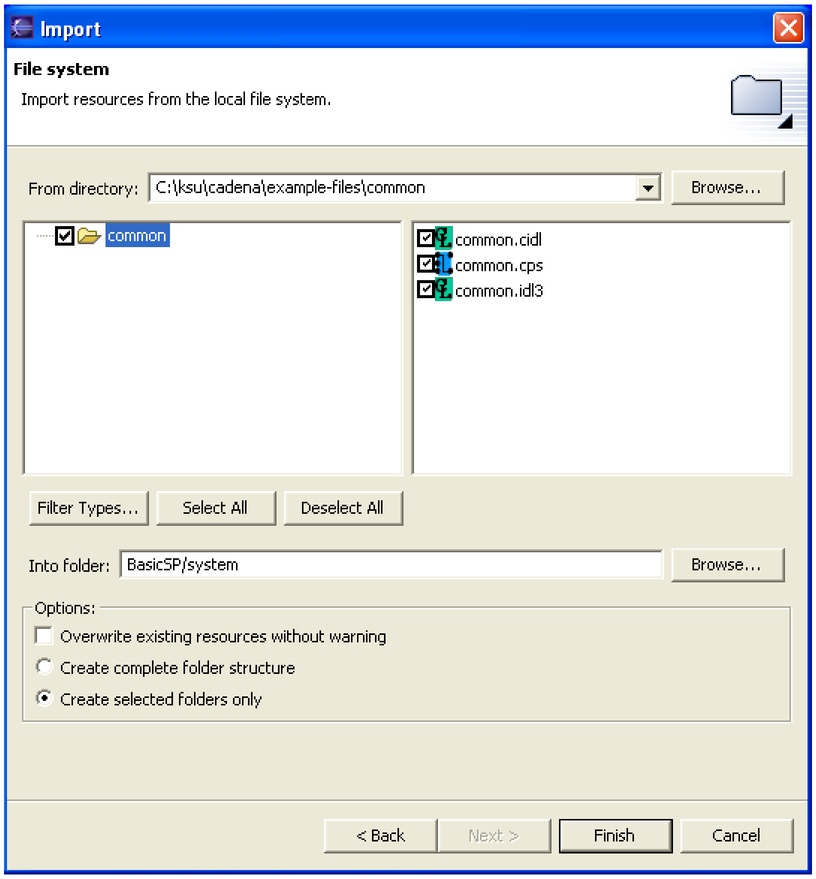
- `basicsp-cadl.scenario`
- `Boeing_OEP_CCM.profile`
- `common.cps`
- `common.idl3`

Files of type ".scenario" and ".profile" need to be imported into `system\cad\` directory, files of type ".cps" need to be imported into `system\cps\`, and files of type ".idl3" need to be imported into `system\idl\`.

To import `common.idl3` and `common.cps:`

- In the Navigator View, expand all folders.
- Select the `system` folder.
- Click the right mouse button, Select "Import..."
- Select "File system" as the import source.
- Select "Next".
- For the directory, select `[Cadena installation]\example-files\common`.

Figure 1.5. Importing the common files


Import

File system

Import resources from the local file system.

From directory: C:\ksu\cadena\example-files\common

[✓] common

[✓] common.cidl
[✓] common.cps
[✓] common.idl3

Into folder: BasicSP/system

Options:
[ ] Overwrite existing resources without warning
( ) Create complete folder structure
(●) Create selected folders only

- Select `common.idl3` and `common.cps` as seen in the figure above.
- Select "Finish" to import the selected files.

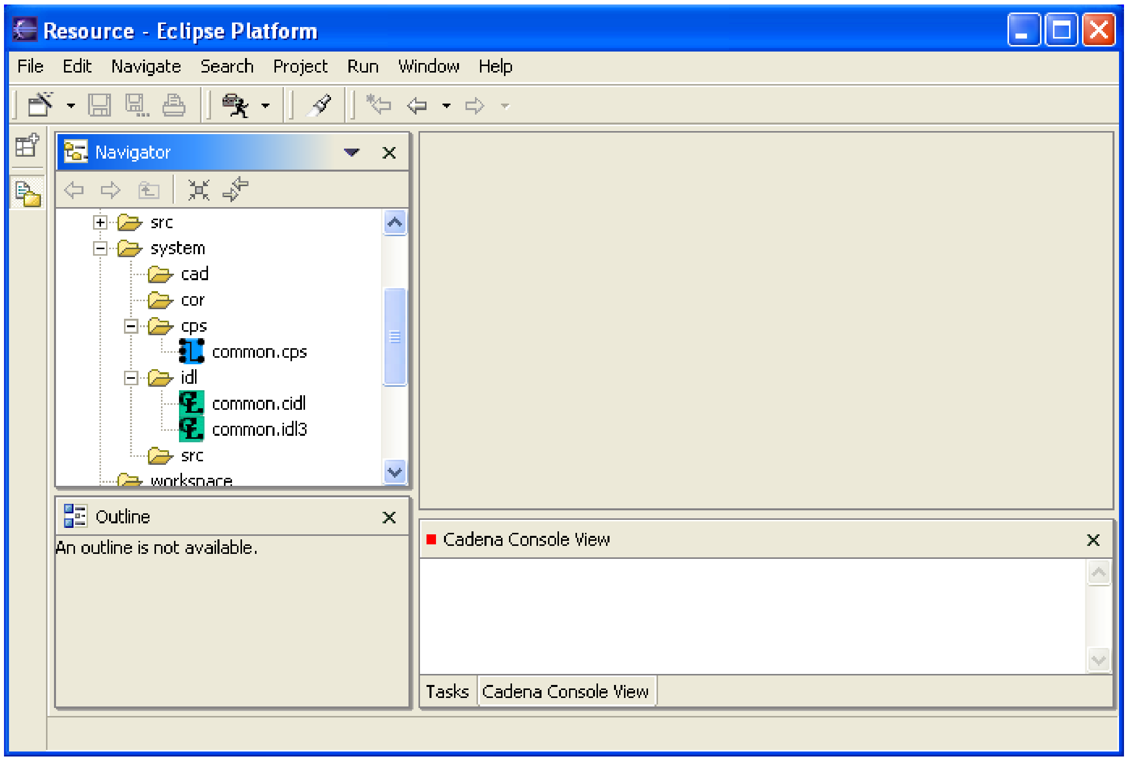
Tip

The Navigator provides support for drag and drop file moves. By dragging and dropping files within the Navigator view, project files can be sorted quickly.

- In the Navigator view, expand the `BasicSP\system` folder.
- Drag `common.cps` and drop it in `BasicSP\system\cps`.
- Drag `common.idl3` and drop it in `BasicSP\system\idl`.

The Navigator view should now resemble the following figure:

Figure 1.6. Sorted common files



An outline is not available.

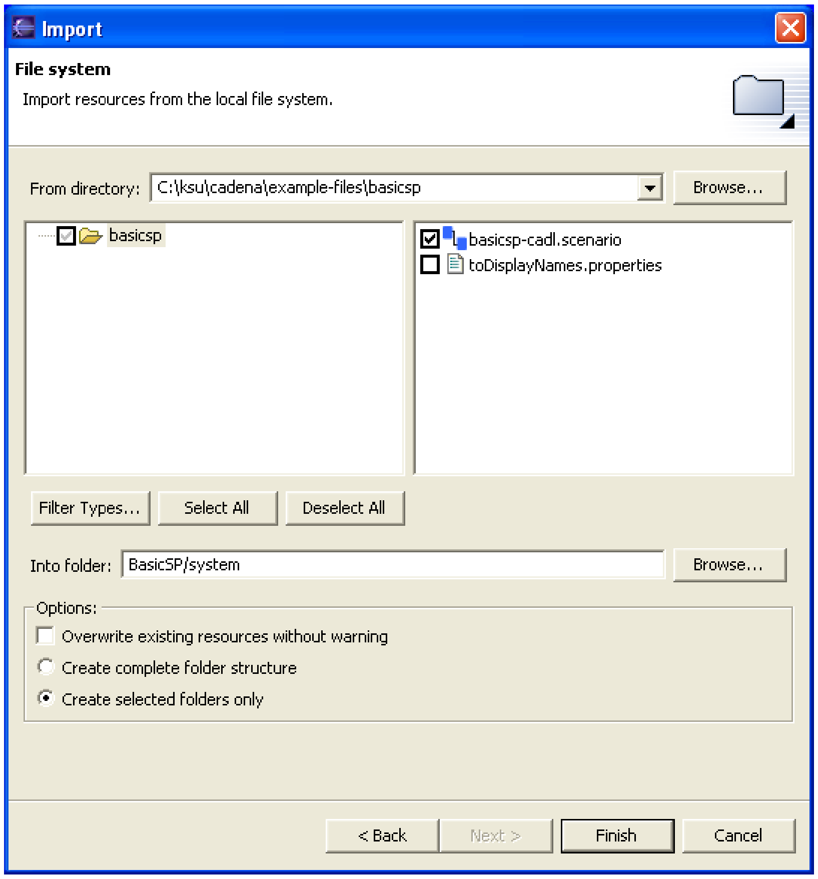
Importing the `basicsp-cadl.scenario` file is done similarly. To import the file:

- In the Navigator View, expand all folders.
- Select the `BasicSP\system\cad` folder.
- Click the right mouse button, Select "Import..."
- Select "File system" as the import source.
- Select "Next".
- For the directory, select `[Cadena installation]\example-files\basicsp`.

Tip

If working in Linux, Eclipse doesn't always clear the text area when you switch directories, so you may have to delete text if it appears in the text area.

Figure 1.7. Importing the scenario file



File system

Import resources from the local file system.

From directory: C:\ksu\cadena\example-files\basicsp

basicsp

[✓] basicsp-cadl.scenario

[ ] toDisplayNames.properties

Into folder: BasicSP/system

Options:

[ ] Overwrite existing resources without warning

( ) Create complete folder structure

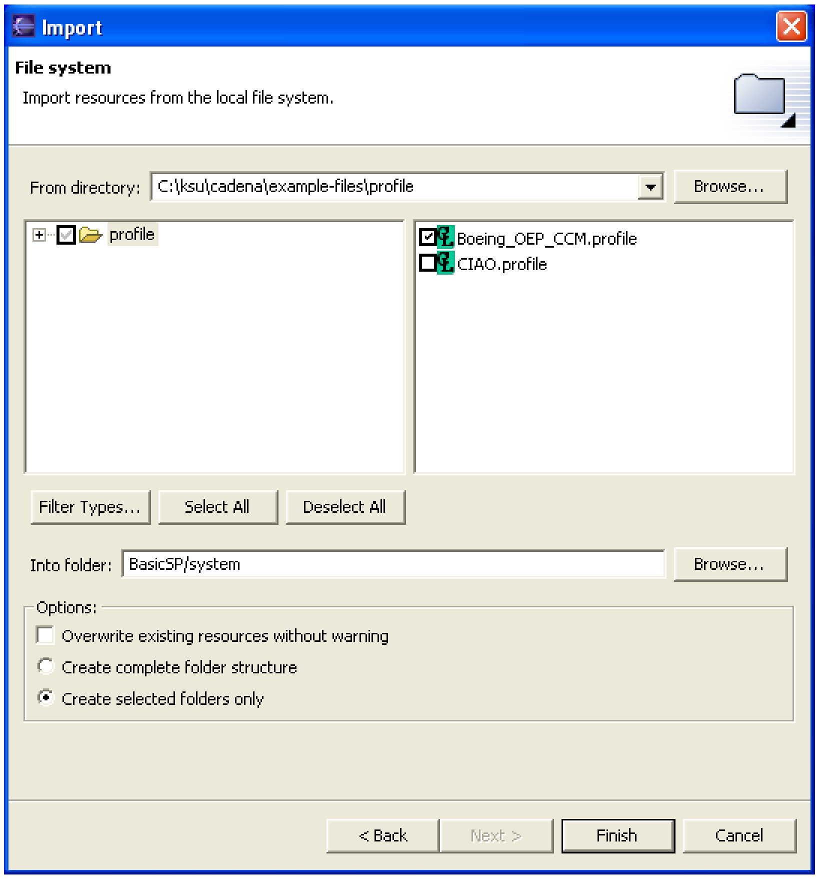
(●) Create selected folders only

- Select `basicsp-cadl.scenario`.
- Select "Finish" to import the selected file.

Finally, the `Boeing_OEP_CCM.profile` needs to be imported. Again, this is done in a similiar fashion:

- In the Navigator View, expand all folders.
- Select the `BasicSP\system\cad` folder.
- Click the right mouse button, Select "Import..."
- Select "File system" as the import source.
- Select "Next".
- For the directory, select `[Cadena installation]\example-files\profiles`.

Figure 1.8. Importing the profile file



From directory: C:\ksu\cadena\example-files\profile

profile

[✓] Boeing_OEP_CCM.profile
[ ] CIAO.profile

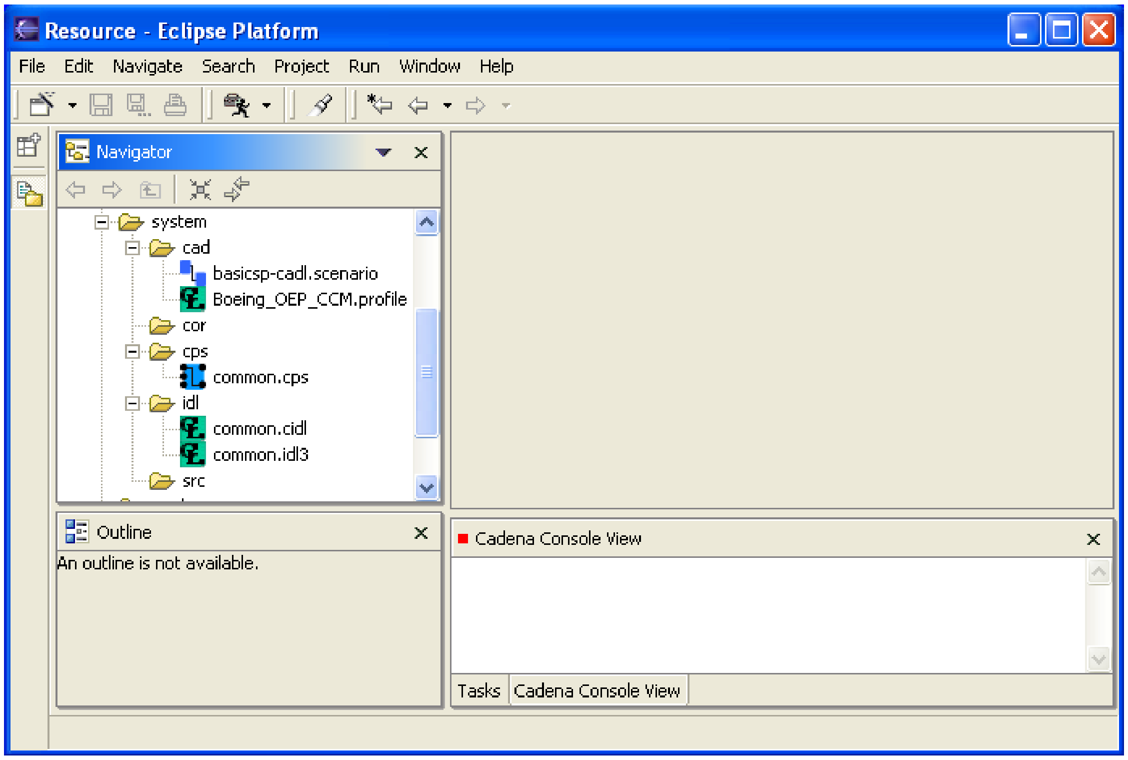
Into folder: BasicSP/system

Options:
[ ] Overwrite existing resources without warning
( ) Create complete folder structure
(●) Create selected folders only

- Select `Boeing_OEP_CCM.profile`.
- Select "Finish" to import the selected file.

The Navigator view in the following figure illustrates a project that is set up correctly.

Figure 1.9. A correctly setup project



An outline is not available.

Chapter 2. IDL Editor

The IDL editor is used to edit and compile IDL3 files. IDL3 files are compiled using OpenCCM. OpenCCM is an open-source compilation and generation tool for CORBA component model specifications. Additional OpenCCM information is available from the OpenCCM website [http://www.objectweb.org/openccm/].

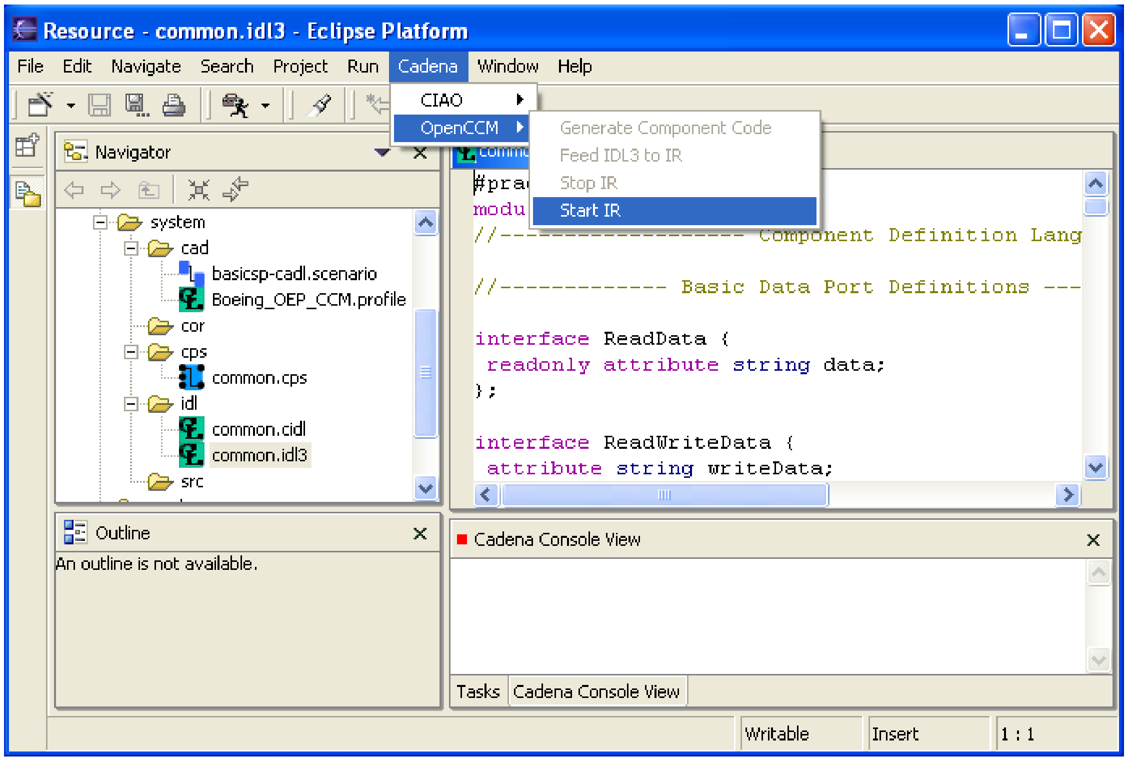
OpenCCM utilizes an Interface Repository (IR) to compile the IDL3 files.

Starting the Interface Repository

The IR must be started before the IDL3 files can be compiled.

1. Open "common.idl3".
2. Make sure the focus is on the common.idl3 file editor.
3. Select Cadena->OpenCCM->Start IR from the menu bar.

Figure 2.1. Starting the IR



An outline is not available.

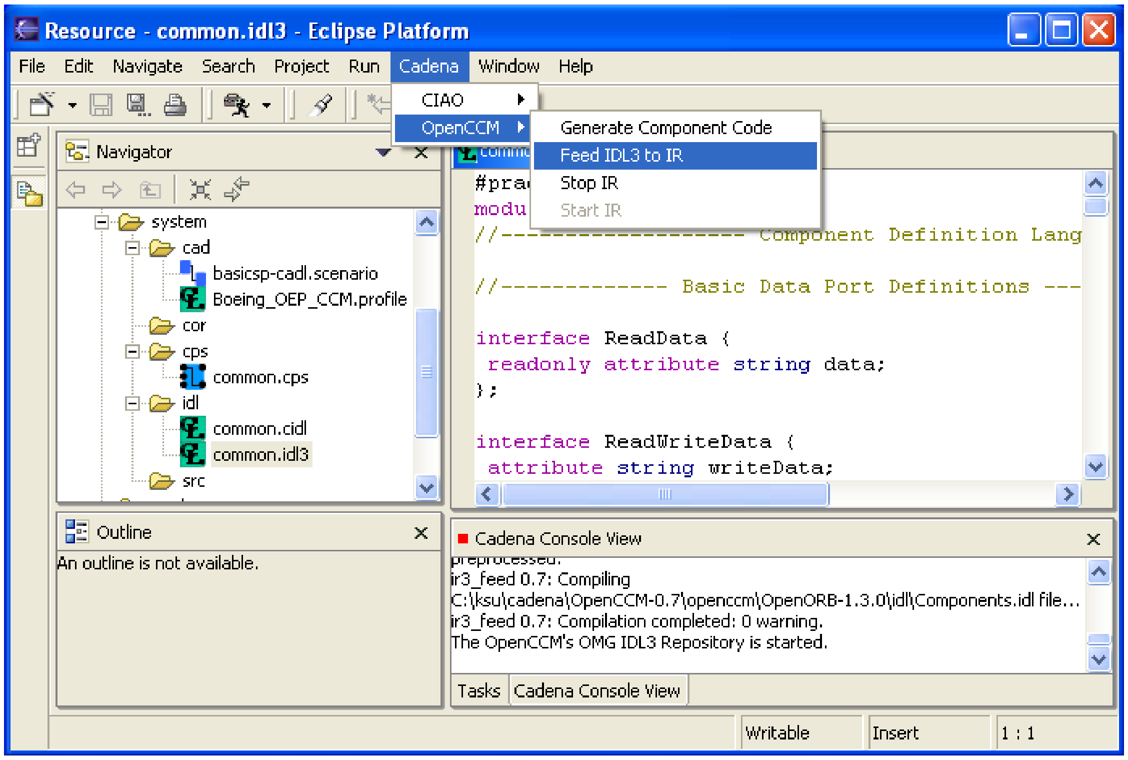
```
#pra
modu
//------------------- Component Definition Lang

//------------- Basic Data Port Definitions ---

interface ReadData {
 readonly attribute string data;
};

interface ReadWriteData {
 attribute string writeData;
```

Compiling IDL Files

To load common.idl3 into the IR:

1. Make sure the focus is on the common.idl3 file editor.
2. Select Cadena->OpenCCM->Feed IDL3 to IR from the menu bar.

Figure 2.2. Compiling IDL files



```
#pra
modu
//-------------------- Component Definition Lang

//------------- Basic Data Port Definitions ---

interface ReadData {
 readonly attribute string data;
};

interface ReadWriteData {
 attribute string writeData;
```

```
preprocessed.
ir3_feed 0.7: Compiling
C:\ksu\cadena\OpenCCM-0.7\openccm\OpenORB-1.3.0\idl\Components.idl file...
ir3_feed 0.7: Compilation completed: 0 warning.
The OpenCCM's OMG IDL3 Repository is started.
```

Stopping the Interface Repository

Once you have finished using Cadena you can then stop the IR.

Note

You may also need to stop and then restart the IR if you make any changes to the IDL.

1. Make sure the focus is on the common.idl3 file editor.
2. Select Cadena->OpenCCM->Stop IR from the menu bar.

OpenCCM Code Generation

Generating Component Code

Stubs, skeletons, and implementations can be generated for the the components that are currently loaded in the IR.

1. Make sure the focus is on the common.idl3 file editor.
2. Select Cadena->OpenCCM->Generate Component Code from the menu bar.

The generated files are placed in different locations depending on what kind of file they are. As seen in the figure above, stubs are placed within "BasicSP\generated\stubs", skeletons are placed within "BasicSP\generated\skeletons", and the implementations are placed with in "BasicSP\src".

Editing Implementation Classes

Business logic may now be inserted into the monolithic implementation code that was just generated. Before the business logic can be applied, however, the home implementation code that was also generated must be modified. That is because the home implementation code manages the components and creates the monolithic implementations. Inside each home implementation, a `create` method must return a monolithic implementation. The following figures show the modification of the home implementation file for the BMDevice component. Note that the original method returns null, while the updated method returns an actual implementation.

Example 2.1. Original `create` method

```
//
//  IDL:cadena/common/CCM_BMDeviceHomeImplicitcreate:1.0
//
/**
 **  Implementation of the ::common::CCM_BMDeviceHomeImplicit::create operation.
 **/
public org.omg.Components.EnterpriseComponent
create()
throws org.omg.Components.CCMException
{
    //
    //  TODO : implement
    //
    return null;
}
```

Example 2.2. Updated `create` method

```
//
//  IDL:cadena/common/CCM_BMDeviceHomeImplicitcreate:1.0
//
/**
 **  Implementation of the ::common::CCM_BMDeviceHomeImplicit::create operation.
 **/
public org.omg.Components.EnterpriseComponent
create()
throws org.omg.Components.CCMException
{
    return new BMDeviceMonolithicImpl();
}
```

Similiar modifications need to be made to all of the home implementation files.

Chapter 3. Scenario Editor

Scenario files describe a system of inter-connected component instances.

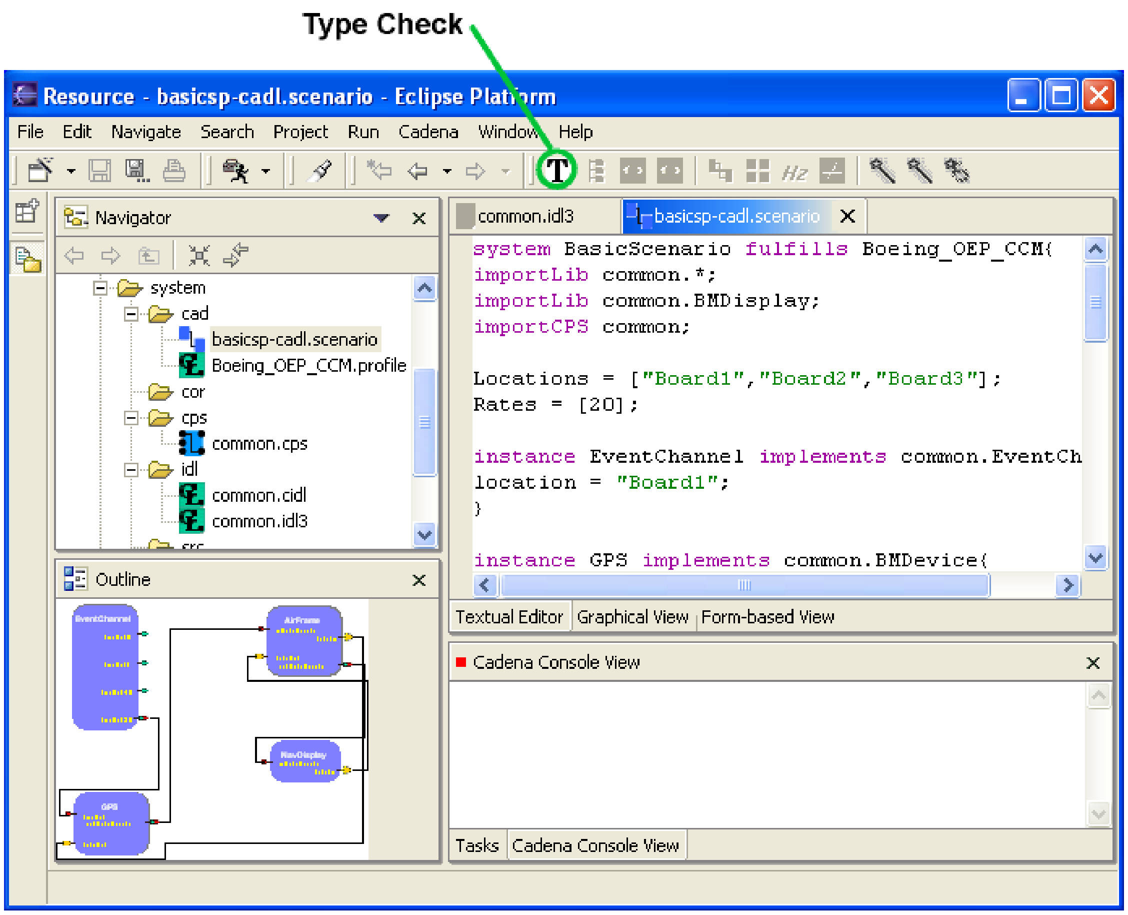
If the IDL3 file has been properly fed into the IR, the scenario editor can be opened by double-clicking on `basicsp-cadl.scenario` in the Navigator view. The scenario editor consists of three tabbed views representing the same scenario. These three tabbed panes provide textual, graphical, and form-based representations of the scenario. When a scenario is changed in one pane, the other two panes are updated to reflect any changes.

Tip

Any one of the scenario editor representations can be maximized by double-clicking the editor title.

Textual Editor

Figure 3.1. Textual Scenario Editor



```
system BasicScenario fulfills Boeing_OEP_CCM{
importLib common.*;
importLib common.BMDisplay;
importCPS common;

Locations = ["Board1","Board2","Board3"];
Rates = [20];

instance EventChannel implements common.EventCh
location = "Board1";
}

instance GPS implements common.BMDevice{
```

Editing Text Files

The textual editor displays the scenario file in text form. When editing the scenario from the textual editor pane, the type checking button is enabled. Selecting this button will type check the scenario and display any errors in the task view. If the task view is not shown, it may be opened by performing the following:

1. Select Window->Show View->Other from the menu.
2. Select Basic->Tasks from the dialog that is displayed.

The scenario editor depends on the IDL editor because IDL3 files define the components that are used in scenarios.

Example 3.1. BasicSP scenario

```
system BasicScenario fulfills Boeing_OEP_CCM {                   ❶
  importLib common.*;                                            ❷
  importCPS common;                                              ❸

  Locations = ["Board1","Board2","Board3"];                      ❹
  Rates = [20];

  instance GPS implements common.BMDevice{                       ❺
    location = "Board1";                                         ❻
  }

  instance EventChannel implements common.EventChannel{
    location = "Board1";
  }

  instance AirFrame implements common.BMClosedED{
    location = "Board2";
  }

  instance NavDisplay implements common.BMDisplay{
    location = "Board3";
  }

  connection(EventChannel.timeOut20 to GPS.timeOut) {            ❼
    runRate = 20;                                                ❽
    synchronously = true;
  }

  connection(GPS.outDataAvailable to AirFrame.inDataAvailable) {
    runRate = 20;
  }

  connection(AirFrame.dataIn to GPS.dataOut) {
  }

  connection(AirFrame.outDataAvailable to NavDisplay.inDataAvailable) {
    runRate = 20;
  }

  connection(NavDisplay.dataIn to AirFrame.dataOut) {
  }
}
```

❶ The first thing to note here is that all scenarios must be enclosed within a `system` block. A `system` must also have an identifier. In this example the identifier is "BasicSP". The name of the scenario file does not need to be the same as the system name.

The `fulfills` keyword is used to declare which profile this scenario is using. Profiles specify the valid properties that can be used in a scenario. This particular scenario fulfills the `Boeing_OEP_CCM` profile. For more on profiles, please see Chapter 5, *Profile Editor*.

❷ The first keyword inside the system block is `importLib`. This keyword tells the system which components it needs to import from the IR. If the component is not imported, it cannot be used in the scenario. Groups of components can be imported by using "<module name>.*", or individual components can be imported by giving their full name, such as "common.EventChannel". In this example, all of the components that are contained within the "common" module are being imported.

❸ The `importCPS` keyword specifies which CPS files the system is dependent on. For every importCPS statement, there should be a corresponding file in the "`system/cps`" directory. Since this example imports the "common" CPS file, there should be a CPS file called "`common.cps`" within the "`system/cps`" directory.

❹ "Locations" and "Rates" are examples of properties attached to the system. Properties provide a mechanism by which to better describe systems. Valid properties which may be attached to the system are specified in the associated profile file. The association between the system and profile is made by the `fulfills` keyword. In this example the system fulfills the "Boeing_OEP_profile," profile (line 1).

In this example there are two properties: "Locations" and "Rates". "Locations" gives the set of locations upon which the components can be distributed. "Rates" gives the valid run rates of connections.

❺ As the name implies, `instance` represents an instantiation of an IDL defined component. In this example, `GPS` is an instance of the component `BMDevice`. It is important to note that components referenced in the scenario file must be declared in the IDL3 file.

❻ Properties can be attached to instances like they can be attached to systems. In this example the "location" property of `GPS` is assigned the value, "Board1".

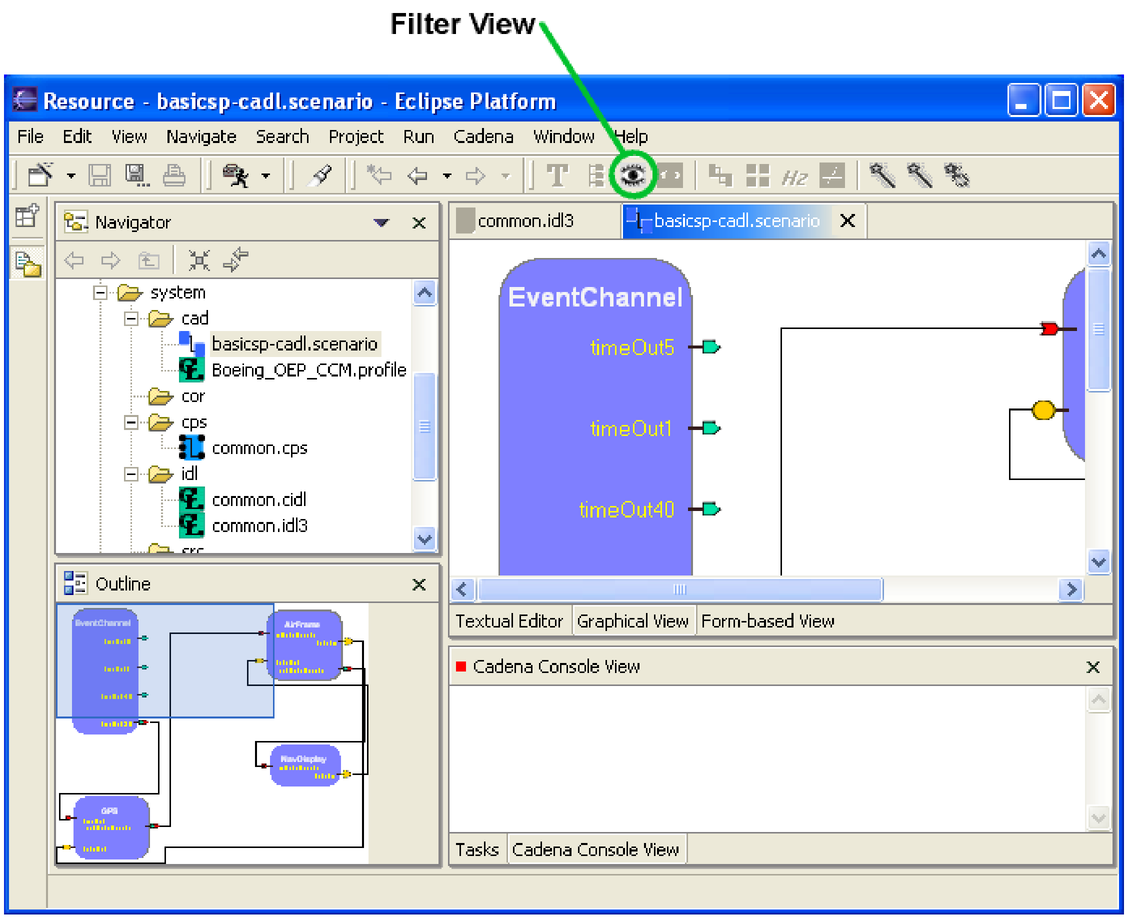
❼ Connections between components are defined in the scenario file by the `connection` keyword. In this example the `timeOut20` port of the `EventChannel` component is connected to the `timeOut` port of the `GPS` component. Both facet/receptacle and event source/event sink connections can be specified.

❽ As with systems and instances, connections may also have properties attached to them. In this example the connection has assigned the property "runRate" a value of 20.

Graphical View

The graphical view presents the scenario file as a diagram of components connected through either event or data connections. This view also provides the opportunity to query the model on several different levels.

Figure 3.2. Graphical Scenario View



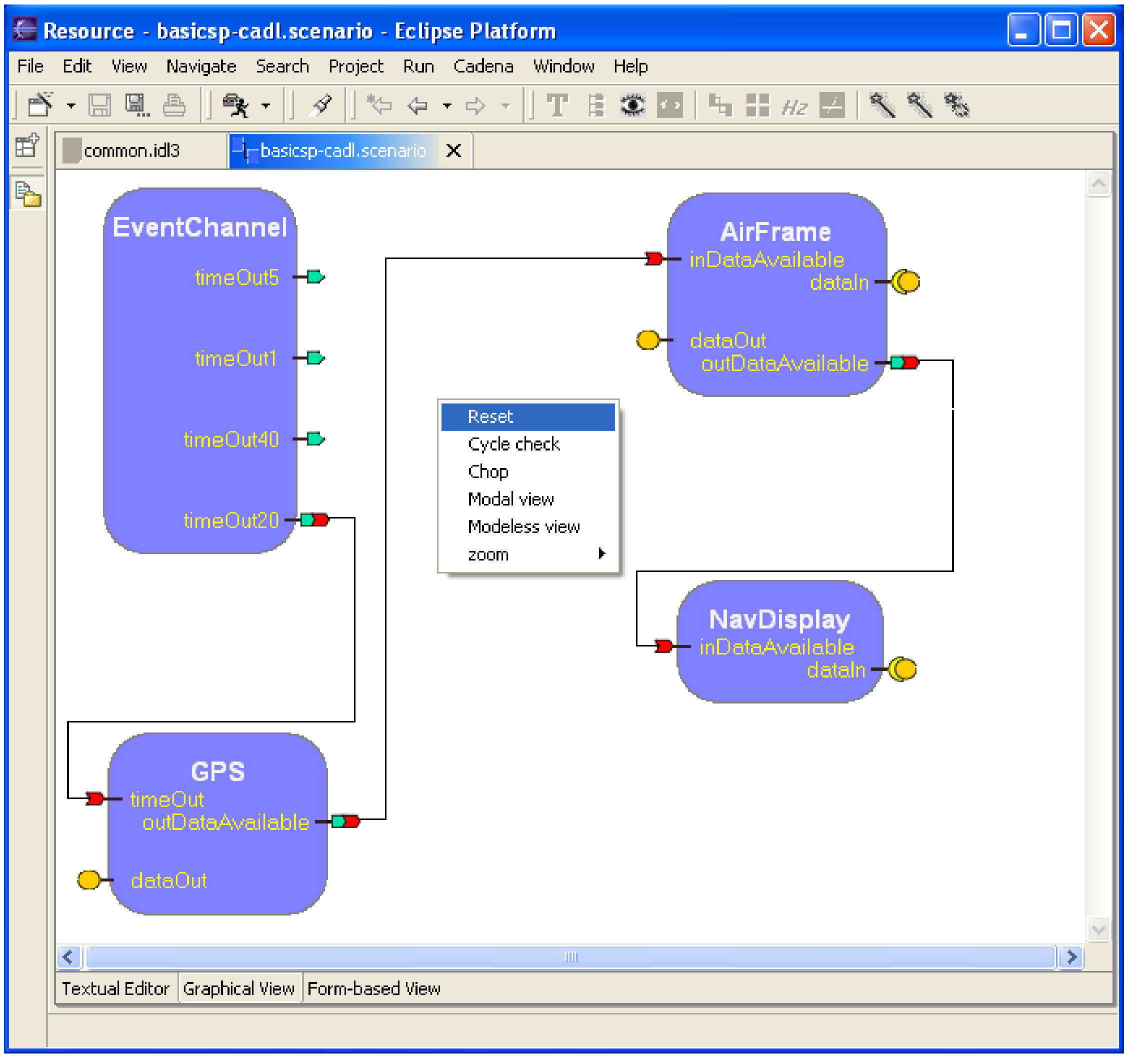
Resetting the Graphical View

When an analysis function is performed (such as a cycle check or a chop), some of the components and connections may be highlighted in the graphical view. The components will remain highlighted until the view is explicitly reset. This allows multiple analysis functions to be overlaid on top of each other. For instance, it would be possible to see the effects of both the forward and backward slice of a component by simply performing the two analysis's without resetting the view.

When the graphical view does need to be reset, perform the following steps:

1. In the graphical view, right click anywhere.
2. Select "Reset".

Figure 3.3. Resetting the Graphical View

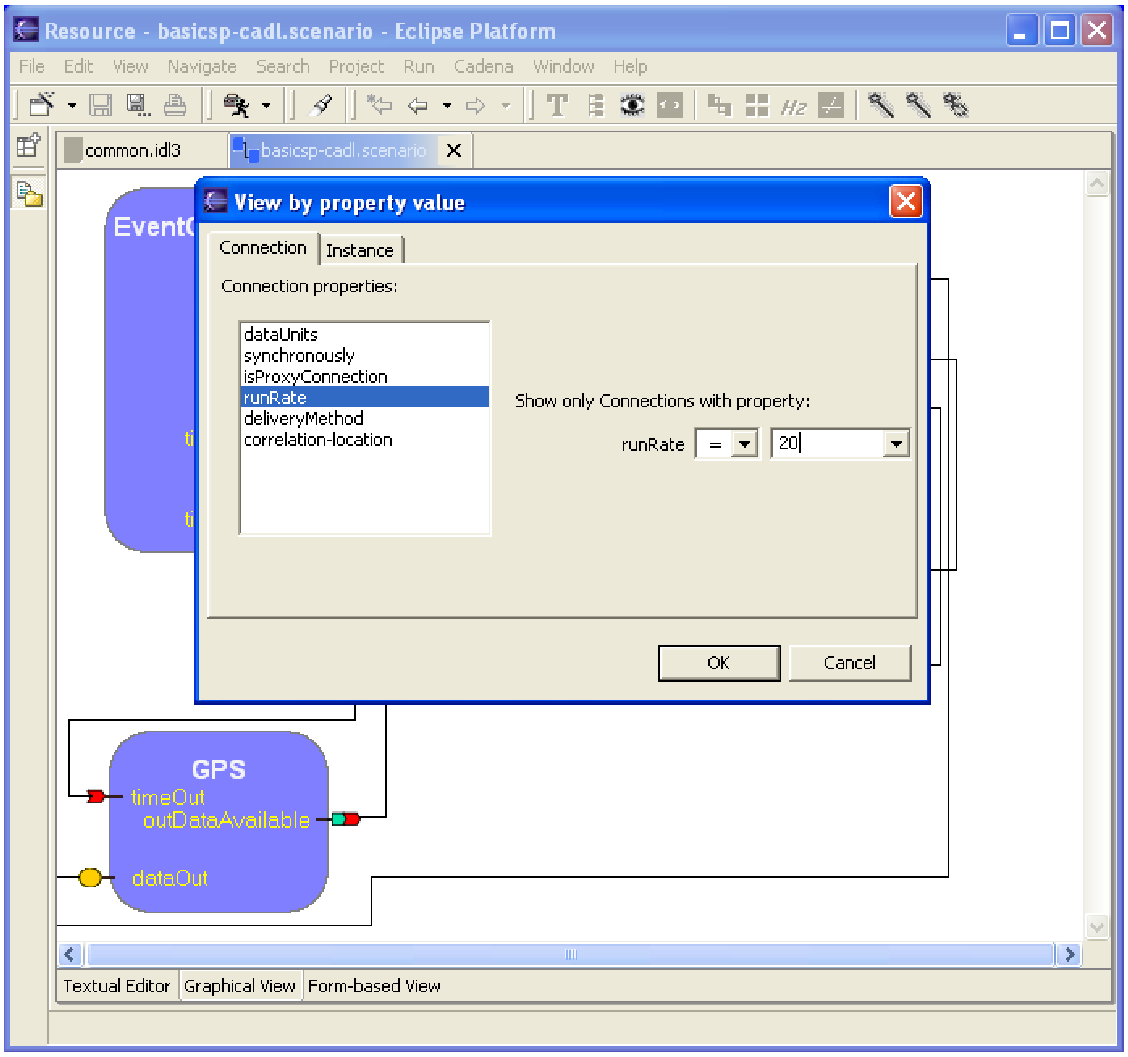


Filtering the View

When dealing with large scenarios, the graphical view may become overwhelming due to number of components and connections that are shown. To help with this problem, the graphical view can be filtered so that only certain component instances and connections are visible. The filter operates on the values of the properties. To filter the graphical view:

1. Select the "Filter" action from the toolbar.
2. Use the filter dialog to create the filter. Filters can be applied to connections or instances.
3. Click the "Ok" button. The graphical view should now be updated to reflect the filter.

Figure 3.4. Filtering the View



Connection properties:

dataUnits
synchronously
isProxyConnection
runRate
deliveryMethod
correlation-location

Show only Connections with property:

runRate = 20

Checking for Cycles

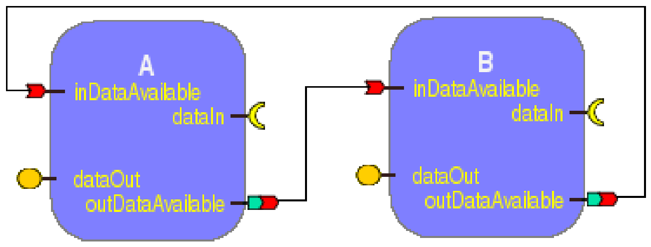
One of the features of the graphical view is the ability to check the scenario for the occurence of cycles. A cycle occurs when the system is configured in such a way that a port can trigger a chain of events which result in the original port being triggered again.

Example 3.2. A Scenario With a Cycle

Consider the following example. Assume that when components A and B receive an event on their inDataAvailable ports, they send an event out from their outDataAvailable ports:

A.outDataAvailable -> B.inDataAvailable -> B.outDataAvailable -> A.inDataAvailable -> A.outDataAvailable.

Since A.outDataAvailable can effectively trigger itself, a cycle exists.



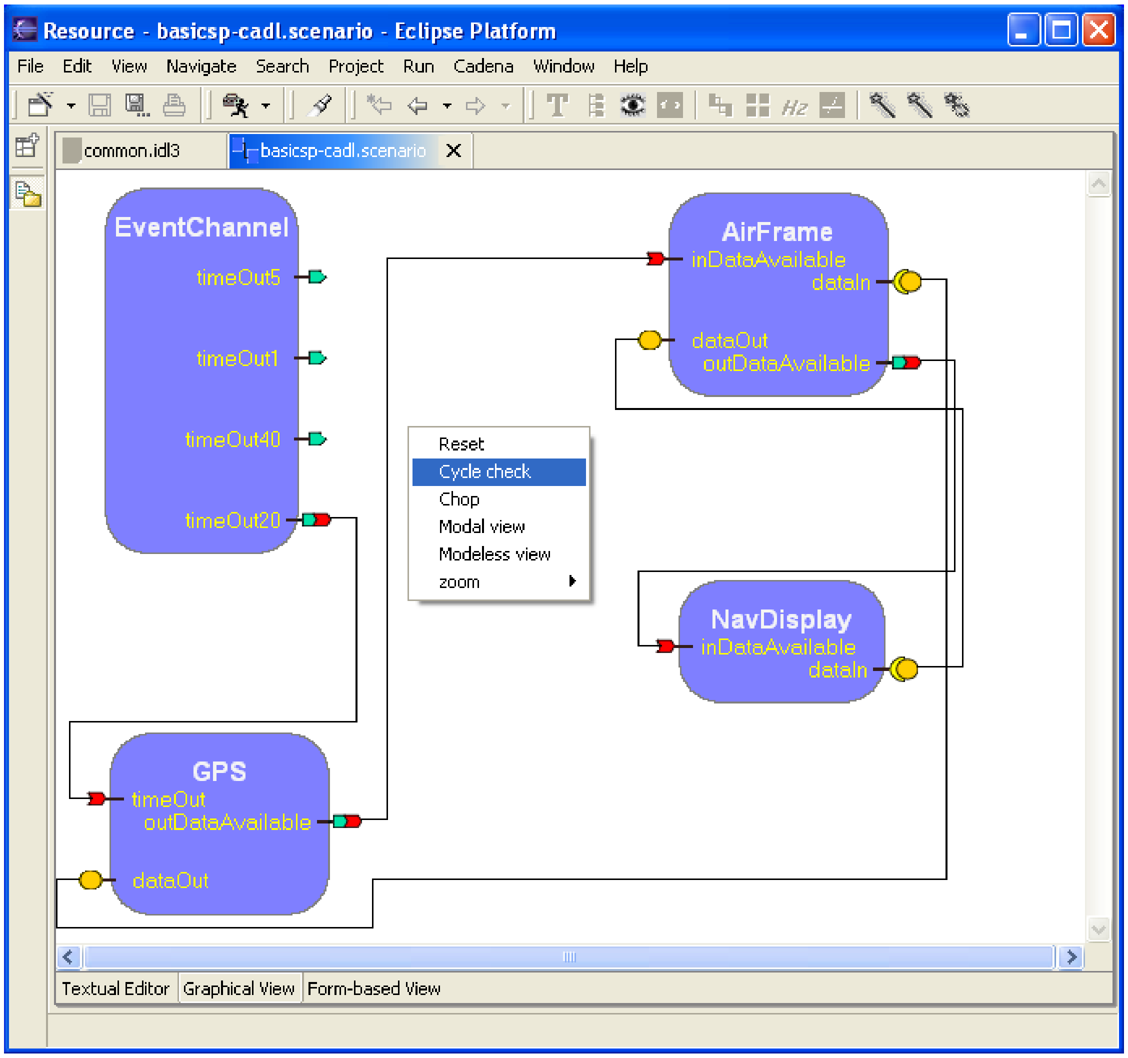
Note

The dependencies between ports of different instances of components is obvious via connections between them. However, information such as the outputs triggered by an input port in a particular mode is not obvious. These kinds of intracomponent dependencies are specified in Cadena Properties Specification (CPS) files. If CPS files are not available during the analysis, then a worst-case assumption is made. This worst-case assumption is that any input port can trigger all output ports. For more information please see Chapter 4, *CPS Editor*.

Follow these steps to check the scenario for cycles:

1. From the graphical view, right click a component.
2. Select "Cycle check".
3. A cycle should NOT be detected.

Figure 3.5. Checking for Cycles



Slicing

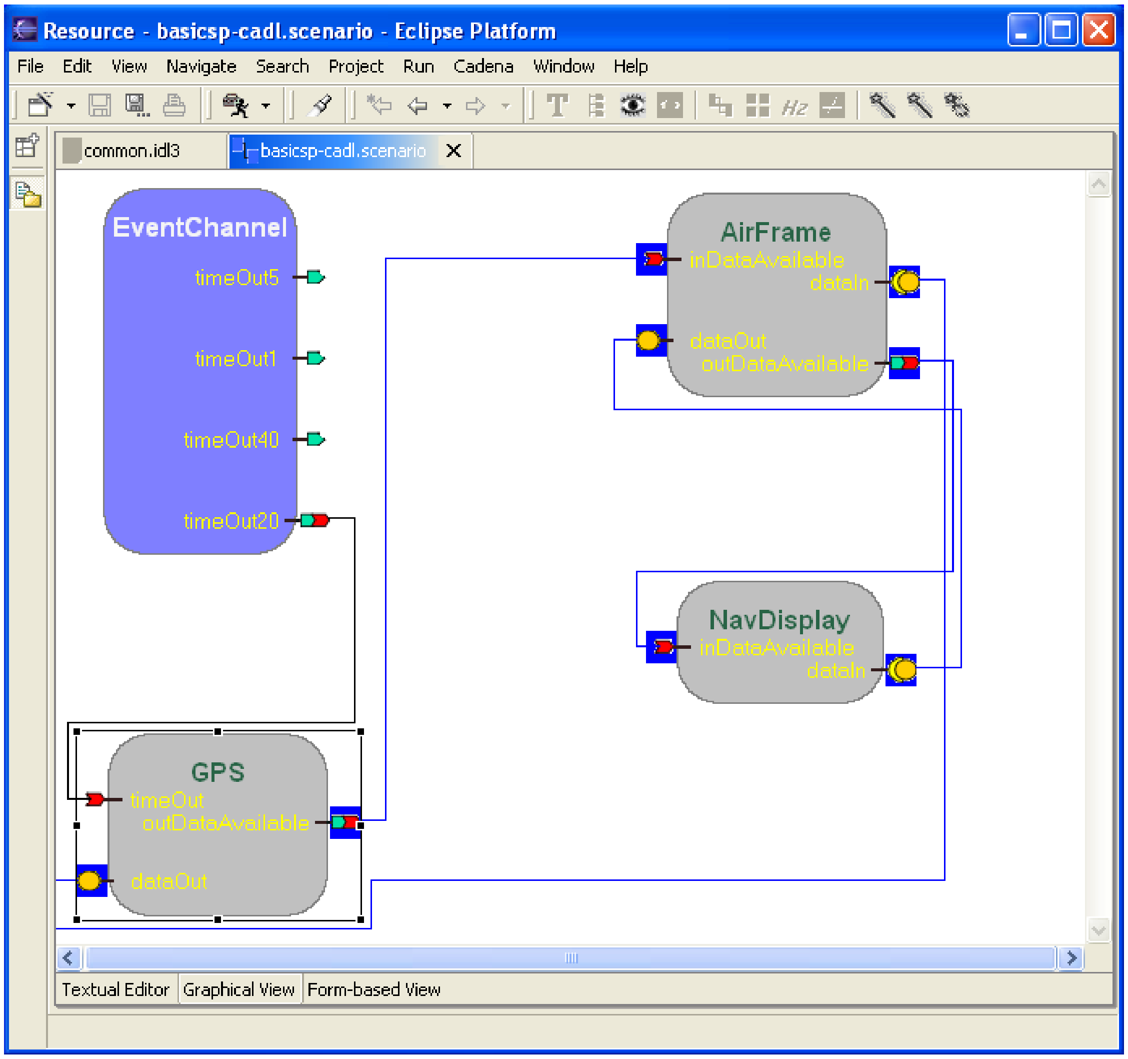
Now try a forward port-level slice:

1. Right click the GPS component.
2. Select "Forward port-level slice".
3. Select "outDataAvailable".
4. Select "OK".

All ports and components downstream of GPS.outDataAvailable have been highlighted. We can follow the forward port-level slice from GPS.outDataAvailable to NavDisplay.dataIn by examining the scenario file in the textual editor. The numbers in the figure below show which connections are affected by this forward port-level slice.

Backward slicing, the reverse notion of forward slicing, is also supported by the graphical view. While forward slicing calculates all ports that are affected by a given port, backward slicing calculates all ports that may affect the given port. The following figure illustrates the result of performing a backward slice from the EventChannel component.

Figure 3.6. The results of a forward port level slice



Chopping

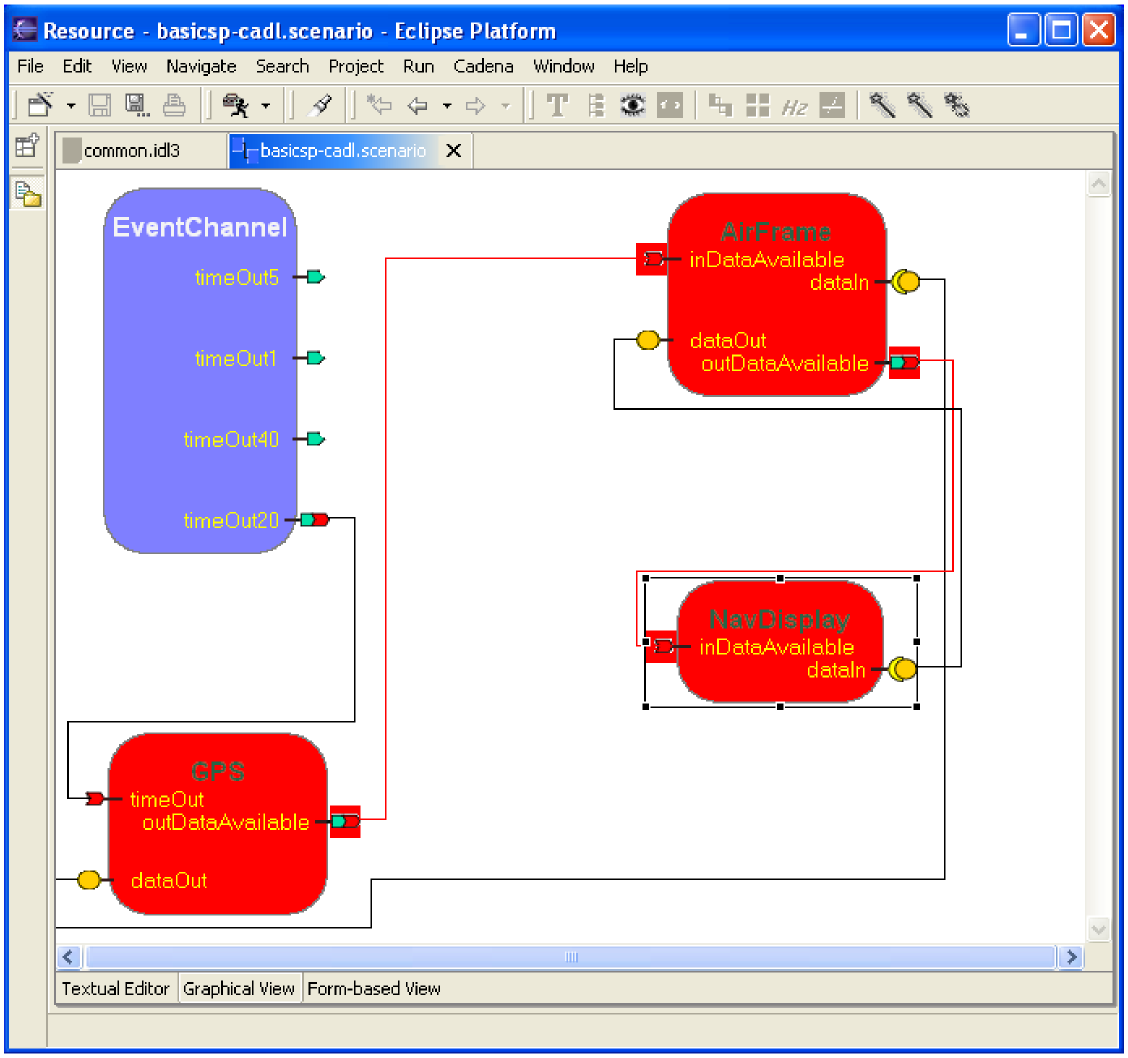
Once a pair of ports have been designated as the starting and ending points, chopping will provide the set of ports that occur along the path between this given pair of ports.

1. Right click the GPS component.
2. Select "Pick chop start".
3. Select "outDataAvailable".
4. Select "OK".

5. Right click the NavDisplay component.
6. Select "Pick chop end".
7. Select "inDataAvailable".
8. Select "OK".
9. Right click within the graphical view.
10. Select "Chop".

The following figure should mirror the result

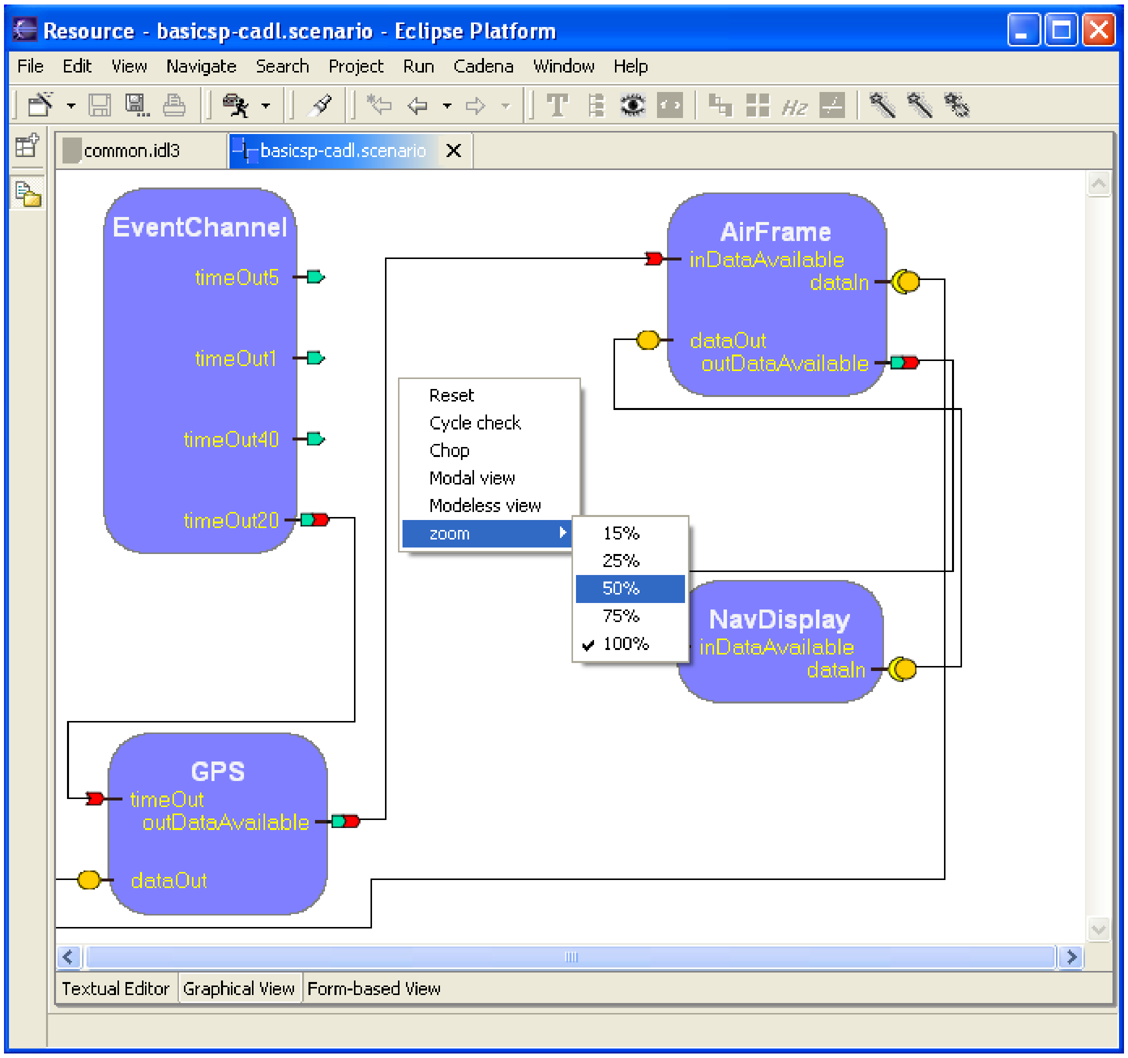
Figure 3.7. The result of a chop



Zooming

When using the graphical view, the outline view displays the complete graphical workspace so that it is easier to maneuver if the project has many components and connections. The graphical view also has a zoom feature, which can be used by right-clicking within the editor window and selecting a percentage.

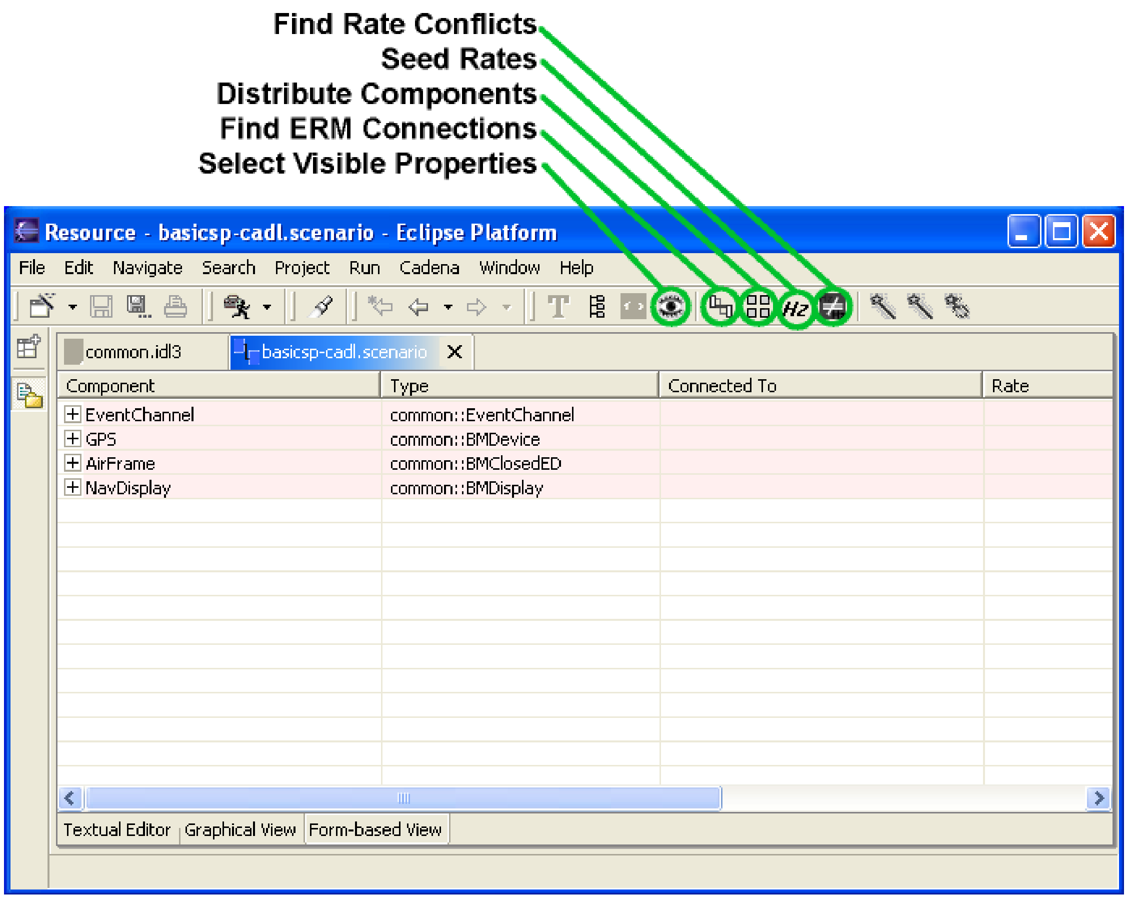
Figure 3.8. Zooming



Form-based Editor

In the form-based representation of the scenario, information about components and connections is presented as a table (spreadsheet). All of the connections that each component maintains are displayed in this view, whereas the textual editor lists a connection under only one of the components.

Figure 3.9. Form-based Editor



Component	Type	Connected To	Rate
EventChannel	common::EventChannel		
GPS	common::BMDevice		
AirFrame	common::BMClosedED		
NavDisplay	common::BMDisplay		

Editing the Form

Unlike the graphical representation (which is just a view), this representation of the scenario is a full-fledged editor. Components and connections can be added and/or deleted in the form-based view. It is possible to add imports, too. The connections appear as children in a tree routed below a component. To toggle the expansion of all nodes in the table, click on the "Toggle branch expansion" button in the toolbar.

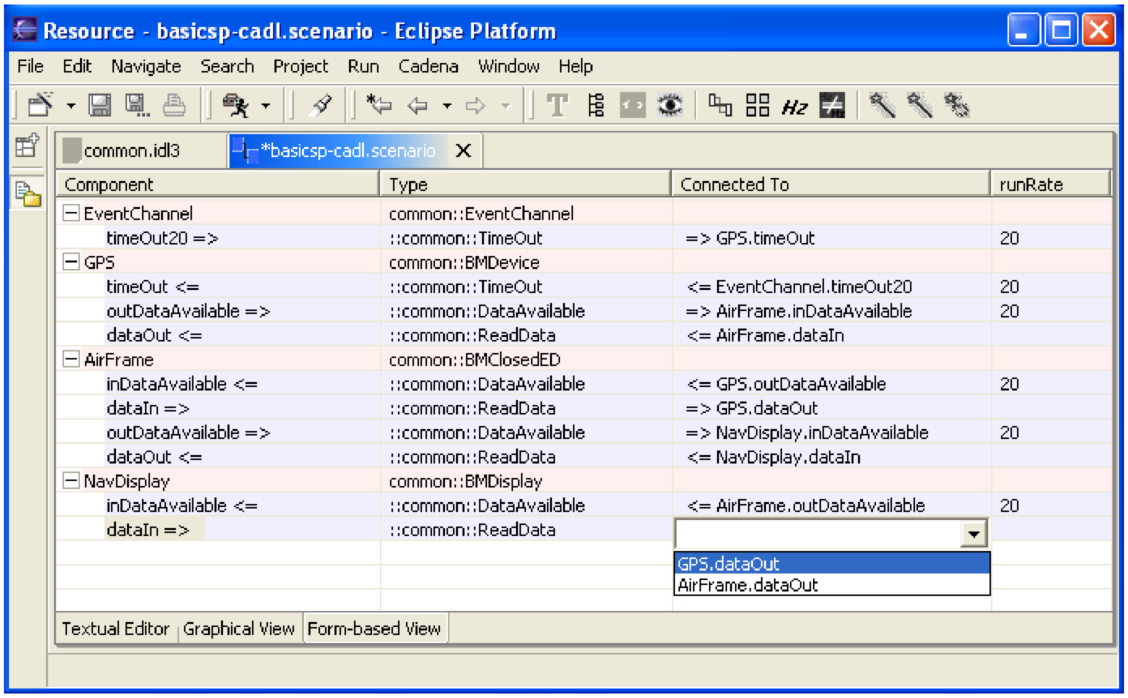
Upon adding a connection, the unspecified end of the connection is defaulted. This connection can be edited by focusing on the connection in the "Component/Port" column and double-clicking in the "Connected to" column in the same row. All ports to which the given port may possibly be connected to will appear. After a new port has been selected, pressing "return" will confirm the change. Likewise, the "Delivery method" associated with a non-data connection can be changed.

Each component is associated with a location. This can be edited by double-clicking in the "Location" column after focusing on the row of interest. Upon entering a valid location and pressing "return", the model is updated. If the location did not exist in the model, then it is added to the model. Similar actions can be performed to set the rates associated with connections.

Note

Changes made to import statements, rates, and locations (in the form-based view or textual editor) are not visible in the graphical view, but do occur.

Figure 3.10. Changing a port connection



Component	Type	Connected To	runRate
EventChannel	common::EventChannel		
timeOut20 =>	::common::TimeOut	=> GPS.timeOut	20
GPS	common::BMDevice		
timeOut <=	::common::TimeOut	<= EventChannel.timeOut20	20
outDataAvailable =>	::common::DataAvailable	=> AirFrame.inDataAvailable	20
dataOut <=	::common::ReadData	<= AirFrame.dataIn	
AirFrame	common::BMClosedED		
inDataAvailable <=	::common::DataAvailable	<= GPS.outDataAvailable	20
dataIn =>	::common::ReadData	=> GPS.dataOut	
outDataAvailable =>	::common::DataAvailable	=> NavDisplay.inDataAvailable	20
dataOut <=	::common::ReadData	<= NavDisplay.dataIn	
NavDisplay	common::BMDisplay		
inDataAvailable <=	::common::DataAvailable	<= AirFrame.outDataAvailable	20
dataIn =>	::common::ReadData		

GPS.dataOut
AirFrame.dataOut

Changing the Visible Properties

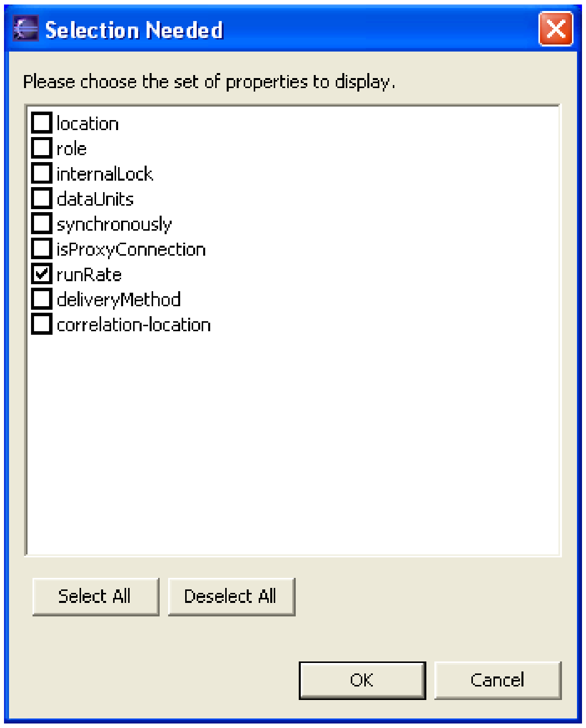
At certain times, it may be desirable to hide certain properties that aren't of interest. For instance, the developer may only be interested in seeing the run rates of connections. This can be done by:

1. Select the "Select visible properties" action from the toolbar.
2. Use the filter dialog to select which properties to display. All properties that can be applied to instances and connections are displayed in the selection dialog.
3. Click the "Ok" button. The editor should be updated such that only the properties that were selected are visible.

Note

The "Component", "Type", and "Connected To" columns are always visible.

Figure 3.11. Selecting the visible properties



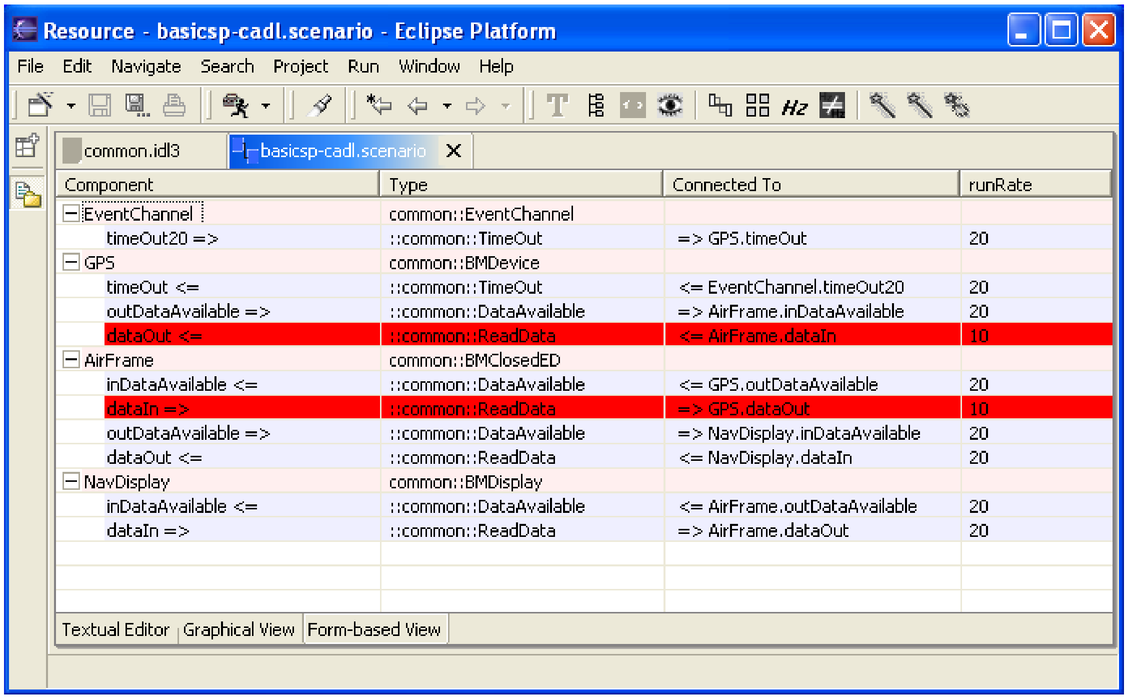
Please choose the set of properties to display.

- [ ] location
- [ ] role
- [ ] internalLock
- [ ] dataUnits
- [ ] synchronously
- [ ] isProxyConnection
- [✓] runRate
- [ ] deliveryMethod
- [ ] correlation-location

Seeding Rates

It is possible to have a system where a port deals with connections that have differing rates. These connections can be detected by pressing the "Find rate conflicts" button that is on the toolbar. These connections and all connections affected by such connections are then highlighted in RED. Conflicts of this type can be easily resolved by using the automatic rate seeding feature. This feature can be invoked by clicking the "Seed rates" button in the toolbar. The tool will seed rates in a rate monotonic fashion. For this to work successfully, however, the connections involving trigger ports in the system should have their rates specified.

Figure 3.12. Detecting Rate Conflicts


Resource - basicsp-cadl.scenario - Eclipse Platform
File Edit Navigate Search Project Run Window Help
common.idl3
basicsp-cadl.scenario
Textual Editor
Graphical View
Form-based View

Component	Type	Connected To	runRate
EventChannel	common::EventChannel		
timeOut20 =>	::common::TimeOut	=> GPS.timeOut	20
GPS	common::BMDevice		
timeOut <=	::common::TimeOut	<= EventChannel.timeOut20	20
outDataAvailable =>	::common::DataAvailable	=> AirFrame.inDataAvailable	20
dataOut <=	::common::ReadData	<= AirFrame.dataIn	10
AirFrame	common::BMClosedED		
inDataAvailable <=	::common::DataAvailable	<= GPS.outDataAvailable	20
dataIn =>	::common::ReadData	=> GPS.dataOut	10
outDataAvailable =>	::common::DataAvailable	=> NavDisplay.inDataAvailable	20
dataOut <=	::common::ReadData	<= NavDisplay.dataIn	20
NavDisplay	common::BMDisplay		
inDataAvailable <=	::common::DataAvailable	<= AirFrame.outDataAvailable	20
dataIn =>	::common::ReadData	=> AirFrame.dataOut	20

Distributing Components

In the form-based view, it is possible to assign locations to a few components, then ask the tool to distribute the unassigned components among the available locations. This is accomplished by clicking the "Distribute the components" button in the toolbar. Note that the distribution will be better when there are fewer connections that have non-zero rates. The reason for this is that the distribution algorithm will try to minimize the inter-location traffic.

Finding ERM Connections

Another feature of the form-based editor is the ability to detect remote event deliveries between collocated components into local dispatches. This feature is available by clicking the "Find ERM connections" button in the toolbar. Again, this feature works best after automated distribution.

OpenCCM Code Generation

Generating Scenario Code

Cadena can generate a Java™ "main" class that will instantiate an OpenCCM system that corresponds to the scenario that was created. To build this class, click on the "generate java code" button on the toolbar.

Generating Build Scripts

The correct scripts must be generated to compile and run the project. This is triggered by pressing the "generate build and run scripts" menu button (see figure 3.1). The run scripts can be found in the "bin" directory, and the build scripts are placed in the BasicSP (project) directory.

Compiling and Running

Once the business logic has been implemented within the monolithic implementations and build scripts have been generated, then the project can be compiled from a separate command window. Go to the IDL editor and stop the IR before completing the following steps:

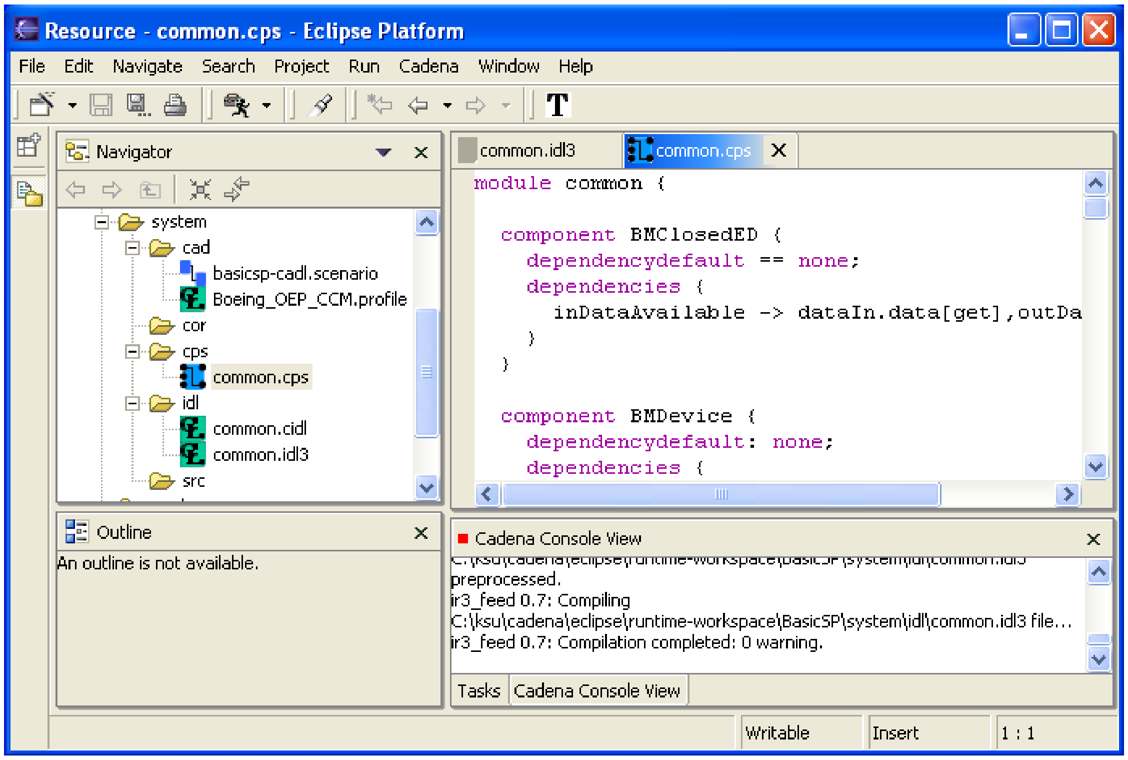
1. Open a command prompt window and change directory to the Cadena installation.
2. The Cadena environment needs to be set up in order to compile the project. For this, type the following at the prompt: "OpenCCM-0.7\openccm\OpenORB-1.3.0\bin\envi_OpenCCM.bat" (for Windows), or ". OpenCCM-0.7/openccm/OpenORB-1.3.0/bin/envi.OpenCCM.sh" (for Linux).
3. Now change the directory to the eclipse workspace. This is the 'workspace' directory under the cadena installation directory.
4. The project created using Cadena will be listed in this directory. Change the directory to the project that has to be compiled.
5. Type "build" ("build.sh" for Linux) at the commmand prompt.
6. Once the build is successful, the implementations can be run using the scripts generated.

To run the project, type "bin\start_java" ("bin/start_java.sh" for Linux) at the command prompt. This goes through the general process of starting the component servers and the name service before running the project itself. The execution of the project can be terminated by running the "bin\stop_java" script ("bin/stop_java.sh" for Linux). These scripts must be run from the [Cadena installation]\workspace\BasicSP directory.

Chapter 4. CPS Editor

CPS files specify various aspects of component behavior, such as intracomponent dependencies between ports and simple behavioral descriptions of a component's event handlers and other methods.

Figure 4.1. CPS Editor



```
module common {

  component BMClosedED {
    dependencydefault == none;
    dependencies {
      inDataAvailable -> dataIn.data[get],outDa
    }
  }

  component BMDevice {
    dependencydefault: none;
    dependencies {
```

```
C:\ksu\cadena\eclipse\runtime-workspace\BasicSP\system\idl\common.idl3
preprocessed.
ir3_feed 0.7: Compiling
C:\ksu\cadena\eclipse\runtime-workspace\BasicSP\system\idl\common.idl3 file...
ir3_feed 0.7: Compilation completed: 0 warning.
```

Editing Text Files

If the IDL3 file has been properly fed into the IR, the CPS editor can be opened by double-clicking on `common.cps` in the navigator view. Currently, the CPS editor provides simple text editing.

The CPS editor does provide a type checking function. The type checker can explicitly be invoked by pressing the type checker button that is in the tool bar. The type checker will also be invoked whenver the CPS file is modified and then saved. All errors will show up in the tasks view. If the tasks view is not visible, it can be opened by:

1. Select Window->Show View->Other from the menu.
2. Select Basic->Tasks from the dialog that is displayed.

CPS Language

The examples presented below describe the structure of these files and illustrate the different aspects of the language.

Example 4.1. A basic CPS file

```
module common {                                                   ❶

  component BMClosedED {                                          ❷
    dependencydefault == none;                                    ❸
  }

  component BMOpenED {
    dependencydefault == all;
  }

}
```

❶ As this first example shows, the contents of CPS files are enclosed within `module` blocks. The `module` keyword is followed by an identifier that names the module. This identifier must match the name of the CPS file. In this example, since the name of this module is "common", the file must be named `common.cps`. In addition to this requirement, there must also be an IDL module with the same name as this CPS module. So for this example, the IDL3 file must contain a module named "common", too.

❷ `component` blocks constitute the body of a `module` block. The `component` keyword is followed by the name of the component, which MUST correspond to a component defined within the IDL3 file. For this example, there must be a component named "common :: BMClosedED" in `common.idl3` since there is a component "BMClosedED" in this CPS file.

❸ The two components in this example both use `dependencydefault`, which may be set to either `all` or `none`. A setting of `none` allows new dependencies to be added to an empty dependency relation. Dependencies do not exist unless they are explicitly declared. A setting of `all` allows all possible dependencies between ports. The dependencies can still be pruned, though. Once a port is mentioned on the left-hand-side (such as `inDataAvailable`), then only declared dependencies apply for that port.

Basic Port Dependencies

Example 4.2. Explicitly defined port dependencies

```
  component BMClosedED {
    dependencydefault == none;
    dependencies {
      inDataAvailable -> dataIn.data[get],outDataAvailable;❶
    }
  }
```

❶ This example shows how to explicitly declare dependencies between ports. The dependence declarations take the form *trigger-port-action* -> *response-port-action*. The trigger-port-action should be either a facet or an event sink. The response-port-action should either be a receptacle or an event source. Example 2 shows that when an event is received by the `inDataAvailable` event sink, it may trigger the `ReadData` interface of another component, and it may publish a `DataAvailable` event on it's `outDataAvailable` event source (the interfaces are defined in the IDL3 file).

Mode Variables

Mode variables come in three flavors, public, private open, and private closed. These categories reflect the differences in how much of a mode variable's definition is visible outside the host component. Consider three different scenarios:

A *public mode variable* is used to tag an IDL attribute as a mode variable. Consider a component "BMSensorProxy" whose IDL has been designed with support for switching the data acquisition on and off by including a simple read/write attribute `enabled` (whose type is the enumeration `OnOffMode`) into BMSensorProxy's definition. This allows any client of a BMSensorProxy instance to manipulate the status of `enabled`. Now the CPS author wishes to tag `enabled` as a mode variable because the runtime behavior of BMSensorProxy shifts according to whether the component is enabled. This scenario calls for the use of a public mode variable:

Example 4.3. Public Mode Variable

```
module common {
    component BMSensoryProxy {
        ....
        mode enabled represents common.BMSensorProxy.enabled
            init common.OnOffMode.enabled;
        ....
    }
}
```

The other two remaining flavors of mode variables are designed to be used when the IDL designer has not provided an attribute to explicitly encode the modality of a component. A *private open mode* variable is declared when one wishes to create a control flow variable whose values range over a publicly known IDL enumerated type. Private mode variables are not accessible outside the component boundary. The "BMLazyActive" component uses a private mode variable of the "open" kind in a typical way: data is either `fresh` or `stale`. The component's externally accessible methods change strategies for supplying data depending on the freshness of the data source. There is, however, no reason to expose the modality of the component to the outside world.

Example 4.4. Private Open Mode Variable

```
module common {
    component BMLazyActive {
        ....
        mode status of common.LazyActiveMode
            init common.LazyActiveMode.stale;
    }
}
```

If there is no predefined IDL enumerated type from which to draw the private mode variable's values, then the enumerated type can be inlined to create a *private closed mode* variable:

Example 4.5. Private Closed Mode Variable

```
module common {
    component BMLazyActive {
        ....
        mode status of { stale, fresh }
            init status.stale;
    }
}
```

Modal Port Dependencies

The port dependencies for components may change to reflect what mode the component is in. For example, the "BMModal" component can be enabled or disabled by using a mode variable. When the component is enabled and an event is received on it's "inDataAvailable", it will fetch some data and send an event out on it's "outDataAvailable" port. When the component is disabled, nothing will happen when an event comes in. The dependencies can be captured with a modal port dependency:

Example 4.6.

```
  component BMModal {

    mode State of common.OnOffMode
        init common.OnOffMode.enabled;

    dependencydefault: none;

    dependencies {
      case State of {
        enabled:
          inDataAvailable -> dataIn.data[get], outDataAvailable;
        disabled:
          inDataAvailable -> ;
      }
    }
  }
```

Chapter 5. Profile Editor

The Profile defines what properties are attached to architectural elements. Properties may be of the following types: integer, boolean, string, or an ordered sequence of integers, booleans, or strings. Properties may be attached to one of the following architectural elements: system, instance, or connection.

Although properties may be either "optional" or "required", it is recommended that optional properties be avoided whenever possible. Required properties guarantee homogenous property sets throughout the model. Common required properties promote application of constraints and thorough analysis of the system.

Properties may be used to describe many different aspects of a system. The following list contains a few of the aspects which may be captured by properties:

- deployment specifics
- quality of service requirements
- managerial information
- testing requirements

Profile Language

The example below demonstrates the structure of the Profile language.

Example 5.1. Simple profile

```
contract Boeing_OEP_CCM{                                                   ❶
  properties::system{                                                      ❷
    optional("Rates", sequence(INT));                                      ❹
    required("Locations", sequence(STRING));
  }

  properties::instance{
    required("location", STRING);                                          ❸
    optional("role", STRING); //either "master" or "proxy"
  }

  properties::connection{
    optional("dataUnits", STRING);
    optional("isProxyConnection", BOOLEAN);  //only for master-proxy connections
    optional("runRate", INT);
  }
}
```

❶ The first thing to notice is that everything is enclosed within a `contract` block. There is also an identifer, *Boeing_OEP_CCM* that declares the name of this contract. The name of the contract should also match the name of the file that contains it. For this contract, the name of the file must be `Boeing_OEP_CCM.profile`.

❹ Contracts are used to specify what properties can be attached to the different architectural elements of the system. The only valid architectural elements are `system`, `instance`, and `connection`. The `properties::system` block contains the set of properties attached to the `system`.

❸ The "required("location",...)" line demonstrates the attachment of a required string property named "location" to all `instance` elements in a Scenario description.

Notice that "location" is declared a `required` property. Required properties must be defined for each `instance` in the scenario. Property types can be specified as optional with the `optional` keyword).

Properties must have a type. The `location` property is defined to be of type `STRING`. When `location` is assigned a value in the scenario, the value must be a string. Examples of valid values for this property include "Board1", "foo", and "x".

❹ The `Rates` property demonstrates the use of a `sequence` type. Sequences are ordered sets of a homogenous type. This particular property type is a `sequence` of `INT`s. Sequences may not be empty.

Examples of valid values for this property include [1,2,3], [0], and [100,-5,0].

Appendix A. Boeing Extensions

Generating Configuration XML

In addition to generating Java code, Cadena can also generate Boeing XML that can be used with OEP build v2.2. To generate the XML, it is necessary to define a "toDisplayNames.properties" file in the "BasicSP\system\src" directory. This can only be done after the IDL3 and scenario files have been written. The toDisplayNames.properties file is used to get the information that is not included in the scenario file. The toDisplayNames.properties file has already been written for the BasicSP example.

To import toDisplayNames.properties:

1. In the Navigator view, select the "BasicSP\system\src" folder.
2. Click the right mouse button, Select "Import..."
3. Select "File system" as the import source.
4. Select "Next".
5. For the directory, select "[cadena installation]\example-files\basicsp".
6. Select "toDisplayNames.properties".
7. Select "Finish" to import the selected file.

The property file provides two functionalities. First, it provides display information for component homes and component instances. Since the scenario file usually uses different instance/component names than the Boeing OEP XML, scenario names must be mapped to the Boeing OEP XML names. This functionality is accomplished within the "toDisplayNames.properties" file. For example, if a component instance is named "DataSource" in the scenario file, and we want to display it as "DATA_SOURCE" in the Boeing OEP XML file, then the property for "DataSource" in "toDisplayNames.properties" should appear as: DataSource = DATA_SOURCE.

Example A.1. toDisplayNames.properties file

```
BMClosedED = BM__CLOSED_ED_COMPONENT
BMDevice = BM__DEVICE_COMPONENT
BMDisplay = BM__DISPLAY_COMPONENT
GPS = GPS
AirFrame = AIRFRAME
NavDisplay = NAV_DISPLAY
EventChannel_EventChannel = EventChannel
GPS_BMDevice = BMDevice
AirFrame_BMClosedED = BMClosedED
NavDisplay_BMDisplay = BMDisplay
```

Secondly, the "toDisplayNames.properties" file provides some implementation information for component instances. In the current CCM translation of OEP scenarios, one component in OEP may correspond to multiple components defined in the IDL3 file because of a different number of ports. We combine the component instance name and component type name from the scenario file as the key, and we set the corresponding component name in OEP as the value. For example, there may be an instance named "GPS" in the scenario file which implements component type "Device". If "GPS" implements component type "BMDevice" in the OEP XML file, then the property for "GPS" should be written as: GPS_Device = BMDevice.

After a "toDisplayNames.properties" file has been added to the project in the "BasicSP\system\src" directory, an OEP XML file may be generated by clicking the "Generate OEP XML" button (indicated in the above example). The OEP XML will be generated within the "BasicSP\generated" directory. For the newly generated XML to show up in the Navigator view, it may be necessary to refresh the project.

Reverse Engineering OEP

With Cadena, it is possible to automatically generate idl3, cps, and scenario file templates based on OEP XML configuration files.

1. In the Navigator View, expand all folders.
2. Select the "system" folder.
3. Click the right mouse button, select "Import..."
4. Select "File system" as the import source.
5. Select "Next".
6. For the directory, navigate to the directory that contains the OEP XML configuration files (this can be found within the OEP distribution).
7. Select the configuration file(s) you are intersted in.
8. Select "Finish" to import the selected file(s).
9. Open the configuration file from which you wish to generate the templates.
10. Comment out the <!DOCTYPE ... > line. (i.e. <!--DOCTYPE ... -->)
11. Select Cadena->Core->Translate OEP to CCM.
12. Select the project in the navigator view, right click and select Refresh

If you succesfully completed the above steps, the generated idl3, cps,and scenario files should now be present within their respective system directories.