▸ ▸ ▸ Part 1: Warm-Up Exercises

This collection of warm-up exercises is designed to complement the modules in the *Mastering Rational Rose RealTime* student manual. Each of these brief exercises focuses on a specific modeling skill or technique that lets you directly apply what you learn during the course lectures.

Before you begin each exercise, your instructor will demonstrate and explain each step in the instructions. When it’s your turn to create the model, you may find that you are able to put the instructions aside and rely completely on what you learned from the lecture and demonstration. You are encouraged to do so. You will learn much more by relying on what you have learned from the instructor than by simply following these instructions mechanically.

Complete instructions for the exercises are included here in case you get stuck during class or wish to refer back to these exercises after the course is over.

Overview of Exercises

Here is an outline of all the warm-up exercises in this collection:

Warm-Up 1: Hello, World

Create a capsule with behavior, but no structure.

Warm-Up 2: Passive Classes

Create a passive class with simple behavior defined in an operation.

Warm-Up 3: Traffic Light

Create a capsule with one port, whose state machine contains three states.

Warm-Up 4: Electronic Lock

Create a model with a passive class whose behavior is defined by a state diagram.

Warm-Up 5: Battleship

Create a model with simple structure and behavior, which uses timers and the log service.

Warm-Up 6: Traffic System

Create a model with moderate behavior and structure.

Warm-Up 7: Client/Server

Create two client / server models with dynamic structure and behavior.

Warm-Up 8: RQA-RT

Create a specification sequence diagram for the Client / Server model and run the Verify Behavior against it.

Table of Figures

Figure 1.1 - Create New Model Dialog 5
Figure 1.2 - Create New Capsule From Browser 6
Figure 1.3 - State Diagram Toolbar and Window 7
Figure 1.4 - Transition Specification 7
Figure 1.5 - Setting the Top Capsule 8
Figure 2.1 - Creating a New Operation In the Browser 12
Figure 2.2 - Code for Greetings main Operation 13
Figure 2.3 - Adding a Library to a Class 14
Figure 2.4 - Component Specification - General Tab 15
Figure 2.5 - Component Specification C++ Compilation Tab 16
Figure 2.6 - Shutdown Model Execution 17
Figure 2.7 - Changing the Greetings Code 18
Figure 2.8 - Running nmake From the Command Prompt 18
Figure 2.9 - Synchronizing the Code 19
Figure 3.1 - Component Specification Dialog 21
Figure 3.2 - Inserting Signals In a Protocol 22
Figure 3.3 - Selecting a Protocol To a Port 23
Figure 3.4 - Conjugating a Port 24
Figure 4.1 - Adding the Standard C++ Libraries 28
Figure 4.2 - main Operation Specification Detail Tab 29
Figure 4.3 - State Diagram and Tool Bar 30
Figure 4.4 - Transitions and States 32
Figure 4.5 - Transitions Specification & Event Editor 33
Figure 4.6 - Lock State Diagram Showing Transitions 35
Figure 4.7 - Finished Model 36
Figure 5.1 - Battleship Sequence Diagram 40
Figure 6.1 - Traffic System Sequence Diagram 42
Figure 8.1 - Client/Server Collaboration Diagram 46
Figure 8.2 - Client/Server Sequence Diagram - One Client 47
Figure 8.3 - Client/Server Sequence Diagram with Data 49
Figure 8.4 - RQA-RT Verify Behavior Dialog 50
Figure 8.5 - Client/Server Sequence Diagram with Local Actions 52

Warm-Up 1: Hello, World

This exercise accompanies Module 2, “Rose RealTime Models.”

You will create a model containing one *capsule,* whose behavior is to print “Hello, World!” on the screen.

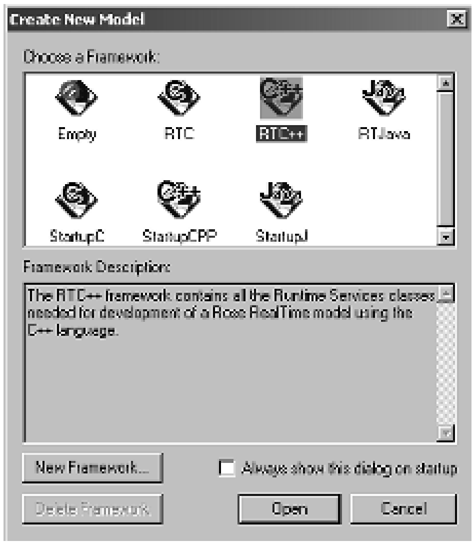
Objectives

After successfully completing this exercise, you will be able to:

- Create a new model.
- Create a capsule in the model.
- Add simple behavior to the capsule.
- Compile, run, and debug the model.

Instructions

1. Start the Rose RealTime toolset.
2. Create a new C++model.
   - 2.1. Select **File** > **New.**
   - 2.2. Double-click **RTC++** framework.
3. **Save the model.**
   - 3.1. Select **File > Save Model As**.
   - 3.2. Browse to C:\MRRT\StudentWork.
   - 3.3. Name the model *HelloWorld.*
   - 3.4. Press ↵(Enter) or click Save
   - 3.5. You will be asked if you want to create a workspace. When a workspace is created, three files are created to save configuration and user options. Click Yes



The RTC++ framework contains all the Runtime Services classes needed for development of a Rose RealTime model using the C++ language.

Figure 1.1 - Create New Model Dialog

4. Create a capsule:

   4.1. Right-click the Logical View package.

   4.2. Select **New** > **Capsule.**

   4.3. Name the capsule.

      4.3.1. Type `HelloWorld` ↵. Do not use blank spaces in the name of any Rose RealTime element.

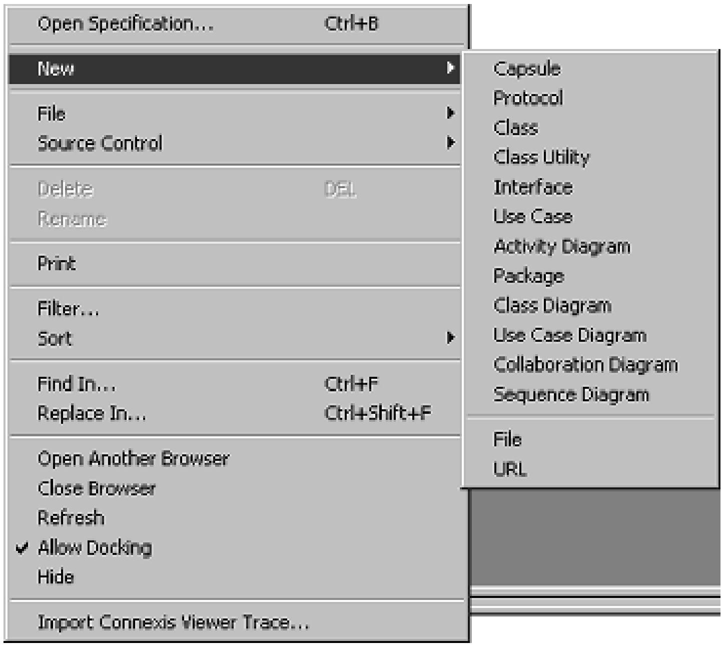


Figure 1.2 - Create New Capsule From Browser

   4.4. Open the capsule's *State Diagram* editor window.

      4.4.1. Right-click the capsule.

      4.4.2. Select **Open State Diagram**.

   4.5. Add a state to the capsule's state diagram.

      4.5.1. Click the State Tool on the toolbar to select it.

      4.5.2. Click inside the State Diagram.

      4.5.3. To name the state, type `Active` ↵.

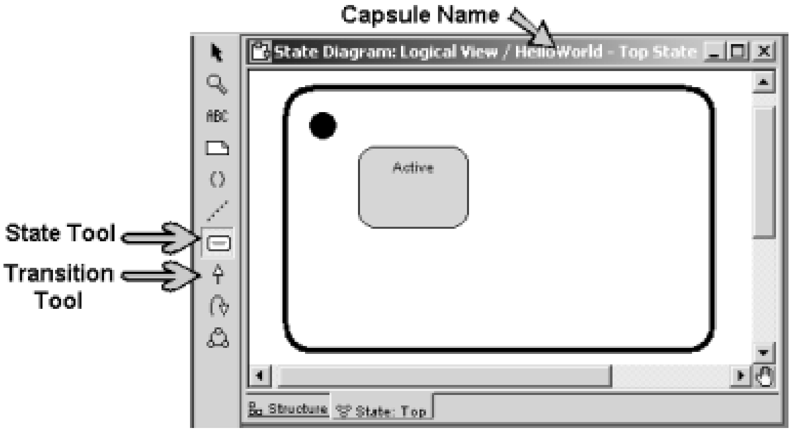


Figure 1.3 - State Diagram Toolbar and Window

4.6. Add an initial transition to the state.

4.6.1. Click the State Transition tool.

4.6.2. Click the initial state dot and drag to *Active*.

4.7. Place code in the transition:

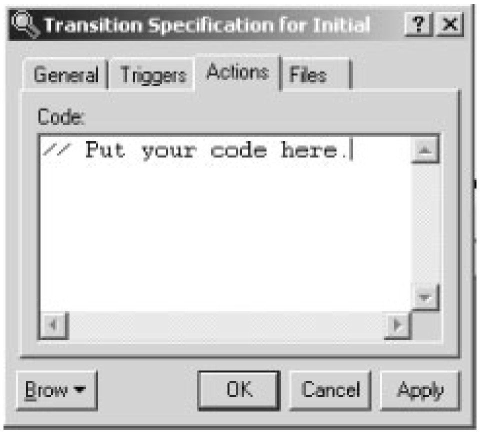
4.7.1. Double-click the initial transition line to open the specification dialog box.

4.7.2. Select the `Actions` tab.

4.7.3. Type the following code:

```
cout << “Hello, World” << endl;
```

4.7.4. Click OK.



```
// Put your code here.
```

Figure 1.4 - Transition Specification

5. Include a library.

   5.1. Double-click the capsule in the Logical View to open the capsule’s Specification dialog box.

   5.2. Click the **C++** tab.

   5.3. In the **HeaderPreface** area type the statement:

   ```
   #include <iostream.h>
   ```

   5.4. Click OK

6. Create a new *component*.

   6.1. Right-click the Component View folder.

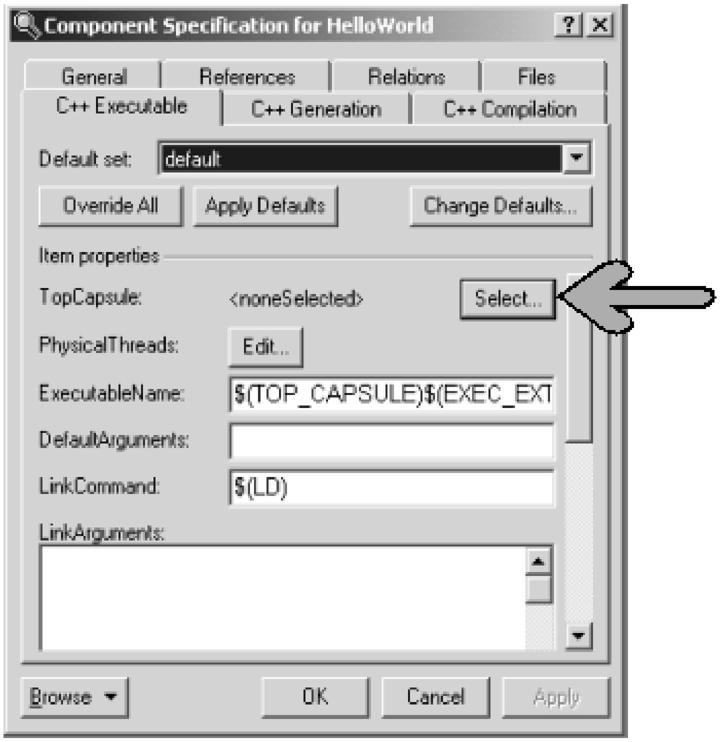
   6.2. Select **New** > **Component.**

   6.3. Name the new component *HelloWorld.*

   6.4. Set the component to *active*. To do this, right-click the component and select **Set As Active**. (Notice that this makes the name of the component in the browser bold.)

   6.5. Drag the *HelloWorld* capsule (from the *Logical View*) onto the component.

   6.6. Double-click the component to open the Component Specification dialog box.



Default set: default

TopCapsule: <noneSelected>

ExecutableName: $(TOP_CAPSULE)$(EXEC_EXT

LinkCommand: $(LD)

Figure 1.5 - Setting the Top Capsule

6.7. Set the *TopCapsule* to *HelloWorld*.

6.7.1. Go to the `C++ Executable` tab of the Component Specification dialog.

6.7.2. Click Select....

6.7.3. Choose HelloWorld from the list.

6.7.4. Click OK.

6.8. Set the *TargetConfiguration* to *NT40T.x86-VisualC++-6.0*

6.8.1. Go to the `C++ Compilation` tab.

6.8.2. Click Select....

6.8.3. Select NT40T.x86-VisualC++6.0 from the list.

6.8.4. Click OK.

6.9. Click OK to save the settings.

7. Create a new processor.

7.1. Right-click the Deployment View folder.

7.2. Select **New** > **Processor**.

7.3. Name the processor *Workstation*.

8. Compile, run, and debug *HelloWorld*.

8.1. Drag the *HelloWorld* component (from the *Component View*) onto *Workstation* (in the *Deployment View*). This creates a *component instance*. Rename it *HelloWorld*.

8.2. Set the component instance to *active* by right-clicking the instance and selecting **Set As Active.** (Notice that this makes the name of the component instance in the browser bold.)

8.3. Save your model.

8.4. Click the **Run** button ⚡. A dialog box appears. Click Yes to build and execute the model.

8.4.1. Watch the Build Log in the lower-left of the Rose RealTime window to see each step in the build as it is done.

8.4.2. When the build is successfully completed, a Command Prompt window appears in front of the Rose RealTime window. Click the Rose RealTime window to bring it back into context.

8.4.3. The Browser now shows the RTS view instead of the Model view.

To start the model, click the **Start** button. ▶ Return to the Command Prompt Window to see the results.

8.4.4. End the run-time session by clicking the **Shutdown** button.⊗

Warm-Up 2: Passive Classes

This exercise accompanies Module 3, “Passive Classes”. It is made up of a single passive class with an operation that contains the code to print the greeting.

When the model is complete and running correctly, you will change the greeting that is printed outside of the Rose RealTime toolset. You will then synchronize the new code with the Rose RealTime model.

Objectives

After successfully completing this exercise, you will be able to:

- Create a model using a passive class.
- Add simple behavior to the class by adding an operation.
- Compile, run, and debug the model.
- Change the code produced by Rose RealTime with a text utility.
- Resynch the model with the changed code.

Instructions

1. Create a new C++ model.

1.1. Select **File** > **New.**

1.2. Double-click **RTC++** framework.

2. Save the model with the name *PassiveClass***.**

2.1. Select **File > Save Model As**.

2.2. Browse to C:\MRRT\StudentWork.

2.3. Type: `PassiveClass`↵.

2.4. You will be asked if you want to create a workspace. Click Yes. This creates three files that contain configuration and user options.

3. Create a passive class:

3.1. Right-click the Logical View package.

3.2. Select **New** > **Class.**

3.3. Name the class *Greetings*.

3.3.1. Type: `Greetings` ↵

3.4. Click OK

4. Create an operation and add code to provide behavior.

4.1. Right-click the class in the browser.

4.2. Select **New > Operation** from the context menu.

4.2.1. Name the operation *main*.

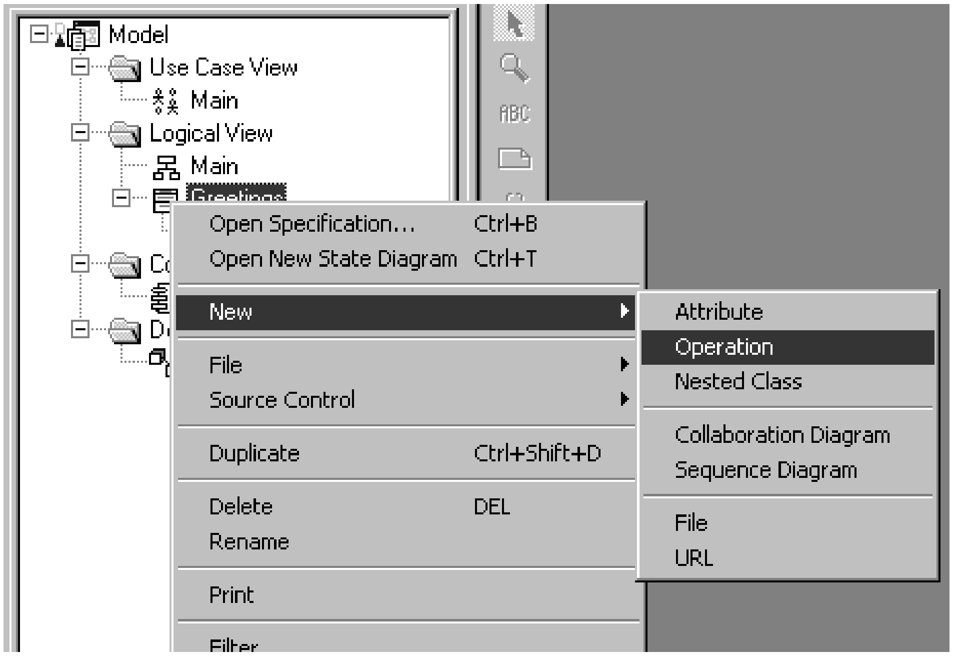


Figure 2.1 - Creating a New Operation In the Browser

4.3. Double-click the operation to open the Operation Specification.

4.4. Choose the Detail tab.

4.5. Set the **Return type** to `int`.

4.6. Create a parameter named *argc* .

4.6.1. Right-click in the Parameters box and select **Insert**.

4.6.2. Type: argc ↵

4.6.3. Tab to the area to the right of the operation parameter name and below the Type heading. Click the highlighted box to open the drop-down list.

4.6.4. Select *int* from the list.

4.7. Create a parameter named *argv*. (See Figure 2.3.)

4.7.1. Right-click in the Parameters box and select **Insert**.

4.7.2. Type: argv[ ] ↵

4.7.3. Double-click the parameter name to display the Parameter Specification.

4.7.4. In the **Type** box, type: const char* const* ↵

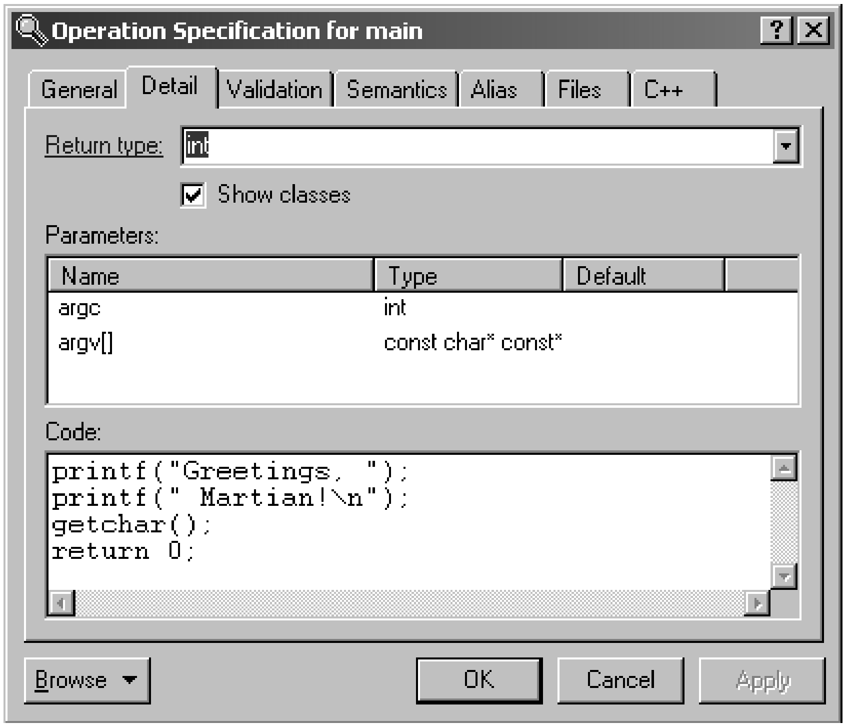
4.8. Click in the **Code** box and type the following code:

```
printf(“ Greetings,  ”);
printf(“ Earthling \n”);
getchar();
return 0;
```

4.9. Select the C++ tab

4.10. Set **OperationKind** to *global*.

4.11. Click OK.



Return type: int

☑ Show classes

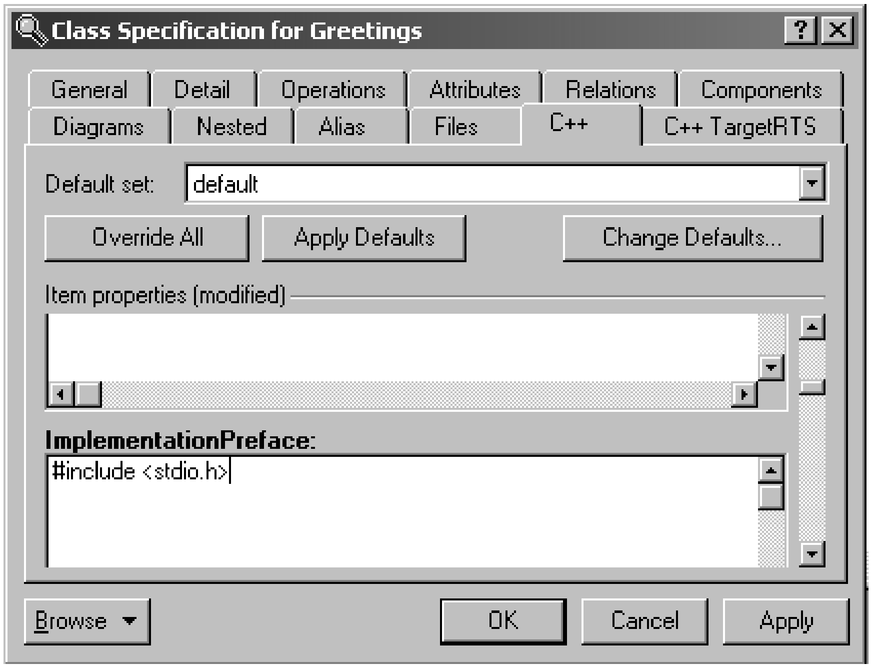
Parameters:

Name	Type	Default
argc	int	
argv[]	const char* const*	

Code:

```
printf("Greetings, ");
printf(" Martian!\n");
getchar();
return 0;
```

Figure 2.2 - Code for Greetings *main* Operation



Default set: default

ImplementationPreface:

```
#include <stdio.h>
```

Figure 2.3 - Adding a Library to a Class

5. **Include the I/O library.**

   5.1. Double-click the class in the Logical View to open the class' Specification.

   5.2. Click the C++ tab.

   5.3. Place the following statement in the **ImplementationPreface** area.

   ```
   #include <stdio.h>
   ```

   Note: You may have to scroll down to find this area of the specification dialog.

   5.4. Click OK.

6. **Create a new component.**

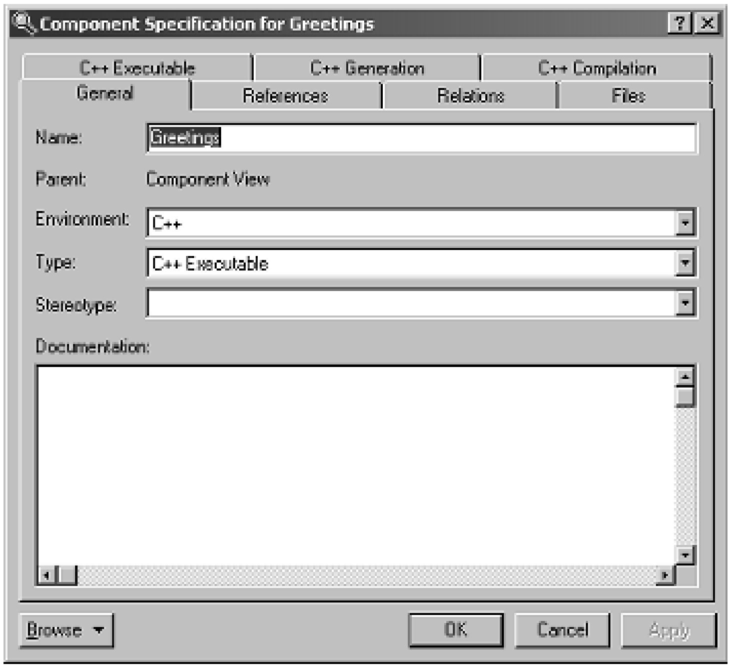
   6.1. Right-click the Component View folder.

   6.2. Select **New** > **Component.**

   6.3. Name the new component *Greetings.*

   6.4. Create a reference for *Greetings* class by dragging the class from the browser *Logical View* onto the *Greetings* component in the *Component View*. (Note, there is no visual cue that the reference has been made. If you open the Component Specification dialog box, the class will be listed on the References tab.)

   6.5. Set the component to active. To do this, right-click the component and select **Set As Active**.



Name: Greetings

Parent: Component View

Environment: C++

Type: C++ Executable

Stereotype:

Documentation:

Figure 2.4 - Component Specification - General Tab

6.6. Double-click the component to open the Component Specification dialog box.

6.7. Select the `General` tab.

6.8. Set the following parameters:

6.8.1. Set **Environment** to *C++* .

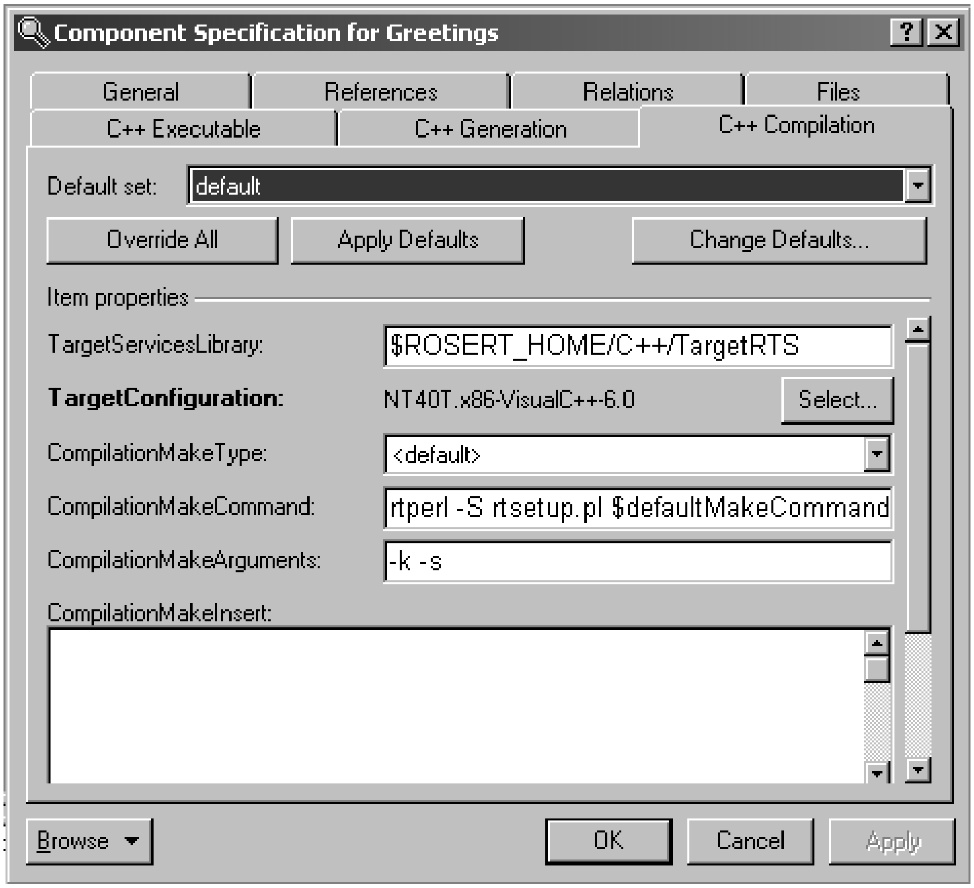
6.9. Set the TargetConfiguration parameter for the component.

6.9.1. On the `C++ Compilation` tab, click Select...

6.9.2. Select *NT40T.x86-VisualC++-6.0* from the list.

6.9.3. Click OK to accept the target configuration.

6.9.4. Click OK to save the settings.



Default set: default

TargetServicesLibrary: $ROSERT_HOME/C++/TargetRTS

TargetConfiguration: NT40T.x86-VisualC++-6.0

CompilationMakeType: <default>

CompilationMakeCommand: rtperl -S rtsetup.pl $defaultMakeCommand

CompilationMakeArguments: -k -s

CompilationMakeInsert:

Figure 2.5 - Component Specification C++ Compilation Tab

7. **Create a new processor.**

   7.1. Right-click the Deployment View folder.

   7.2. Select **New** > **Processor**.

   7.3. Name the processor *Workstation*.

8. **Save your model.**

9. **Create a component instance.**

   9.1. Drag the *Greetings* component from the Component View onto *Workstation* in the Deployment View*)*. This creates a component instance. Rename it *Greetings*.

   9.2. Double-click the component instance to display the Component Instance Specification dialog. On the `Detail` tab, clear the box labeled **Attach to target on startup**.

   9.2.1. Click OK

10. Compile, run, and debug the model.

10.1. Set the component instance to *active* by right-clicking the instance and selecting **Set As Active.**

10.2. Click the **Run** button ⚡. A dialog box appears. Click Yes to build and execute the model.

10.2.1. A Command Prompt window will appear to show the results.

10.2.2. To end execution, close the Command Prompt window, right-click on the component instance, and select Shutdown. (Ignore the error. Click OK to close the error dialog.) The reason for this step is that the executable has already quit, but the scripts in Rose RealTime have not.

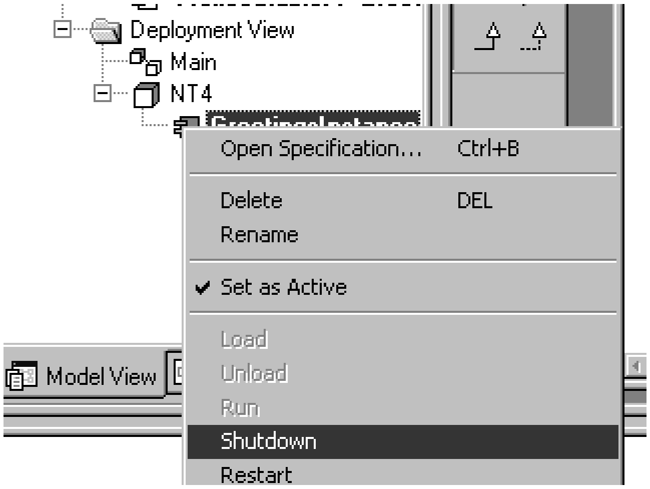
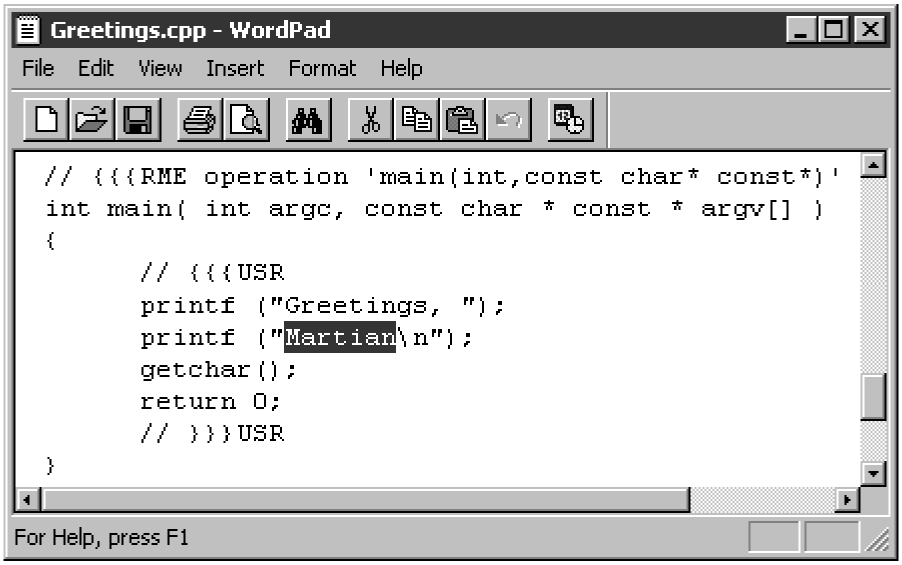


Figure 2.6 - Shutdown Model Execution

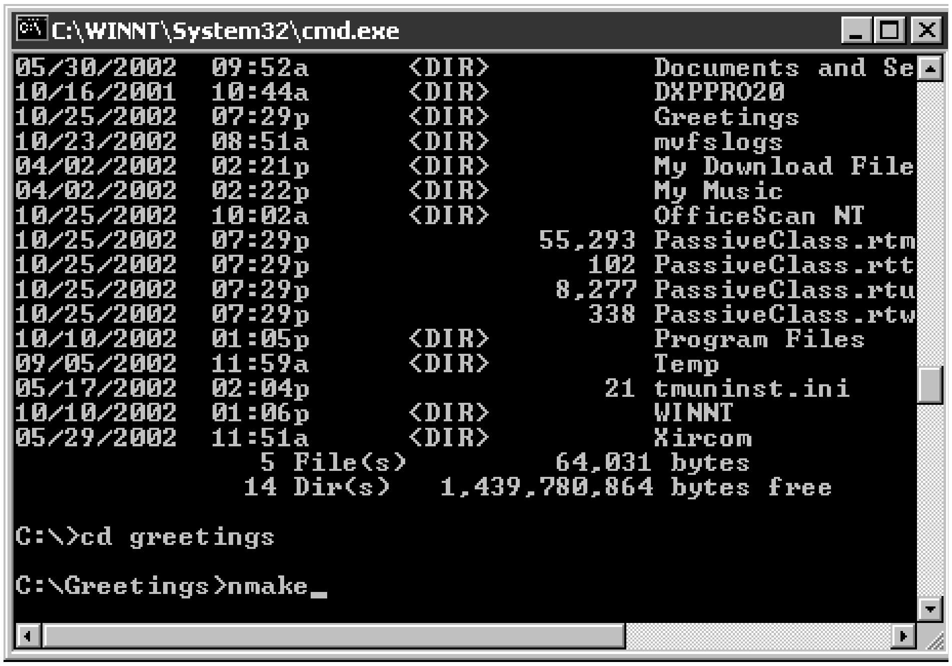


```
// {{{RME operation 'main(int,const char* const*)'
int main( int argc, const char * const * argv[] )
{
      // {{{USR
      printf ("Greetings, ");
      printf ("Martian\n");
      getchar();
      return 0;
      // }}}USR
}
```

Figure 2.7 - Changing the Greetings Code

10.3. Use the editor of your choice to open the corresponding source file and edit the printed message to your favorite greeting. (The file you want should be C:\MRRT\Greetings\src\Greetings.cpp.) Save the changes.

10.4. Open a Command Prompt window and navigate to the directory at the top of the model, (C:\MRRT\Greetings), and type **nmake**.



```
05/30/2002  09:52a      <DIR>          Documents and Se
10/16/2001  10:44a      <DIR>          DXPPRO20
10/25/2002  07:29p      <DIR>          Greetings
10/23/2002  08:51a      <DIR>          mvfslogs
04/02/2002  02:21p      <DIR>          My Download File
04/02/2002  02:22p      <DIR>          My Music
10/25/2002  10:02a      <DIR>          OfficeScan NT
10/25/2002  07:29p              55,293 PassiveClass.rtm
10/25/2002  07:29p                 102 PassiveClass.rtt
10/25/2002  07:29p               8,277 PassiveClass.rtu
10/25/2002  07:29p                 338 PassiveClass.rtw
10/10/2002  01:05p      <DIR>          Program Files
09/05/2002  11:59a      <DIR>          Temp
05/17/2002  02:04p                  21 tmuninst.ini
10/10/2002  01:06p      <DIR>          WINNT
05/29/2002  11:51a      <DIR>          Xircom
               5 File(s)         64,031 bytes
              14 Dir(s)   1,439,780,864 bytes free

C:\>cd greetings

C:\Greetings>nmake_
```

Figure 2.8 - Running nmake From the Command Prompt

10.5. Go to the *bld* directory and run the model. The greeting you see when you run will show you the changes you made. Shutdown the execution of the model.

10.6. Encorporate the changed code into your model:

10.6.1. Right-click the *Greetings* component in the Component View.

10.6.2. Select **Build > CodeSync.**

10.6.3. Click OK.

10.6.4. Build the component again.

10.7. Return to the Command Prompt window and run the model to verify that the changes were correctly synchronized.

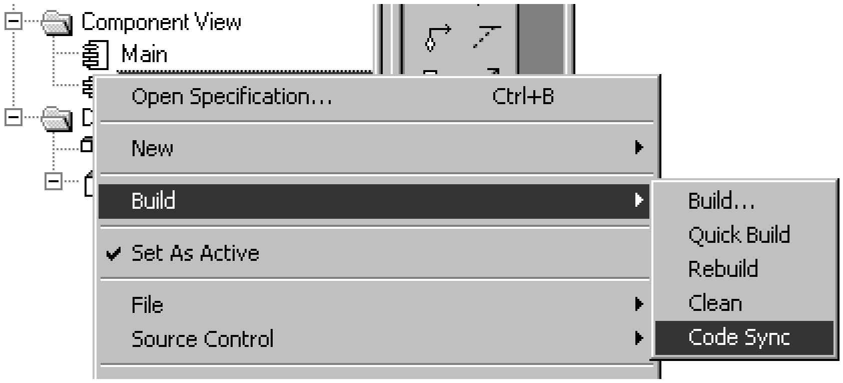


Figure 2.9 - Synchronizing the Code

Warm-Up 3: Traffic Light

This exercise accompanies Module 5, “State Modeling”.

In this exercise, you will create a simple traffic light application composed of a capsule with one port, whose state machine contains three states. The requirements for this model are specified in a sequence diagram.

Objectives

After successfully completing this exercise, you will be able to:

- Build a model from requirements given in the form of a sequence diagram.
- Create a simple protocol class.
- Define behavior (finite state machine), composed of multiple states and transitions.
- Compile, run, debug, and verify the multi-state model, using sequence diagrams generated during run time.

Instructions

1. Use the following sequence diagram as a set of requirements.

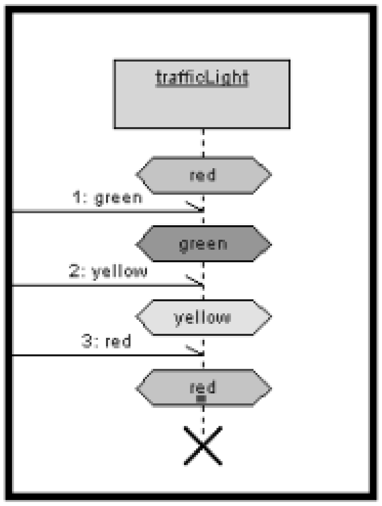


Figure 3.1 - Component Specification Dialog

2. Create a model named *TrafficLight*. Save it in StudentWork.

3. Create a protocol class named *LightControl*. To do this:

   3.1. Right-click the Logical View package.

   3.2. Select **New** > **Protocol**.

   3.3. Name the protocol by typing in the name followed by ↵.

   3.4. Add signals named *red*, *yellow*, and *green*. They must all be In Signals or Out Signals. If you mix the directions, the model won't work. The direction of the signals and the conjugation of the ports are interdependent. Because we will use this model later, all signals for this model must be Out Signals.

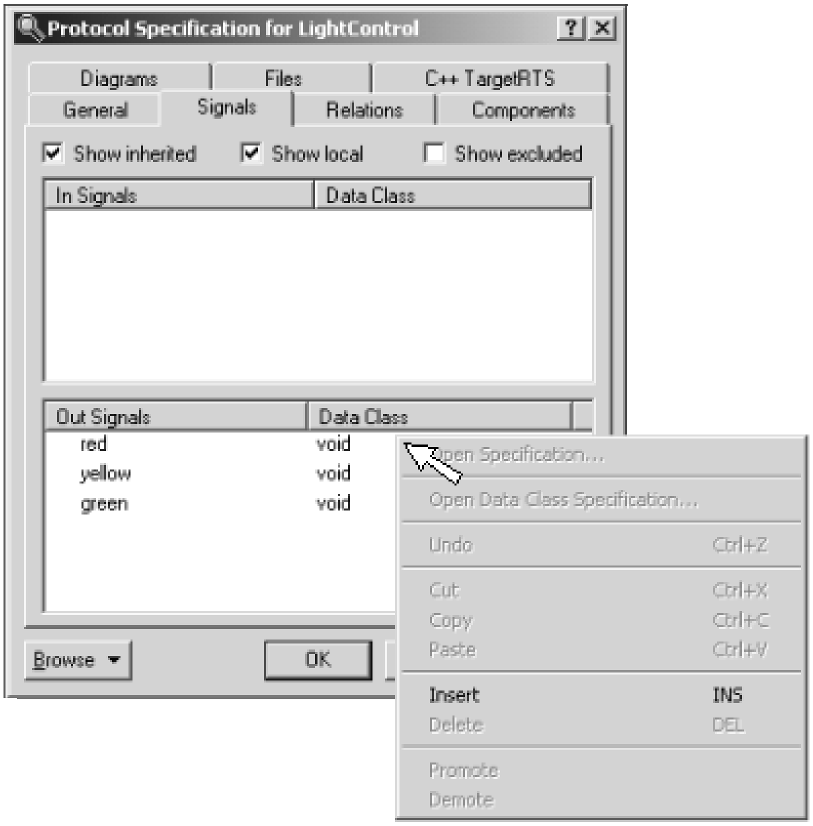
      3.4.1. Open the protocol specification dialog box by double-clicking the protocol in the Model Browser.

      3.4.2. Click the Signals tab.

      3.4.3. Right-click in the **Out Signals** area and select **Insert**.

      3.4.4. Type the name of the signal ↵

   3.5. Click OK to save your settings.


Protocol Specification for LightControl

☑ Show inherited ☑ Show local ☐ Show excluded

In Signals	Data Class

Out Signals	Data Class
red	void
yellow	void
green	void

Figure 3.2 - Inserting Signals In a Protocol

4. Create a capsule class named *TrafficLight*.

5. Define the structure of *TrafficLight*:

   5.1. Open the structure diagram for *TrafficLight* by expanding its tree and double-clicking on **Structure Diagram**.

   5.2. Add an end port, based on *LightControl*, named *control*.

   5.2.1. Select the Port Tool ▣ . Click the boundary line of the capsule structure. Select *LightControl* from the list by double-clicking it.

   5.2.2. Type the name of the port followed by ↵.

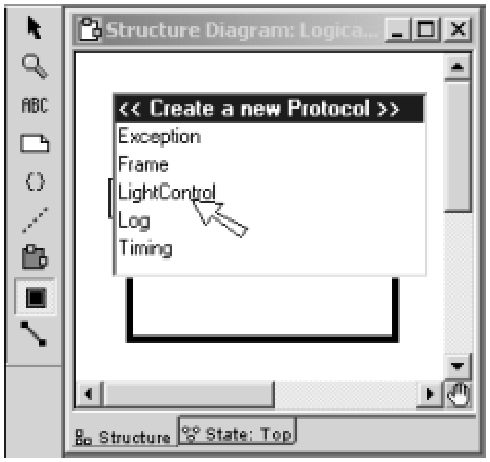


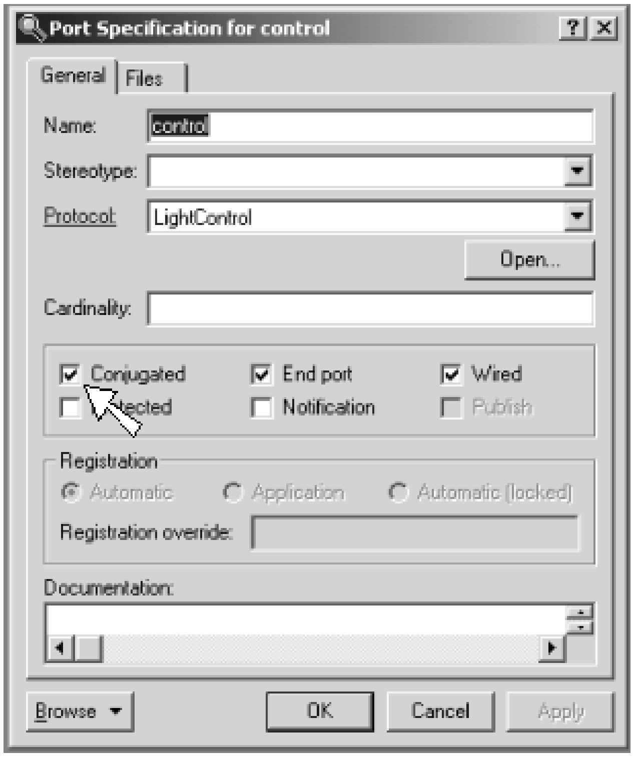
Figure 3.3 - Selecting a Protocol To a Port

   5.3. Determine the proper conjugation for the port, and set it. To help you decide if you want a base or conjugatred port, think about the fact that this traffic light will be one of four in a typical four-way intersection. If you select the wrong conjugation, you will not see the correct signals when you associate a trigger with each of the transitions you create in step 5.4.

   5.3.1. Open the Port Specification dialog box by double-clicking the port on the diagram.

   5.3.2. Make sure that the End Port box is selected. Select or clear the Conjugated box based on what type is needed.

   5.3.3. Click OK .



Name: control

Stereotype:

Protocol: LightControl

Cardinality:

☑ Conjugated ☑ End port ☑ Wired

☐ Protected ☐ Notification ☐ Publish

Registration

● Automatic ○ Application ○ Automatic (locked)

Registration override:

Documentation:

Figure 3.4 - Conjugating a Port

6. Define the behavior of *TrafficLight*:

6.1. Add three states named *Red*, *Yellow*, and *Green*.

6.1.1. Open the capsule's state diagram by double-clicking on **State Diagram** in the capsule's tree.

6.1.2. Using the State Tool , create three states and name them as directed above. You can arrange them to emulate a traffic signal.

6.2. Make *Red* the initial state.

6.2.1. Using the Transition Tool , create a transition to *Red* from the initial point by clicking on the initial point and dragging to *Red.*

6.3. Add three transitions.

6.3.1. Create transition *goGreen* from *Red* to *Green.*

6.3.2. Create transition *goYellow* from *Green* to *Yellow*.

6.3.3. Create transition *goRed* from *Yellow* to *Red.*

6.4. Add triggers for each transition.

6.4.1. Double-click on the transition on the diagram to display the Transition Specification dialog box.

6.4.2. Select the Triggers tab.

6.4.3. Right-click in the *Triggers* area and select **Insert**.

6.4.4. Select *control* from the Port list. This launches the Event Editor.

6.4.5. In the Signal list, check the box for the correct signal color.

6.4.6. Click OK. to close the Event Editor dialog.

6.4.7. Repeat for each of the remaining two change-light-color transitions.

6.4.8. Click OK. to save the triggers you created and close the Transition Specification dialog.

7. Build and run *TrafficLight* in the same way you did *HelloWorld*.

8. Test and debug *TrafficLight*.

8.1. Switch context from the Command Prompt window to the RunTime System (RTS).

8.2. Add a probe to the *control* port.

8.2.1. Right-click *TrafficLight* and select **Open Structure Monitor**.

8.2.2. Select the Probe Tool ⊞. Click the port in the structure monitor.

8.3. Create three signals: *red*, *yellow*, and *green*.

8.3.1. Expand the *Probes* folder and double-click the probe to open its specification dialog box.

8.3.2. Click the Detail tab, right-click the entry field, and select **Insert**.

8.3.3. Use the **Edit Inject Message** specification to name the signal. Set **Priority** to *General* and **Direction** to *In*.

8.3.4. Click OK. to save the signal.

8.3.5. Repeat for the remaining signals.

8.3.6. Click OK. to close the Probe Specification.

8.4. Inject *red*, *yellow*, and *green* signals to cycle the behavior.

8.4.1. Click **Start** ▶ to run the model.

8.4.2. Right-click *TrafficLight* and select **Open State Monitor**.

8.4.3. Double-click *control/green* in the probe folder. This injects the *green* signal and causes the model to transition to the *Green* state.

8.4.4. Double-click *control/yellow* in the probe folder. This injects the *yellow* signal and causes the model to transition to the *Yellow* state.

8.4.5. Double-click *control/red* in the probe folder. This injects the *red* signal and causes the model to transition to the *Red* state.

9. Click the **Shutdown** button ⊗ to halt the execution of the model.

10. Save the model.

▶ ▶ ▶ Warm-Up 4: Electronic Lock

This exercise accompanies Module 5, “State Modeling”. It models the behavior of an electronic lock The model starts with the lock in the locked state. To unlock the lock, you must enter 1 and then 2. The lock is locked by entering the letter L. For help, enter a question mark.

The model is composed of a passive class with a state machine that provides the behavior. The state machine has three states: *Locked*, *Unlocking*, and *Unlocked*. The *Unlocking* states waits for the second character of the combination, changing to the *Locked* or *Unlocked* state depending on the character received.

Objectives

After successfully completing this exercise, you will be able to:

- Add a state machine to a passive class.
- Create a trigger operation for a passive class state machine.
- Add code to a transition.

Instructions

1. **Start a new model named** *Lock***.**

2. **Save the model**

3. **Create a passive class named** *ElectronicLock*:

   3.1. Right-click the Logical View package.

   3.2. Select **New** > **Class.**

   3.3. Name the class.

   3.3.1. Type: ElectronicLock ↵.

4. **Include the I/O libraries.**

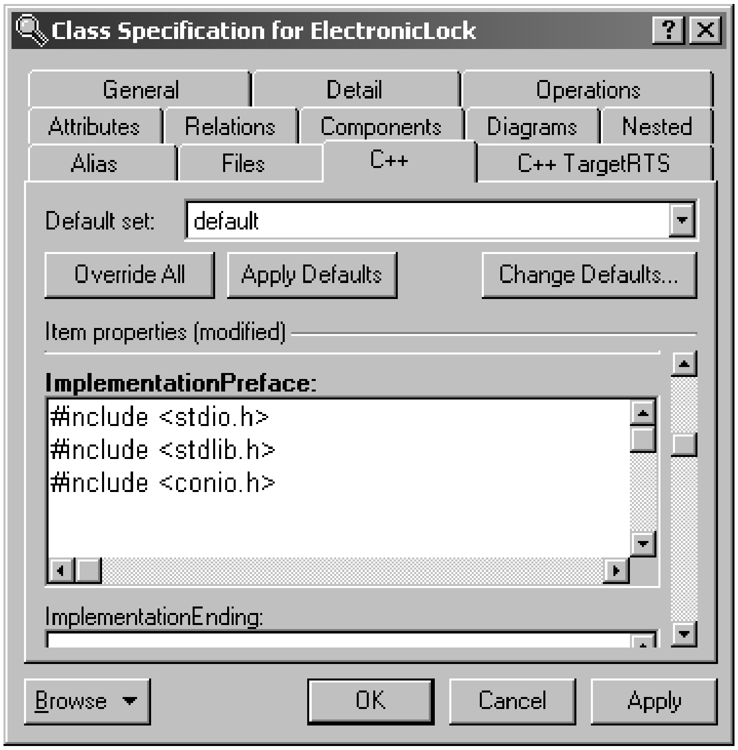
   4.1. Double-click on the class icon to open the Class Specification dialog.

   4.2. Go to the <u>C++</u> tab of the Class Specification.

   4.3. Place the following statements in the **ImplementationPreface** area.

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <conio.h>
   ```

   4.4. Click OK.



Default set: default

ImplementationPreface:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

ImplementationEnding:

Figure 4.1 - Adding the Standard C++ Libraries

5. **Create a main operation.**

   5.1. Right-click the class in the browser.

   5.2. Select **New** > **Operation** from the context menu.

   5.3. Name the operation *main*.

   5.4. Double-click the operation to open the Operation Specification dialog box.

   5.5. Select the <u>C++</u> tab.

   5.6. Set **OperationKind** to *global*.

   5.7. Choose the <u>`Detail`</u> tab.

   5.8. Set the **Return type** to *int*.

5.9. Create a parameter named *argc*.

5.9.1. Right-click in the Parameters box.

5.9.2. Select Insert.

5.9.3. Type: argc ↵

5.9.4. Click the area to the right of the parameter name and below the Type heading. Move your cursor slightly and click again.

5.9.5. Select *int* from the list.

5.10. Create a parameter named *argv*.

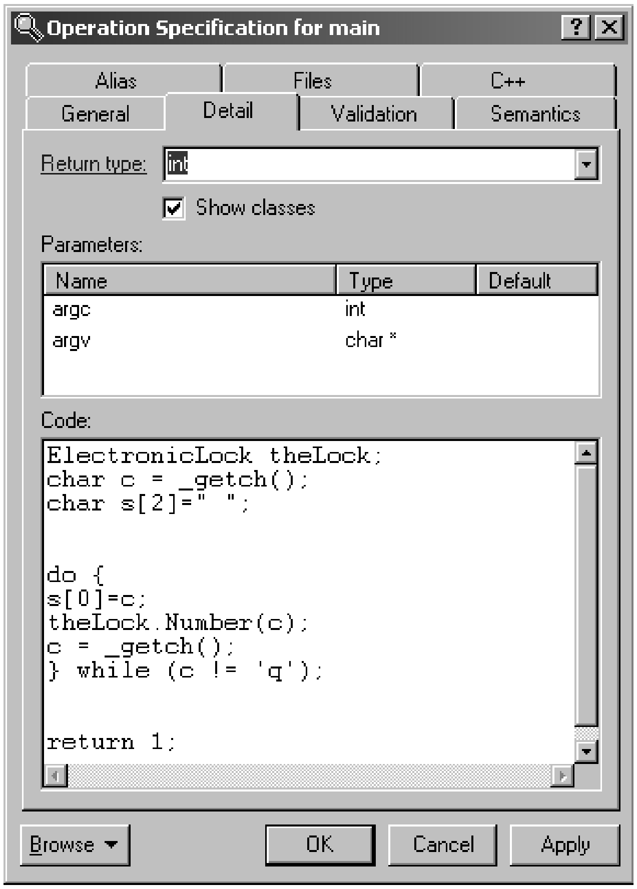
5.10.1. Right-click in the Parameters box.

5.10.2. Select Insert.

5.10.3. Type: argv [ ] ↵

5.10.4. Double-click the parameter name to display the Parameter Specification dialog box and return to the Operation specification dialog.

5.10.5. In the **Type** box, type: char* ↵



Return type: int

☑ Show classes

Parameters:

Name	Type	Default
argc	int	
argv	char *	

Code:

```
ElectronicLock theLock;
char c = _getch();
char s[2]=" ";


do {
s[0]=c;
theLock.Number(c);
c = _getch();
} while (c != 'q');


return 1;
```

Figure 4.2 - *main* **Operation Specification Detail Tab**

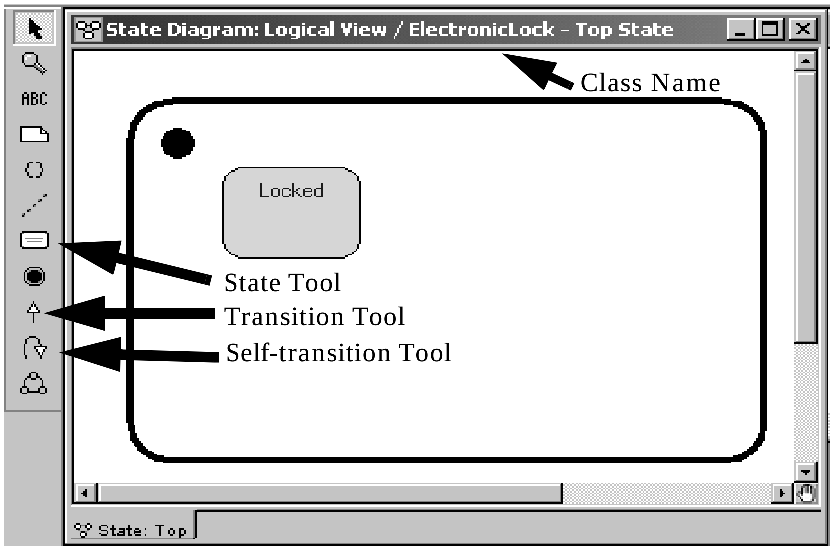


Figure 4.3 - State Diagram and Tool Bar

5.11. Click in the **Code** box and type the following code:

```
ElectronicLock theLock;
char c;
do {
theLock.Number(c);
c = _getch();
} while (c != 'q');
return 1;
```

5.12. Click OK.

6. **Create an operation to act as a trigger.**

6.1. Right-click the class in the browser.

6.2. Select **New > Operation** from the context menu.

6.3. Name the operation *Number*.

6.4. Double-click the operation to open the Operation Specification dialog box.

6.5. Choose the General tab.

6.6. Type: trigger in the Stereotype box.

6.7. Select the C++ tab.

6.8. Set **OperationKind** to *global*.

6.9. Select the Detail tab.

6.10. Set the **Return type** to *void*.

6.11. Create a parameter.

6.11.1. Name the parameter *num*.

6.11.2. Set the parameter type as *char*.

6.12. Click OK.

7. Create the state machine.

7.1. Right-click the *ElectronicLock* class in the browser.

7.2. Select **Open New State Diagram**.

7.3. Place a state on the diagram.

7.3.1. Click the State Tool on the toolbar to select it.

7.3.2. Click inside the state diagram.

7.3.3. To name the state, type: Locked↵.

7.4. Create an initial transition.

7.4.1. Click the State Transition tool .

7.4.2. Click-and-drag from the initial state dot to *Locked*.

7.5. Place code in the transition:

7.5.1. Double-click the initial transition line to open the specification dialog box.

7.5.2. Select the Actions tab.

7.5.3. Type the following code:

```
printf("*****************************\n");
printf(" ACME Electronic Lock System \n");
printf("*****************************\n\n");
```

7.5.4. Click OK.

7.6. Create a self-transition on the Locked state.

7.6.1. Click the Self-transition tool .

7.6.2. Click inside the *Locked* state.

7.6.3. Name the transition *question*.

7.7. Define the *question* self-transition on the *Locked* state. This transition is taken if a question mark is entered. Normally, instructions on what to do would be placed here. For now, just print a message of acknowledgement.

7.7.1. Double-click the *question* transition on the *Locked* state to open the Transition Specification dialog.

7.7.2. Go to the Triggers tab.

7.7.3. Right-click and select **Insert**.

7.7.4. Select *Number* from the drop-down box.

7.7.5. In the Guard window, type: `num == '?'`

7.7.6. Click OK.

7.7.7. Go to the Actions tab on the Transition Specification dialog and enter the following in the Code window:

```
printf ("Locked Help Goes Here\n");
```

7.7.8. Click OK.

7.8. Add additional states and transitions to match the following diagram.

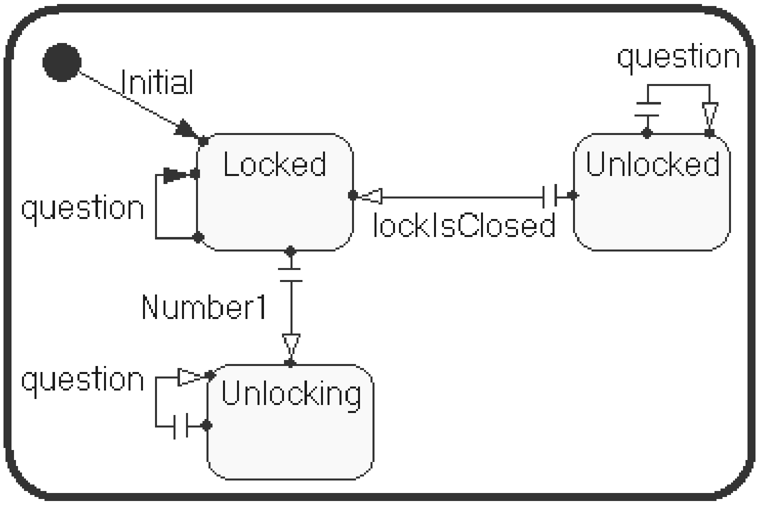


Figure 4.4 - Transitions and States

7.9. Define the *lockIsClosed* transition. This transition is used to relock the lock once it has been unlocked. It fires only when the lock is unlocked and the character L is typed.

7.9.1. Double-click the *lockIsClosed* transition to open the Transition Specification dialog. Move the dialog box far enough to one side that you can see the state diagram.

7.9.2. Go to the Triggers tab.

7.9.3. Right-click and select **Insert**.

7.9.4. The Event Editor dialog will appear. In the **Guard code** area, type:

num == 'L'

Note: Be careful to get the quote marks around the L as shown in Figure 4.5.

7.9.5. Select *Number* from the drop-down box.

7.9.6. Click OK.

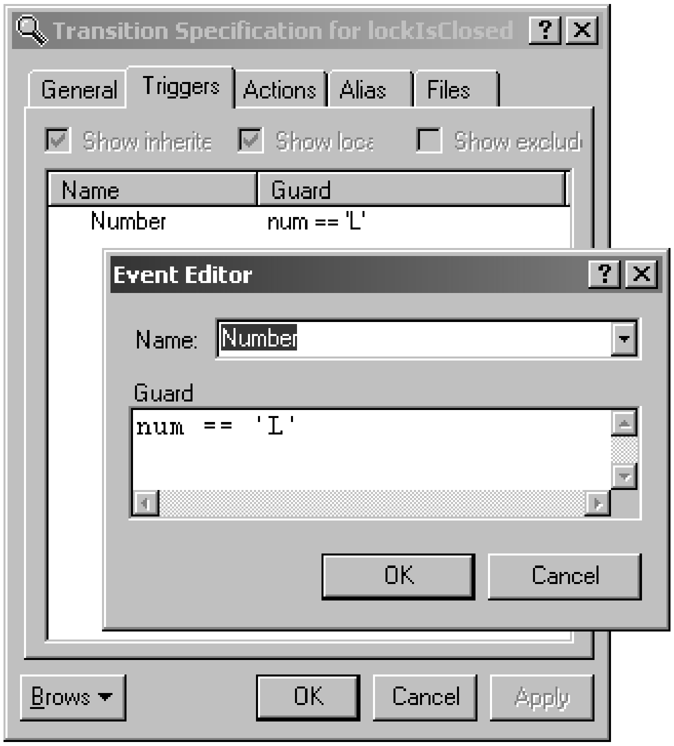
Notice that the transition line is now solid. The capacitance marks disappear when you define the trigger.

7.9.7. Go to the `Actions` tab of the Transition specification dialog and enter the following in the Code window:

```
printf("The lock is LOCKED\n");
```

7.9.8. Click OK.

Notice that the transition arrowhead is now filled.



Name	Guard
Number	num == 'L'

Name: Number

Guard

```
num == 'L'
```

Figure 4.5 - Transitions Specification & Event Editor

7.10. Define the *Number1* transition. This transition fires when the lock is in the locked state and the character 1 is entered as the first character in the unlock sequence. When the transition is taken, the state machine goes into the Unlocking state where it waits for another character in the sequence.

7.10.1. Double-click the *Number1* transition to open the Transition Specification dialog.

7.10.2. Go to the `Triggers` tab.

7.10.3. Right-click and select **Insert**.

7.10.4. Select *Number* from the drop-down box.

7.10.5. In the Guard window, type: `num == '1'`

Note: Be careful to get the quote marks around the one.

7.10.6. Click OK to exit the Event Editor.

7.10.7. Click OK to exit the Transition Specification dialog.

7.11. Define the *question* self-transition on the *Unlocked* state. This transition is taken if a question mark is entered. Normally, instructions on what to do would be placed here. For now, just print a message of acknowledgement.

7.11.1. Double-click the *questions* transition on the *Unlocked* state to open the Transition Specification dialog.

7.11.2. Go to the Triggers tab.

7.11.3. Right-click and select **Insert**.

7.11.4. Select *Number* from the drop-down box.

7.11.5. In the Guard window, type: `num == '?'`

7.11.6. Click OK.

7.11.7. Go to the Actions tab of the Transition specification dialog and enter the following in the Code window:

```
printf ("Unlocked Help Goes Here\n");
```

7.11.8. Click OK.

Once the state machine is in the Unlocking state, it needs to respond according to the input received. To do this, you will add a choice point to filter out the question mark and another to determine if the input is a two or not.

7.12. Create a choice point to determine if the input is a question mark.

7.12.1. Select the choice point tool and create a choice point named *IsQuestion*.

7.12.2. Double-click the choice point to open the Choice Point Specification dialog.

7.12.3. Go to the Condition tab.

7.12.4. Enter the following condition:

num == ‘?’

7.12.5. Click OK.

Notice that there is now a “C” in the choice point. This indicates that there is a condition defined for the decision.

7.13. Create a choice point to determine if the input is a two or not.

7.13.1. Select the choice point tool and create a choice point named *IsTwo.*

7.13.2. Open the Choice Point Specification dialog.

7.13.3. Go to the Condition tab.

7.13.4. Enter the following condition:

num == ‘2’

7.13.5. Click OK.

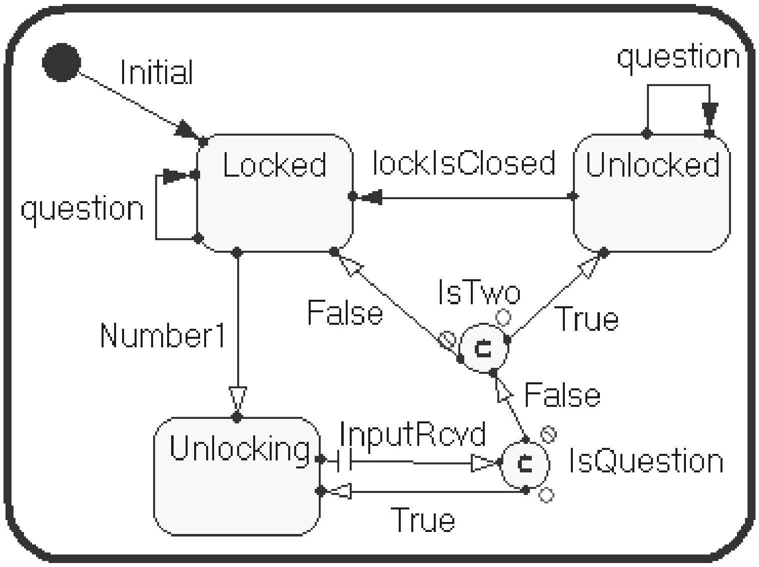


Figure 4.6 - Lock State Diagram Showing Transitions

7.14. Create the transitions shown in Figure 4.6.

7.15. Define the *InputRcvd* transition.

7.15.1. Double-click the *InputRcvd* transition to open the Transition Specification dialog.

7.15.2. Go to the `Triggers` tab.

7.15.3. Right-click and select **Insert**.

7.15.4. Select *Number* from the drop-down box.

7.15.5. Click OK to exit the Event Editor.

7.15.6. Click OK to exit the Transition Specification dialog.

7.16. Add code to the *True* transition leaving *IsQuestion* to handle the case where a question mark is input. The transition returns to the *Unlocking* state to wait for additional input.

7.16.1. Double-click the *True* transition to open the Transition Specification dialog.

7.16.2. Go to the `Actions` tab and enter the following in the Code window:

```
printf ("Unlocking Help Goes Here\n");
```

7.16.3. Click OK.

7.17. Add code to the *True* transition leaving *IsTwo* to handle the case where a two is entered to unlock the lock.

7.17.1. Double-click the *True* transition to open the Transition Specification dialog.

7.17.2. Go to the Actions tab and enter the following in the Code window:

```
printf ("The lock is UNLOCKED\n");
```

7.17.3. Click OK.

7.18. Add code to the *False* transition leaving *IsTwo* to handle any entry other than two. If the second digit isn't a two, the lock resets to the *Locked* state.

7.18.1. Double-click the *False* transition to open the Transition Specification dialog.

7.18.2. Go to the Actions tab and enter the following in the Code window:

```
printf("Sorry, that is not the right
combination\n");
```

7.18.3. Click OK.

7.19. Save the model. At this point it should look something like this:

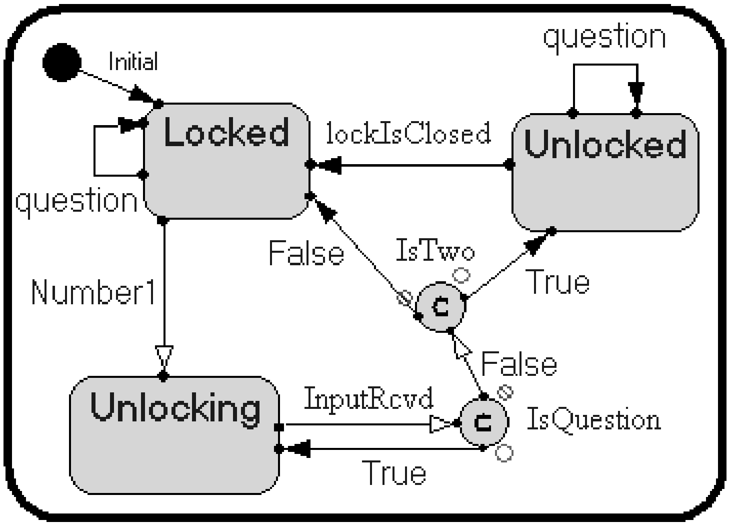


Figure 4.7 - Finished Model

8. **Create a new component.**

8.1. Right-click the Component View folder.

8.2. Select **New** > **Component.**

8.3. Name the new component *LockSystem*.

8.4. Drag the *ElectronicLock* class (from the *Logical View*) onto the component.

8.5. Set the component to active.

8.6. Double-click the *LockSystem* component to open the Component Specification.

8.7. Select the General tab.

8.8. Set **Environment** to *C++* .

8.9. Click OK to save the settings.

8.10. Click on the Build Component tool 🔨.

8.11. If the build is successful, continue. Otherwise, go back and check your work.

9. Create a new processor.

9.1. Right-click the Deployment View folder.

9.2. Select **New** > **Processor**.

9.3. Name the processor *Workstation*.

10. Compile, run, and debug *Lock*.

10.1. Drag the *LockSystem* component onto *Workstation*.

10.2. Double-click the component instance in the Deployment View to display the Component Instance Specification dialog.

10.3. On the Detail tab, clear the box labeled **Attach to target on startup**.

10.4. Click OK.

10.5. Set the component instance to *active* by right-clicking the instance and selecting **Set As Active.**

10.6. Click the **Run** button ⚡. A dialog box appears. Click Yes to build and execute the model.

10.6.1. A Command Prompt window will appear to show the results. Test the combination, relock the lock, and test the two question transitions.

10.6.2. To end execution, close the Command Prompt window, right-click the component instance, and select Shutdown. (Ignore the error. Click OK to close the error dialog.)

Warm-Up 5: Battleship

This exercise accompanies Module 6, “System Services.”

In this exercise, you will create a simple simulation model composed of a battleship that sends sonar signals (*pings*) into the ocean at regular time intervals, and detects and displays returned signals (*echoes*). The model will have simple structure and behavior and use Timer and Log services.

Objectives

After successfully completing this exercise, you will be able to:

- Create a new model from existing model elements.
- Trigger capsule behavior with a timer, using the Timing Service in the services library.
- Display text messages from a capsule, using the Log Service in the services library.
- Compile, run, debug, and verify the model using sequence diagrams generated during run time.

Instructions

1. Open the warm-up exercise model named *Battleship*. This model should be located at C:\MRRT\WarmUp\. If this is not the case, consult your instructor.
2. Save the model in your StudentWork directory.

   2.1. Click Yes at the prompt to create a new workspace.
3. Create a capsule class named *World*, which is the top-level capsule for this application.
4. Place a capsule role, based on the existing capsule class *Ocean*, within *World*. Name it *ocean*.
5. Using the sequence diagram shown on the next page as requirements, design a capsule named *Battleship* that implements the functionality of the object, *battleship*, in the diagram.

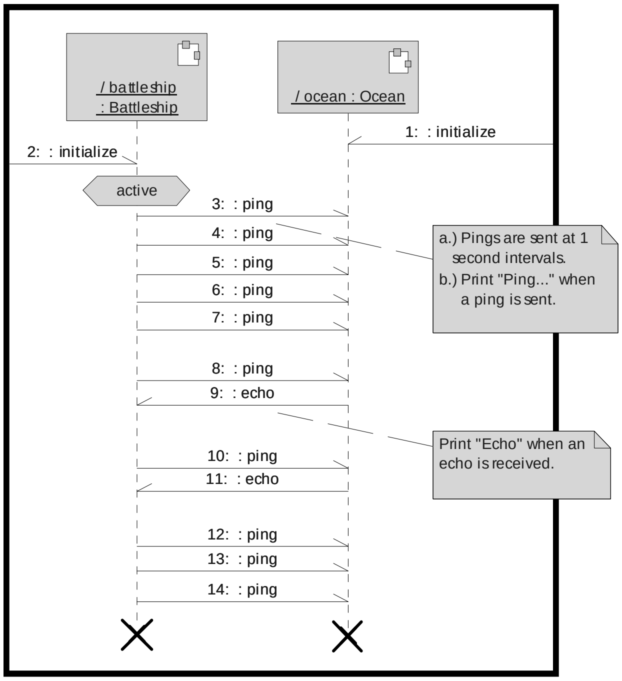


Figure 5.1 - Battleship Sequence Diagram

6. Place a capsule role, based on *Battleship*, in *World*. Connect it to *ocean*.

7. Because a system service is required, you must show the dependencies.

   7.1. Open the *Main* class diagram in the Logical View.

   7.2. Drag *Battleship*, *Ocean*, and *RTTimespec* onto the diagram. (*RTTimespec* is in *RTClasses*.)

   7.3. Using the Dependency tool, draw a dependency from *Battleship* to *RTTimespec* and from *Ocean* to *RTTimespec*.

8. Compile, run, and debug *World*.

9. Verify that the model you have constructed is complete, using a sequence diagram generated during run time.

   9.1. Select *ocean* in the Structure Monitor, then control-click *battleship* so they are both selected.

   9.2. Right-click on an empty part of the diagram and select **Open Trace**.

   9.3. Start the model and let it run until *battleship* receives **echo** messages and then goes back to getting no return message.

   9.4. Create the sequence diagram from the trace.

   9.4.1. Right-click in the trace window and select **Open Sequence Diagram**.

   9.4.2. Choose whether to save the diagram or not, then click **OK.**

10. Shutdown the model.

▶ ▶ ▶ Warm-Up 6: Traffic System

This exercise accompanies Module 7, “Requirements Analysis.”

In this exercise, you will create a simple traffic intersection application based on two, 2-lane roads that intersect. The intersecting roads run north / south and east / west. The *model* you create will have moderately complex behavior and structure.

Objectives

After successfully completing this exercise, you will be able to:

- Create a new model using an existing model and its elements (capsule and protocol classes).
- Create a new capsule class and define its structure and behavior.
- Create a containing capsule.
- Add structure to that capsule by adding capsule roles.
- Compile, run, and debug the new model.

Instructions

1. Open the *TrafficLight* model from Lab 3 that you saved to C:\MRRT\StudentWork. Use it as your starting point by saving it under the name *TrafficSystem*.
2. Use the sequence diagram on the next page as requirements for this model.
3. Create a capsule class named *Controller*.
   - 3.1. Define its structure. (Don’t overlook the fact that you need a timer.)
   - 3.2. Define its behavior.
4. Create a container capsule class named *Intersection*.
   - 4.1. Add a capsule role, based on *Controller*, named *controller*. This controls the color of the lights facing each of four directions.
   - 4.2. Add four capsule roles, based on *TrafficLight*, named *north*, *south*, *east*, and *west*. These are the four sides of your typical traffic light.
   - 4.3. Connect each of the four *TrafficLight* capsule roles to the appropriate side of the *controller* capsule role (one at each compass point) using the Connector tool. To keep the concept straight, *north* should be opposite *south*, *east* should be opposite *west*. Each pair (*north/south* and *east/west*) must always display the same color light.
5. Compile, run, and debug *Intersection*.

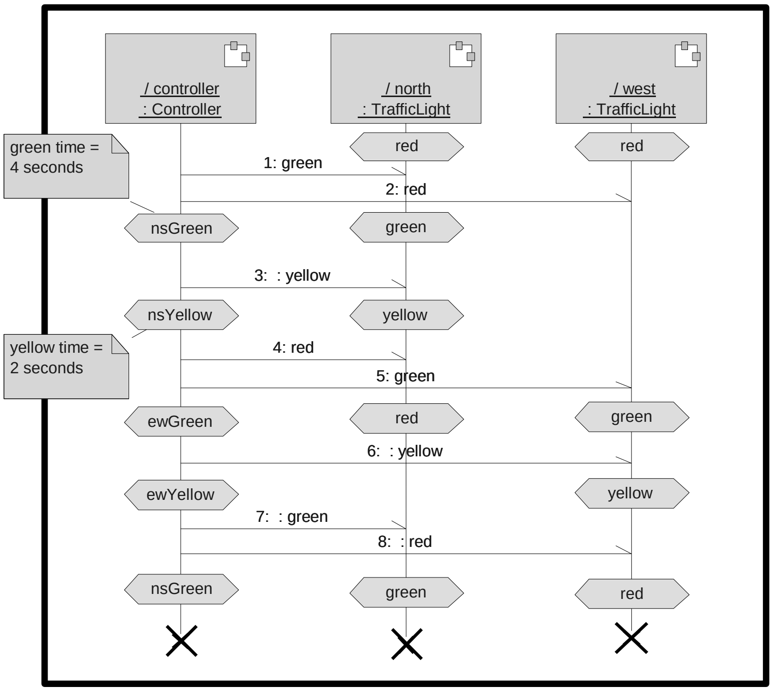


Figure 6.1 - Traffic System Sequence Diagram

▶ ▶ ▶ Warm-Up 7: Client/Server

This exercise accompanies Module 15, “Adaptive Modeling.”

In this exercise, you will create two simple client/server applications. In the first, the client will be incarnated by its container capsule, and in the other, the client will be imported by its container capsule. The models you create will have dynamic structure and behavior.

Objectives

After successfully completing this exercise, you will be able to:

- Create a model with capsules that are incarnated (dynamically created at run time).
- Create a model with capsules that are imported (dynamically “moved” at run time).
- Compile, run, and debug models that use incarnation and multiple containment.

Instructions

1. Open the *ClientServer* model in C:\MRRT\WarmUp.
2. Save the model as *ClientServerIncarnated*.
3. Open the structure diagram for the *TheSystem*.
4. Open the Capsule Role specification dialog box for *client* and change the setting from **Fixed** to **Optional**.
5. Add a protected, unwired port named *frame*, based on the *Frame* service.
   - 5.1. Select the Port tool and click inside the boundary of the *TheSystem* structure diagram. This creates a protected port.
   - 5.2. Select Frame from the list of protocols.
   - 5.3. Type the name of the new port.
6. Add an initial transition to *TheSystem’s* behavior (which is to say, from the initial state to the capsule boundary) containing the following code:

   ```
   frame.incarnate(client);
   ```
7. Build the model and remove any errors.
8. Run the model, watch *client* incarnate, and then use Trace to watch the *client* and *server* begin messaging.

9. Save the model as *ClientServerImported.*
10. Set *client* to *Plug-in* in its Specification dialog box.
11. Add a new client to *TheSystem,* named *realClient*.
12. Change the code in the initial transition of *TheSystem's* behavior to :

    ```
    frame.import(realClient, client);
    ```
13. Build the model and remove any errors.
14. Run the model, watch *client* receive the imported *realClient*, and then use Trace to watch the *client* and *server* begin messaging.

▶ ▶ ▶ Warm-Up 8: RQA-RT

This exercise accompanies Module 16, “Rational Quality Architect - RealTime.”

In this exercise, you will create a simple test harness for a client/server system. You will then use the test harness to perform unit testing.

Objectives

After successfully completing this exercise, you will be able to:

- Create a specification sequence diagram for an existing model.
- Create a test harness from the specification sequence diagram.

Instructions

1. Load *TestClientServer* in C:\MRRT\WarmUp\ as your starting point and save it in your folder.
2. Create a package named *TestResults* in the Logical View.
3. Create a collaboration diagram and name it *SingleClient*.
   - 3.1. Right-click the Logical View folder and select **New** > **Collaboration Diagram**.
   - 3.2. Name the diagram *SingleClient*.
   - 3.3. Double-click *SingleClient* to open the diagram.
4. Drag the Client and Server capsules onto the collaboration diagram and create the binding between them with the Association Tool ╲.
5. Create a specification sequence diagram for the model from the collaboration diagram. Name the diagram *HappyPath*. Creating a collaboration diagram keeps the test artifacts isolated from the rest of the model.
   - 5.1. Select the two capsules in the collaboration diagram. (It’s ok to select the association, too.)
   - 5.2. Right-click the empty part of the collaboration diagram and select **Create Sequence Diagram**. (Note: If you click the selected capsules, you will not get the right menu. See next page for example.)
   - 5.3. The new diagram is named *NewDiagram*. To rename it, right-click *NewDiagram* in the Logical View folder and select **Rename**.

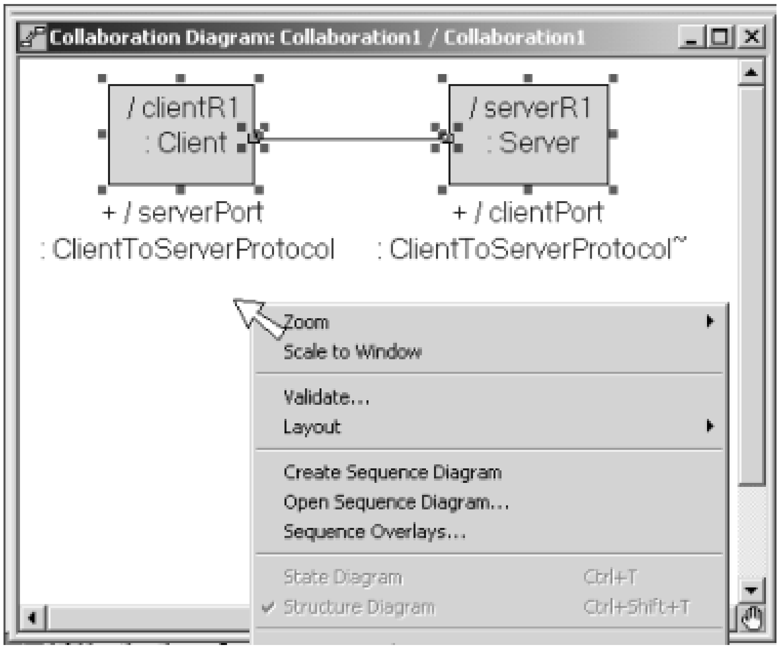


Figure 8.1 - Client/Server Collaboration Diagram

6. Draw an asynchronous message named *requestService* from *Client* to *Server*.

   6.1. Using the Asynchronous Send Message Tool →, click *Client's* lifeline and drag to *Server's* lifeline.

   6.2. Open the Send Message dialog to the `General` tab and type in the name of the message. (Double-click the message line.)

   6.3. Go to the `Port Detail` tab. Set the "From port", "To port" and "Signal" specifiers to indicate a "requestService" message from the client to the server.

7. Finish the sequence diagram as shown on the next page. The (6) on the serviceHandle and returnService messages denotes that there is data associated with the message, in this case the integer 6. You need to specify the data for these two messages on the `Detail` tab of the Send Message specification. For this part of the exercise, it doesn't matter what value you give the data.

8. Create a component called *TestComponent* with the correct Target Configuration.

9. Create a processor called *TestProc* using default settings.

10. Right-click the *HappyPath* sequence diagram, and select Verify Behavior from the context menu. Since RQA-RT has never been run on this collaboration, a dialog box is displayed asking you if you want to use the wizard. Click [Yes].

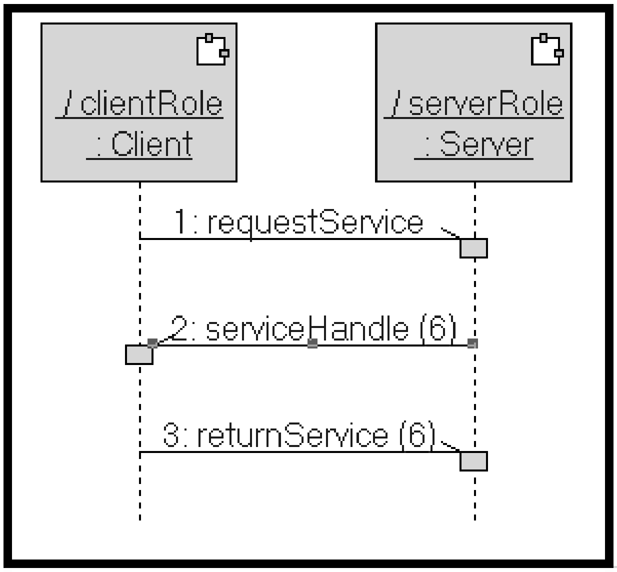


Figure 8.2 - Client/Server Sequence Diagram - One Client

11. On the first page of the wizard, name the test *TestHappyPath.* Select the **Reuse selected component** check box and select TestComponent as the component and *TestProc* as the processor.
12. On the second page of the wizard, select the *clientR1* check box. The client will be generated by RQA-RT, based on the behavior described in the sequence diagram. This lets you test the server.
13. Leave the settings on the third page to the defaults. Click **Next** to continue.
14. Clear **Store results with generated harness**, and then find the *TestResults* package using the browser. Finish the wizard.
15. Click Verify on the Verify Behavior setup dialog. RQA-RT now generates, compiles, links and runs the model, while gathering information about the execution. This information automatically is compared to the specification sequence diagram.
16. After the comparison, a message box is displayed telling you to look in the log. If all is well, there should be no differences. When a difference is reported, there are two parts to each difference. Double-click one side to see the difference on the specification sequence diagram, and the other to see the difference on the sequence diagram generated by RQA-RT.
17. Look in the TestResults package to see the trace sequence diagram and test results summaries. The test harness that was created by the Verify Behavior process is in the package named RQART_TestHappyPath. You can use this test harness again for regression testing.

If there is time remaining, you may do the following exercise.

Drivers and Data

In this exercise, you will run two clients with one server. Assume that the server will only hand out one service at a time.

Objectives

After successfully completing this exercise, you will be able to:

- Create multiple drivers.
- Test with data.

Instructions

1. Load *TestClientServer2* from C:\MRRT\WarmUp\ as your starting point and save it in your StudentWork directory.
2. Create a collaboration diagram in *SpecificationPackage* called *MultipleClients*.
3. In the collaboration diagram, create a capsule role based on *Server* and two capsule roles based on *Client*.
4. Connect the clients to the server.
5. Create a sequence diagram from *MultipleClients* named *MultipleRequest*. Add elements as shown in the sequence diagram on the next page. Don’t forget to specify the ports in the **Send Message** dialog.
6. Create a package for your test results.
7. Create a processor.
8. Run the Verify Behavior process with the client roles set as drivers. This generates both client interactions as drivers. They each have their own behavior as specified in the sequence diagram.
9. Save the Options settings generated by the Verify Behavior Wizard for later use. The settings are relative to the collaboration used.

   9.1. When the wizard is done, the Verify Behavior dialog box is displayed. Save your settings using the **Save** button at the top of the dialog box. You can see what your options are set to by clicking Options... in the lower left of the dialog box.

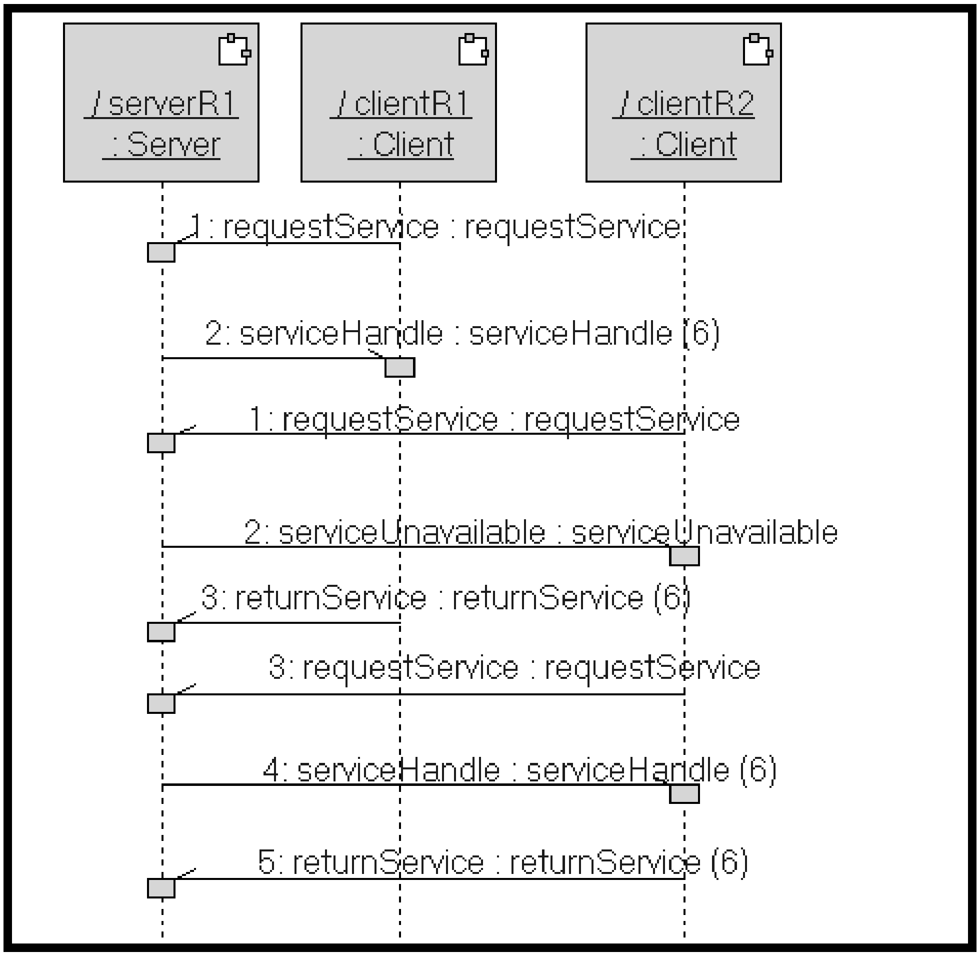
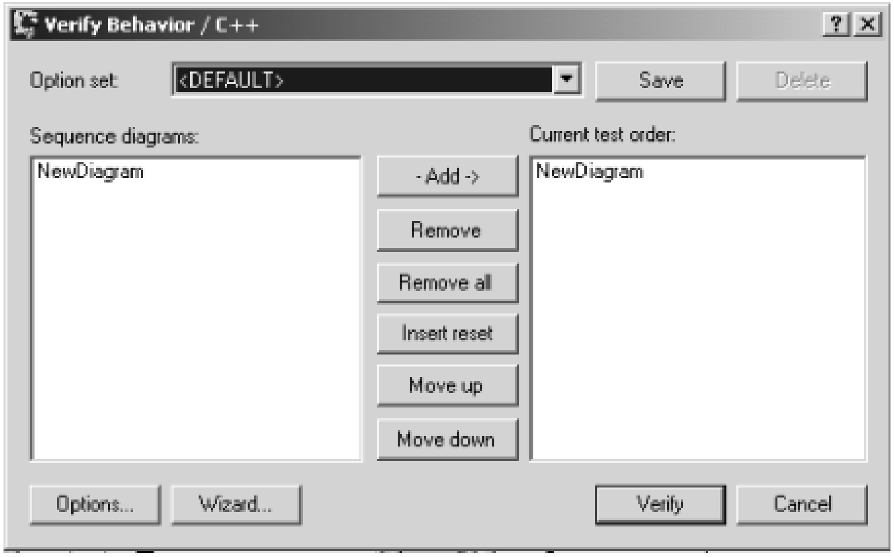

/ serverR1
: Server
/ clientR1
: Client
/ clientR2
: Client
1 : requestService : requestService
2: serviceHandle : serviceHandle (6)
1 : requestService : requestService
2: serviceUnavailable : serviceUnavailable
3: returnService : returnService (6)
3: requestService : requestService
4: serviceHandle : serviceHandle (6)
5: returnService : returnService (6)

Figure 8.3 - Client/Server Sequence Diagram with Data



Option set: <DEFAULT>

Sequence diagrams: NewDiagram

Current test order: NewDiagram

Figure 8.4 - RQA-RT Verify Behavior Dialog

Next you will test data sent with the messages. You do this by changing the *serviceRequest* signal to request a service of a specific type, identified by an integer. The server is required to send back a *serviceHandle* with the same value as in the *serviceRequest.*

10. Save your model as *TestClientServer2withData*.

11. Create a new sequence diagram named *RequestWithData* by duplicating *MultipleRequest* then renaming the duplicate.

12. Modify the *requestService* signal to accept an integer data type.

    12.1. Open the *RequestWithData* sequence diagram.

    12.2. Double-click the *requestService* message to open the Send Message Specification dialog box and go to the `Port Detail` tab.

    12.3. Click the *Signal* link (blue, underlined label) to open its Signal Specification.

    12.4. Change the **Data class** to *int*.

    12.5. Click OK.

13. Add the *requestService* data.

    13.1. Go to the `Detail` tab.

    13.2. Enter the number 6 in the **Data** box.

    13.3. Click OK.

14. Define a variable in the *clientR1* interaction.

    14.1. To open the Interaction Instance Specification dialog box, right-click the *ClientRole1* interaction in the sequence diagram and select **Open Specification**.

14.2. Go to the `Quality Architect-RT` tab.

14.3. In the **Attributes** window enter:

```
serviceNumber : int
```

This defines a variable that you can use as an attribute within this interaction.

14.4. Click OK.

15. Create a local action above the first message on the *clientR1* interaction.

15.1. Using the Local Action tool, click the lifeline for *clientR1* just above the first message.

15.2. Double-click the local action to open the Local Action Specification dialog box.

15.3. Go to the `Quality Architect-RT` tab. In the **Code** box, enter:

```
serviceNumber = 6;
```

15.4. This code is added to the test harness and executes before the first message.

15.5. Click OK.

16. Open the Send Message Specification dialog box for the response message from the server to *clientR1*. Open to the `Quality Architect-RT` tab. In the Receiver Test Driver Code box enter the following:

```
if( *( (int*)(msg -> getData())) != serviceNumber)
  SendACompareFailure (“Servicenumber mismatch”);
```

This states that the data on the message from the model to the driver needs to have data that equals *serviceNumber*. If it doesn’t, a *CompareFailure* is logged by RQA-RT.

16.1. Click OK.

17. Add data for the messages as shown in the sequence diagram on the next page and modify *clientR2* in the same manner as for *clientR1*.

18. Invoke Verify Behavior. Note that the tool checks if the answer from the server is the one expected.

19. Change the code so that the test fails. (The easiest way is to change the value of serviceNumber in the local action. Notice how the difference is reported in the log.

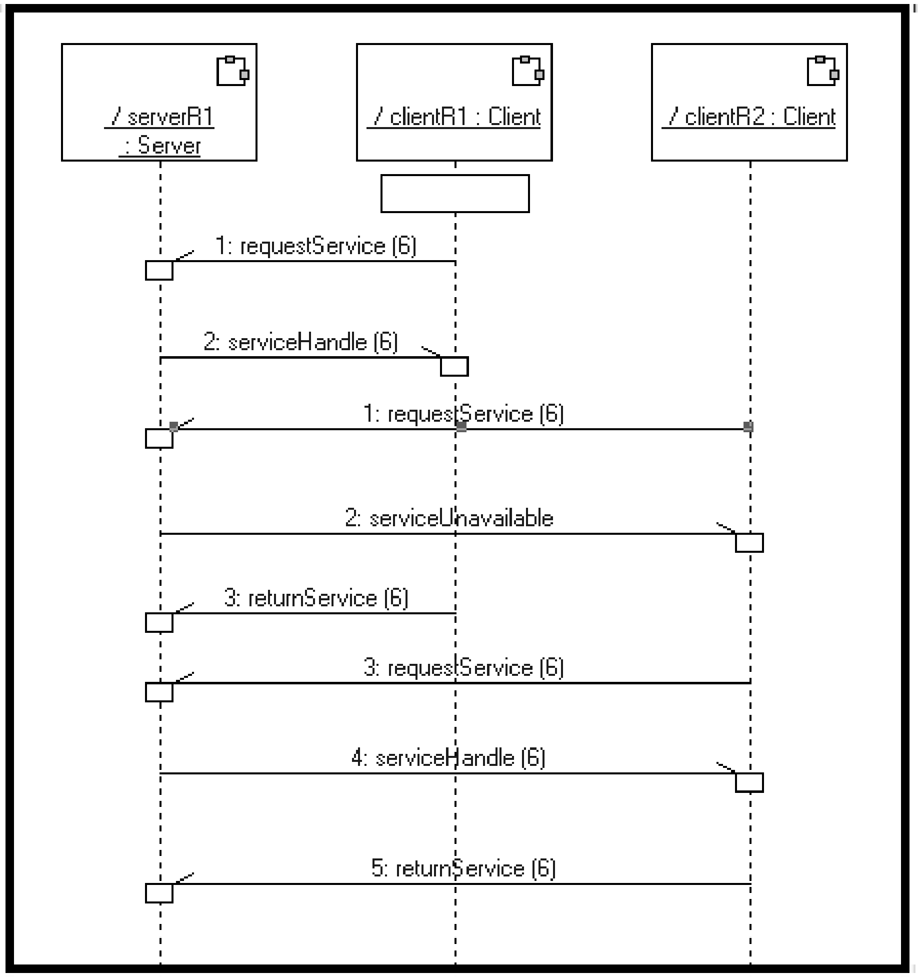


Figure 8.5 - Client/Server Sequence Diagram with Local Actions