

# VEST User's Manual

## Version 4.1

Released June 2004

VEST Group  
Department of Computer Science  
University of Virginia  
Contact: John A. Stankovic  
[stankovic@cs.virginia.edu](mailto:stankovic@cs.virginia.edu)  
<http://www.cs.virginia.edu/brochure/profs/stankovic.html>



**University of Virginia**

<b>1</b>	<b>WHAT'S NEW IN RELEASE 4.1 .....</b>	<b>4</b>
1.1	VEST TOOL .....	4
1.2	UPDATES TO USER'S MANUAL .....	4
<b>2</b>	<b>INTRODUCTION.....</b>	<b>5</b>
2.1	OVERVIEW OF VEST .....	5
2.2	OVERVIEW OF GME .....	6
<b>3</b>	<b>INSTALLATION.....</b>	<b>7</b>
3.1	INSTALLING GME .....	7
3.2	INSTALLING VEST .....	7
3.2.1	<i>Loading the VEST metamodel in GME .....</i>	<i>7</i>
3.2.2	<i>Registering the VEST metamodel.....</i>	<i>10</i>
3.2.3	<i>Building the VEST Interpreters.....</i>	<i>12</i>
3.2.4	<i>Loading and Running a VEST project.....</i>	<i>12</i>
3.2.5	<i>Manually Adding Interpreters to VEST.....</i>	<i>13</i>
<b>4</b>	<b>THE VEST TOOL .....</b>	<b>15</b>
4.1	COMPOSITION ENVIRONMENT .....	15
4.1.1	<i>Model Editing Window.....</i>	<i>15</i>
4.1.2	<i>Project Browser .....</i>	<i>16</i>
4.1.3	<i>Component Browser.....</i>	<i>18</i>
4.1.4	<i>Attribute Window .....</i>	<i>18</i>
4.1.5	<i>Menu Bar .....</i>	<i>19</i>
4.1.6	<i>Tool Bar .....</i>	<i>20</i>
4.1.7	<i>Mode Bar .....</i>	<i>21</i>
4.1.8	<i>Status Bar.....</i>	<i>22</i>
4.1.9	<i>Title Bar .....</i>	<i>22</i>
4.2	ASPECT CHECKS .....	22
4.2.1	<i>Buffer-Size Check.....</i>	<i>23</i>
4.2.2	<i>Memory Check .....</i>	<i>23</i>
4.2.3	<i>Event Check .....</i>	<i>23</i>
4.2.4	<i>RT Scheduling .....</i>	<i>24</i>
4.3	PREScriptive ASPECTS .....	25
<b>5</b>	<b>USING VEST.....</b>	<b>27</b>
5.1	BUILDING A SYSTEM DESIGN .....	27
	<i>Exercise (Time Required: 1-2 hours) .....</i>	<i>27</i>
5.1.1	<i>Creating a new VEST project.....</i>	<i>28</i>
5.1.2	<i>Creating a Model Layer.....</i>	<i>29</i>
5.1.3	<i>Creating Components .....</i>	<i>30</i>
5.1.4	<i>Creating Relationships between Components.....</i>	<i>31</i>
5.1.5	<i>Setting Component Attribute Values .....</i>	<i>33</i>
5.1.6	<i>Creating Sub-Components .....</i>	<i>34</i>
5.1.7	<i>Creating References.....</i>	<i>35</i>
5.1.8	<i>Creating Event Channels .....</i>	<i>36</i>
5.2	USING ASPECT CHECKS .....	38
5.2.1	<i>Memory Check .....</i>	<i>38</i>
5.2.2	<i>Buffer-size Check .....</i>	<i>39</i>
5.2.3	<i>Event Check .....</i>	<i>41</i>
5.2.4	<i>RT Scheduling Check .....</i>	<i>41</i>
5.3	USING PRESCRIPTIVE ASPECTS .....	44

<b>6</b>	<b>REFERENCES .....</b>	<b>49</b>
<b>7</b>	<b>APPENDIX 1: ACL TO VEST MAPPING .....</b>	<b>50</b>
<b>8</b>	<b>APPENDIX 2: VEST TO XML CONFIGURATION MAPPING .....</b>	<b>52</b>
<b>9</b>	<b>APPENDIX 3: RT SCHEDULING API.....</b>	<b>53</b>
<b>10</b>	<b>APPENDIX 4: VPAL BNF SPECIFICATION .....</b>	<b>55</b>
<b>11</b>	<b>APPENDIX 5: DESIGNING PRISM SYSTEMS IN VEST .....</b>	<b>57</b>
11.1	HARDWARE LAYER .....	57
11.2	SOFTWARE LAYER .....	57
11.2.1	<i>Event Channels.....</i>	<i>59</i>
11.2.2	<i>Receptacles/Facets .....</i>	<i>60</i>
11.2.3	<i>Processor Mappings.....</i>	<i>60</i>
11.2.4	<i>Specifying Timeouts.....</i>	<i>60</i>

# 1 What's New in Release 4.1

## 1.1 *VEST Tool*

- VEST 4.1 includes an updated meta-model that
  - works with GME version 4.3.17
  - contains new event channel/path modeling
  - has previous inconsistencies and errors corrected
    - § All Booleans are enumerations of type {0,1}
    - § All attribute data types are set correctly (eg. WCET is of type Integer, not String)
- All Product Scenario files have been updated with new metamodel
- Event path modeling has been updated in order to reflect the Boeing Boldstroke Event Path more accurately.
  - Each event is assigned a sequence in the corresponding VEST model. The sequence is specified by the system designer.
  - When the designer performs schedulability analysis, VEST makes sure that the events/methods are assigned to proper time slots based on the assigned sequence and then calls the appropriate scheduling algorithms
- Prescriptive Aspect interpreter code is significantly improved for usability and robustness
  - CREATE statement is now implemented. Previously, only GET and SET statements were supported
- Build process for VEST interpreters is simplified and automated

## 1.2 *Updates to User's Manual*

- Documentation for the following are included in this update:
  - Building the VEST interpreters
  - Events Check Description and Usage
- Figures and appendices updated

## 2 Introduction

### 2.1 Overview of VEST

VEST (Virginia Embedded Systems Toolkit) provides an environment for constructing and analyzing component-based distributed real-time embedded systems. VEST helps developers select or create passive software components, compose them into a product, map the passive components onto active structures such as threads, map threads onto specific hardware, and perform dependency checks and non-functional analyses to offer as many guarantees as possible along many dimensions including real-time performance and reliability. Distributed embedded systems issues are explicitly addressed via the mapping of components to active threads and to hardware, the ability to include middleware as components, and the specification of a network and distributed nodes.

The VEST environment is composed of the following:

- 4 Component Libraries
- Prescriptive Aspects Library
- Aspect Checks
- Composition Environment

A brief description of each of these follows. For more detailed information, please refer to [1].

- **Component Libraries:** Because VEST supports real-time distributed embedded systems, the VEST component libraries contain both software and descriptions of hardware components and networks. VEST components can be *abstract* or *actual*. An abstract component is a design entity that represents the requirements, e.g., a timer with certain requirements or a generic processor is an abstract component. An actual component is the implementation or description of a reusable entity. A specific timer module written in C and a Motorola MPC7455 are examples of actual components. Sets of reflective information exist for each of these component types. The reflective information of an abstract component includes its interface and requirements such as for security. The reflective information for actual components includes categories such as linking information, location of source code, worst-case execution time, memory footprint, and other reflective information needed to analyze crosscutting dependencies. The extent of the reflective information and its extensibility are some of the key features that distinguish VEST from many other tools. To support the whole design process of embedded systems, VEST implements four component libraries: the application library, middleware library, OS library and a hardware library.
- **Prescriptive Aspects Library:** Prescriptive aspects are reusable programming language independent advice that may be applied to a design. For example, a

developer can invoke a set of prescriptive aspects in the library to add a certain security mechanism *en masse* to an avionics product.

- **Aspect Checks:** VEST implements both a set of simple intra- and inter-component aspect checks that crosscut component boundaries. A developer can apply these checks to a system design to discover errors caused by dependencies among components. One aspect check in VEST is the real-time schedulability analysis for both single-node and distributed embedded systems. VEST can also invoke off-the-shelf analysis tools from its GUI environment.
- **Composition Environment:** VEST provides a GUI-based environment that lets developers compose distributed embedded systems from components, perform dependency checks, and invoke prescriptive aspects on a design.

VEST has been implemented using GME (Generic Modeling Environment). GME [2] is a meta-modeling environment with an extensible collection of model editing tools supporting a Model-Based Development approach to system/software engineering. More detailed information on GME is provided in the following subsection.

## 2.2 Overview of GME

The Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems at Vanderbilt University is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the *family* of models that can be created using the resultant modeling environment.

The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. The generated domain-specific environment is then used to build domains. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools. This process is called model interpretation.

The relationship between VEST and GME can be described as follows. The component libraries of VEST are implemented as meta-models in GME. When designing a system using the VEST metamodel, GME provides the underlying support for the graphics and modeling support. The system design developed in VEST can be exported as XML files, which make it possible to import the design to other related tools. However, all dependency checks and the prescriptive aspect interpreters are VEST specific and are implemented as *interpreters* that can be invoked through the buttons on the VEST GUI. All interpreters are implemented in C++ and can access the internal data structures that represent the component-based embedded system design.

## 3 Installation

### 3.1 Installing GME

To use VEST, GME first needs to be installed on your system. GME is available at <http://www.isis.vanderbilt.edu/projects/gme/>. Instructions on how to install and run GME can be found on this website. VEST users should first become familiar with GME and the basic GME GUI commands and interfaces. All GUI capabilities in VEST are similar to GME.

### 3.2 Installing VEST

Once GME is installed on your system, the VEST metamodel needs to be downloaded and loaded into GME. The VEST metamodel is included in your download in the following file: [Install Directory]/VEST/vest.xml. Although distributed in XML format, it can be available in either XML format and as a GME project file.

In order to create any VEST design, the VEST metamodel first needs to be registered with your system. There are two major steps involved here:

- Loading the VEST metamodel in GME
- Registering the VEST metamodel

#### 3.2.1 Loading the VEST metamodel in GME

There are two ways to load the VEST metamodel in GME. The first way is using an XML file and the other is using a GME project file. Follow the steps outlined below depending on the type of file that you have.

##### 3.2.1.1 XML File

*Step 1:*

- Launch GME

*Step 2:*

- Select “Import XML...” under the File menu as shown in Figure 3-1

*Step 3:*

- This brings up a file selector dialog box as shown in Figure 3-2
- Select the VEST metamodel XML file that you downloaded earlier
- Click “Open”

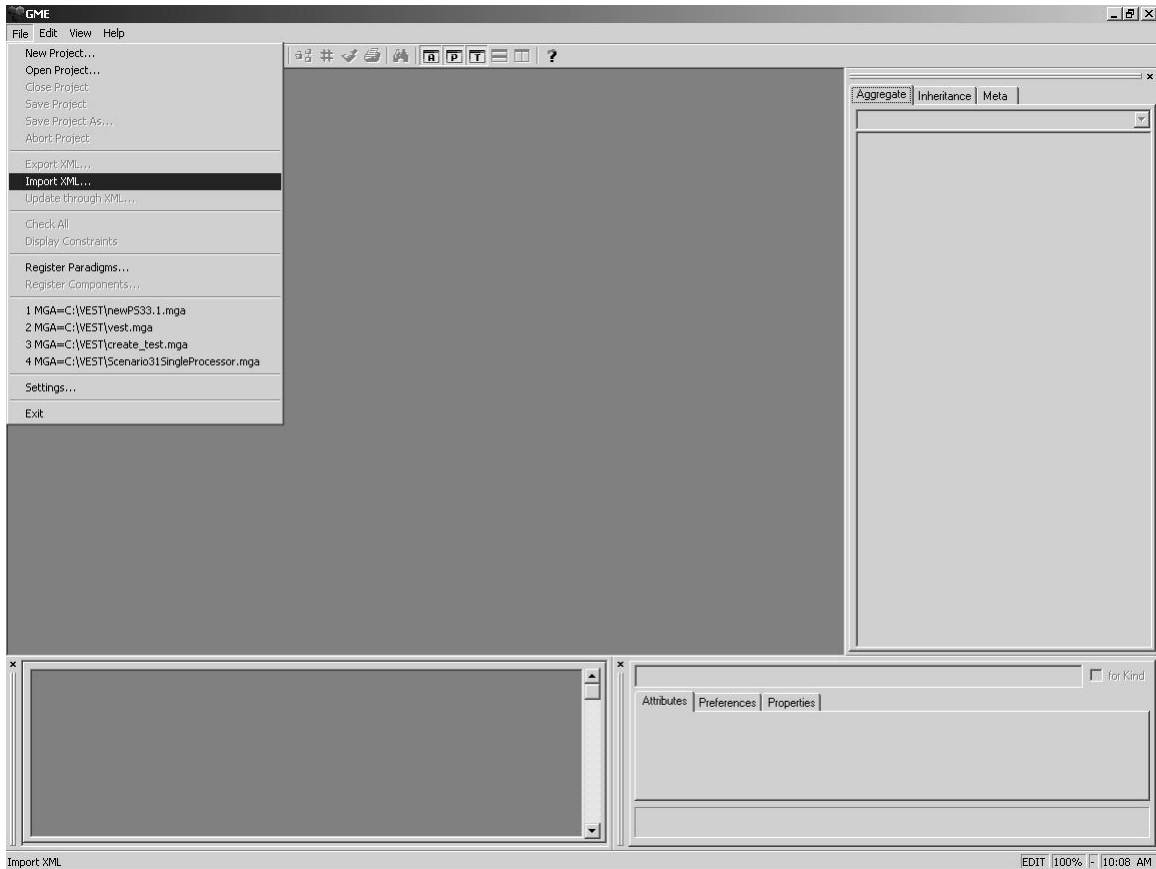


Figure 3-1

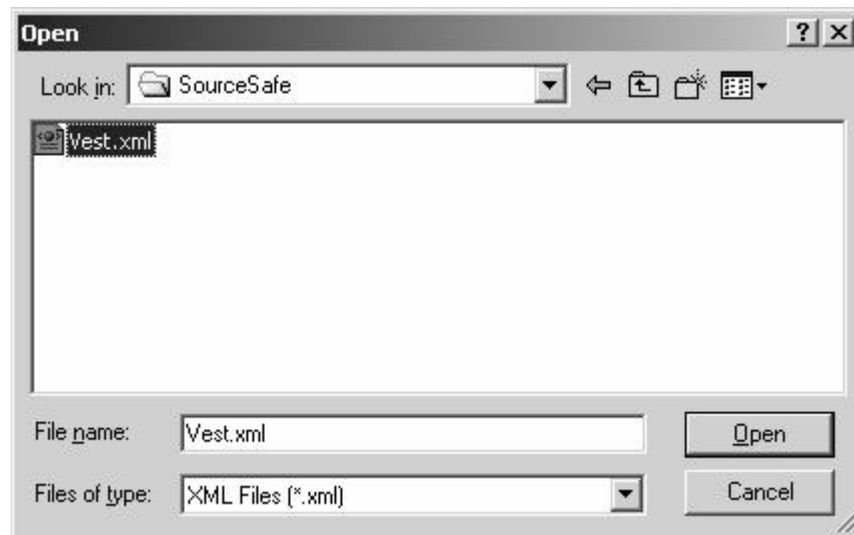
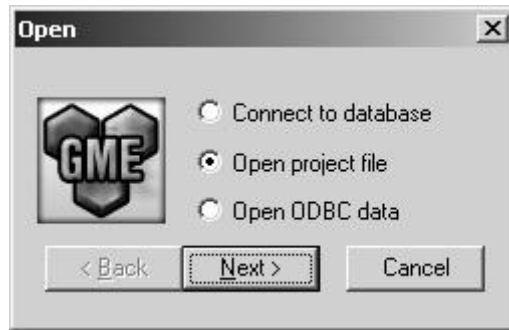


Figure 3-2

**Step 4:**

- GME reads the XML data and brings up the dialog box shown in Figure 3-3
- We will create a GME project file from our VEST metamodel data and so click “Next”





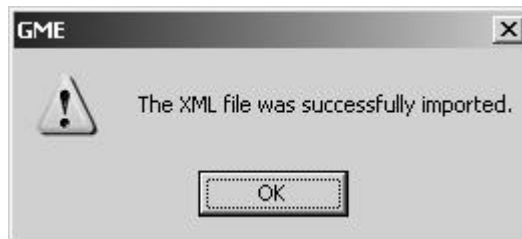
**Figure 3-3**

*Step 5:*

- GME then will ask you to name the project file. Name this file “Vest”
- Click “Open”

*Step 6:*

- GME creates a GME project file from the XML data and then displays the success box shown in Figure 3-4
- Click “OK”
- The VEST metamodel has been successfully loaded in GME at this point
- Now the VEST metamodel needs to be registered with the system. Skip to Section 3.2.2 to perform this step



**Figure 3-4**

### **3.2.1.2 GME Project File**

*Step 1:*

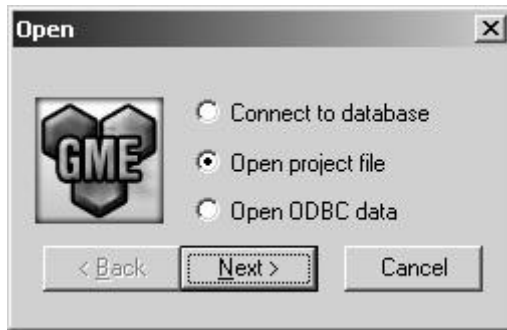
- Launch GME

*Step 2:*

- Open the GME project (mga file extension) containing the VEST metamodel using the “Open Project...” option under the File menu

*Step 3:*

- GME brings up the dialog box shown in Figure 3-5
- Since our GME project is stored in a file, we leave the default setting and click “Next”



**Figure 3-5**


*Step 4:*

- A Open File selector dialog box similar to the one in Figure 3-2 appears
- Select the GME project file containing the VEST metamodel that you downloaded earlier
- Click “Open”
- GME reads the project file and loads the VEST metamodel
- The VEST metamodel now needs to be registered with the system. Continue with Section 3.2.2

### **3.2.2 Registering the VEST metamodel**

We need to interpret and register the Vest metamodel with the system so that we can create VEST projects in GME. Follow these steps to register the VEST metamodel:

*Step 1:*

- Click the MGA Interpreter button (icon ) located on the GME toolbar
- This brings up a “Save As” dialog box to indicate the name of the XML paradigm file name to save the interpreted VEST metamodel to
  - Select the default file name and click “Save”
- This brings up the Global Aspect Order Dialog box as shown in Figure 3-6
  - Select the default order by clicking “OK”
- This brings up the Aspect<sup>1</sup> Mapping dialog box shown in Figure 3-7
  - Click “OK”

*Step 2:*

- GME interprets the VEST metamodel
- Once the model is successfully interpreted, GME shows the dialog box shown in Figure 3-8 and asks if you would like to register the VEST paradigm
- Click “Yes”

---

<sup>1</sup> The concept of Aspect in GME is really a particular view that a user sees. This aspect has nothing at all to do with the use of the term aspect in VEST itself. In VEST, aspects refer to the main principle underlying aspect-oriented programming

- If no error messages are displayed, the VEST paradigm is successfully registered with your system



Figure 3-6

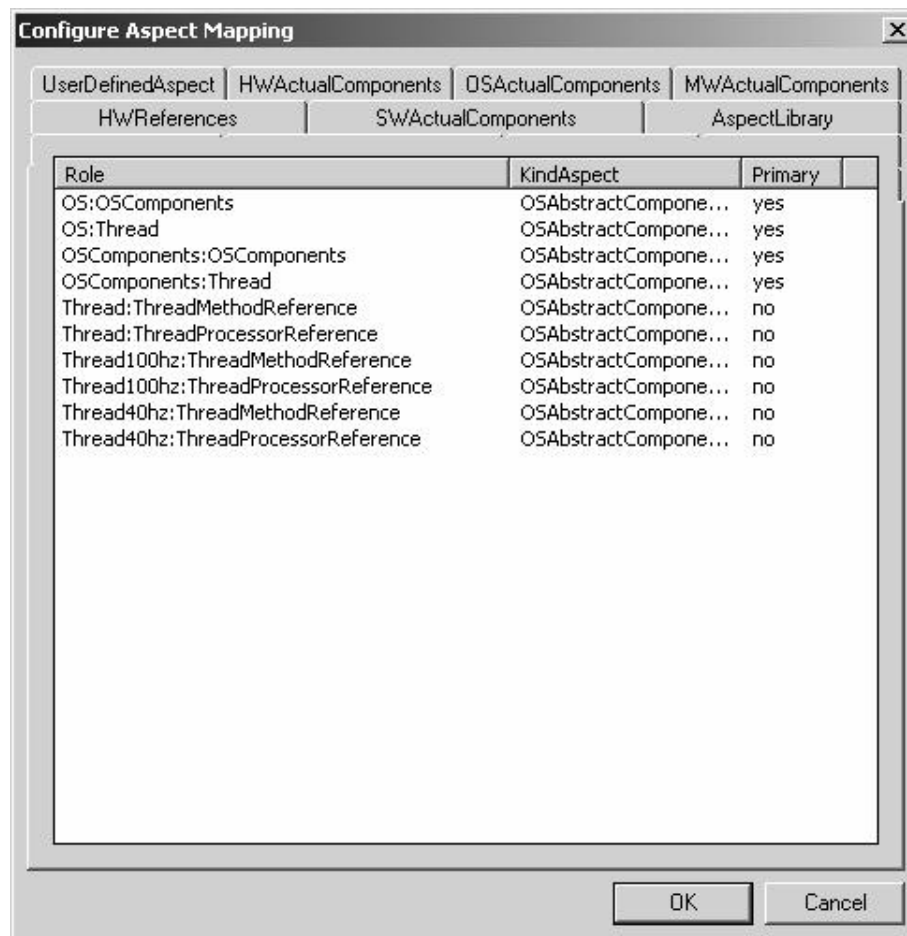


Figure 3-7

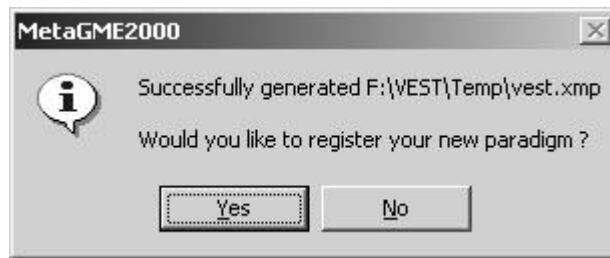


Figure 3-8

### 3.2.3 Building the VEST Interpreters

At this point, you can load and create VEST projects but none of the VEST interpreters will be available for these projects! To use the VEST interpreters, you first need to build the interpreter DLLs on your system. Your current VEST download includes all the code for all of the VEST interpreters. The interpreters can be built using Microsoft Visual Studio .NET 2003. If you do not have this version of Visual Studio, you will need to get it before you can build the interpreters.

To build the interpreters, follow these steps:

- In the [InstallDir]/VEST directory, open the solution named “VestInterpreters.sln” in Visual Studio .NET 2003
- Under the “Build” menu, check the “Configuration Manager” settings. Make sure it is set to “Release”. Change it to “Release” if it is not set to that
- Select “Build Solution” under the Build menu. The interpreters should begin building
- You should not encounter any errors. If you do, contact the VEST team
- Once successfully built, you are ready to load and run a VEST project

### 3.2.4 Loading and Running a VEST project

Once your VEST metamodel is registered with the system, you can then load and run VEST projects. Follow these steps to load and run a VEST project included with your download:

#### 3.2.4.1 XML Format

*Step 1:*

- Open the “Scenario32MultiProcessor” project using the “Import XML” option under the file menu
  - The project file is located at [InstallDir]/VEST

*Step 2:*

- You will see a “Import to new project” dialog box similar to the one in Figure 3-5
  - Leave the default selection as we will store our project in a file

*Step 3:*

- Choose a name and location to store the project file
- Click “Open”
  - GME will read the XML data, convert it to a VEST project and load it

### 3.2.4.2 GME Project File

*Step 1:*

- Click “Open Project...” under the File menu
  - The dialog box in Figure 3-5 will appear

*Step 2:*

- Leave the default selection and click “Next”
  - An File Selector dialog box like Figure 3-2 will appear

*Step 3:*

- Select “Scenario32MultiProcessor.mga” from [InstallDir]/VEST
- Click “Open”
  - GME will open the VEST project and load it

After the project had loaded, you should observe several things. Refer to Figure 4-1

- The Project name “Scenario32MultiProcessor” should appear as the root tree item in the Project Browser window
- There should be 6 icons in the interpreters section as shown in Figure 3-9
- If any of these interpreters are missing from your toolbar, continue with Section 3.2.5
- If all of the icons listed are there, your VEST project has loaded correctly. Each of these checks will be described in detail in later sections

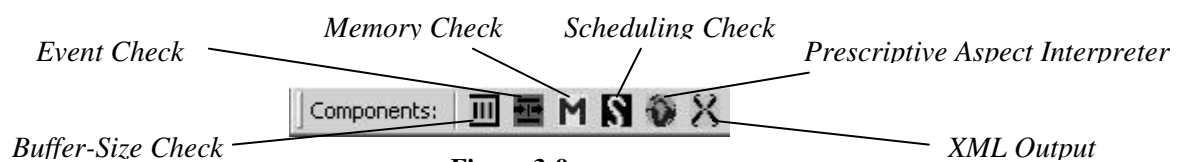


Figure 3-9

### 3.2.5 Manually Adding Interpreters to VEST

To manually add registered interpreters to the VEST toolbar, follow these steps:

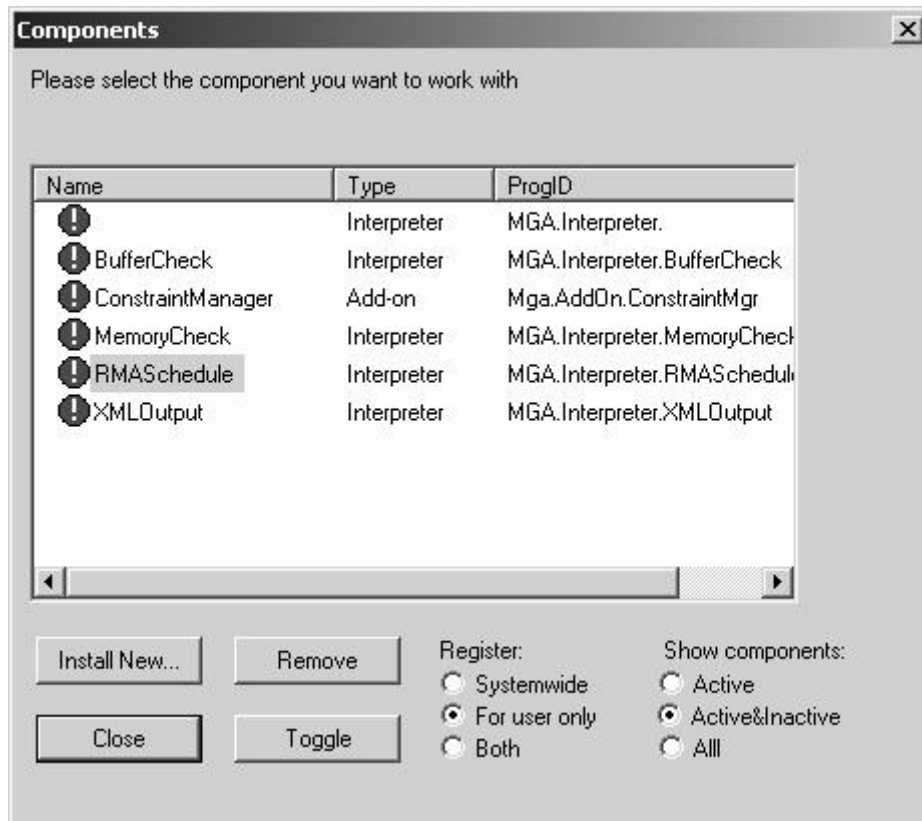
*Step 1:*

- Go to “Register Components” under the File menu
- This will bring up the Components dialog as shown in Figure 3-10
  - This lists all the currently registered components

*Step 2:*

- If the missing interpreter is listed

- Select the interpreter that is missing
- Click “Toggle”
- The green icon to the left of the interpreter name should contain an exclamation mark at this point
- Click “Close”
- The icon should now be visible on the toolbar



**Figure 3-10**

*Step 3:*

- If the interpreter is not listed in the Components dialog
  - Click “Install New”
  - This will bring up a file selector dialog box
  - Select the DLL from the sub-directory of the missing interpreter under [Installation Directory]/VEST
  - Click “Open”
    - The XML Output DLL may not show even after this step. To fix this problem, follow these steps
      - Click on the “All” radio button under “Show Components”
      - You should see an grayed-out entry for XML Output
      - Select this entry and click the “Toggle” button
  - Click “Close” on the Components dialog box
  - The icon for that interpreter should now be visible on the toolbar

## 4 The VEST Tool

### 4.1 Composition Environment

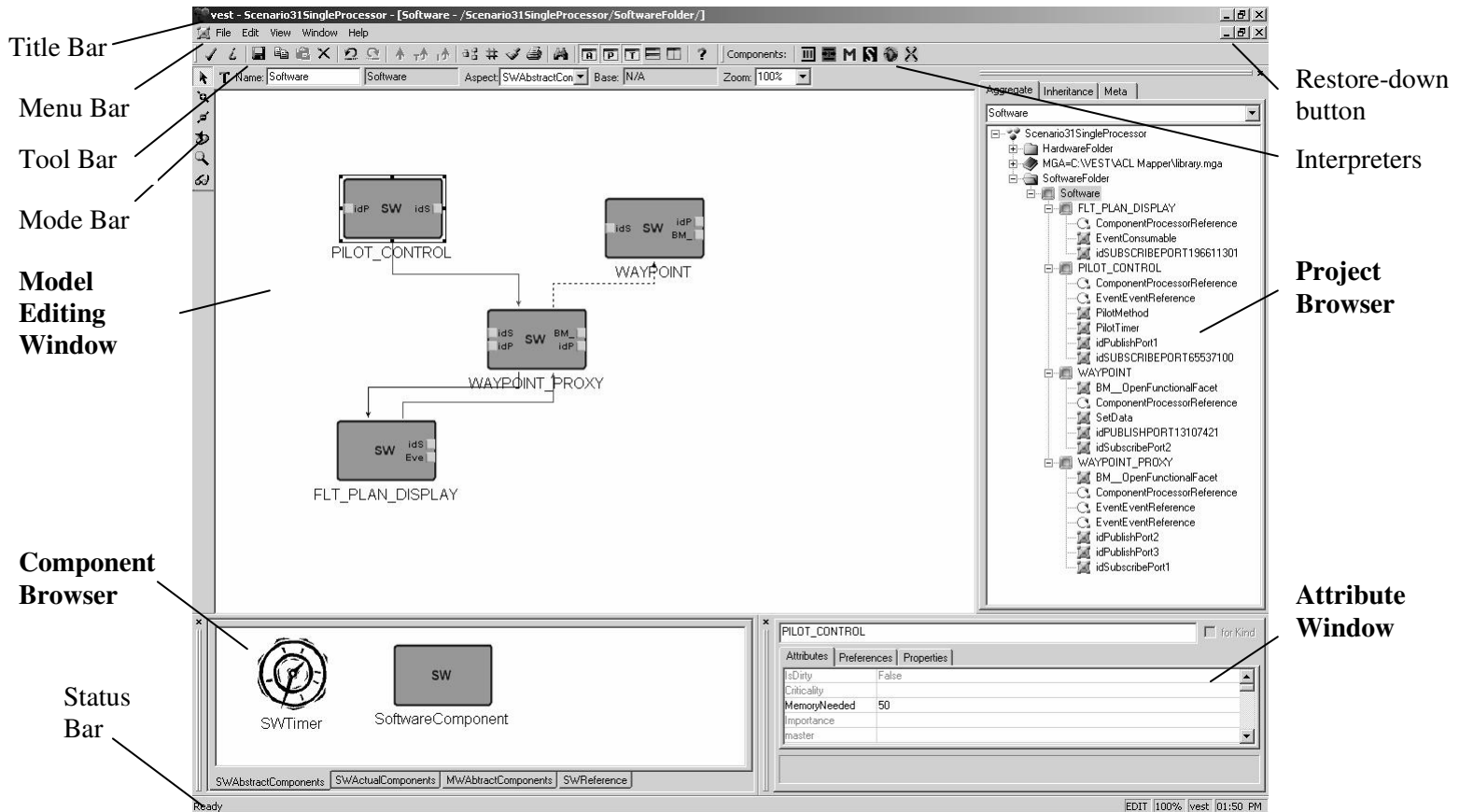


Figure 4-1

The composition environment of VEST is shown in Figure 4-1. It consists of 4 main panels as shown by the bold titles of the figure and other parts that are mostly GME specific. Each of these parts of the tool is described below:

#### 4.1.1 Model Editing Window

It is the main canvas of the tool and contains the product under development. For example, Figure 4-1 shows a product with four actual components (PILOT\_CONTROL, WAYPOINT\_PROXY, WAYPOINT and FLT\_PLAN\_DISPLAY) in the Model Editing Window.

Models in VEST can be hierarchical. This means that a certain component can be composed of sub-components. Components displayed on the same canvas are on the same hierarchical level of the design. The sub-components of a component can be viewed by double-clicking on the higher-level component. This opens up a new canvas

displaying the sub-components of the high-level component. For example, when we double-click the PILOT\_CONTROL component in Figure 4-1 above, we get a new canvas displaying its sub-components as shown in Figure 4-2.

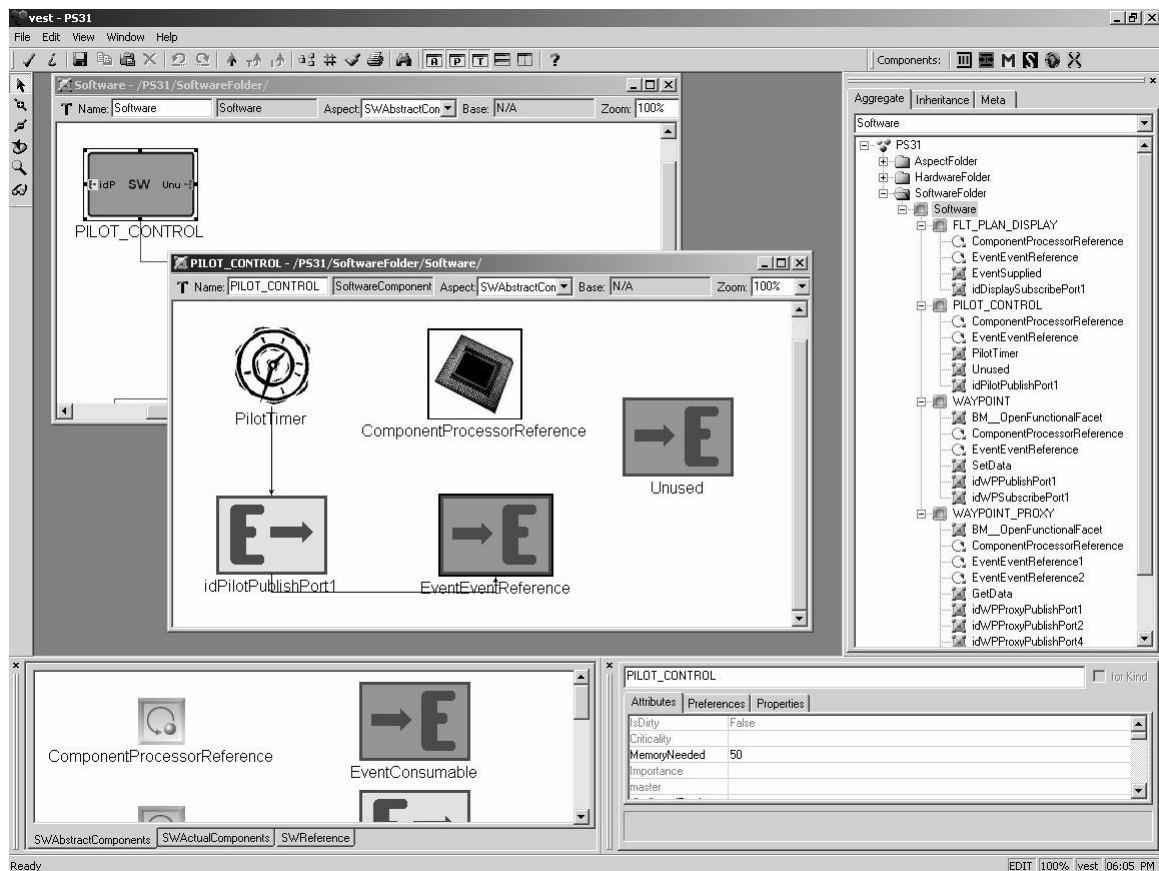


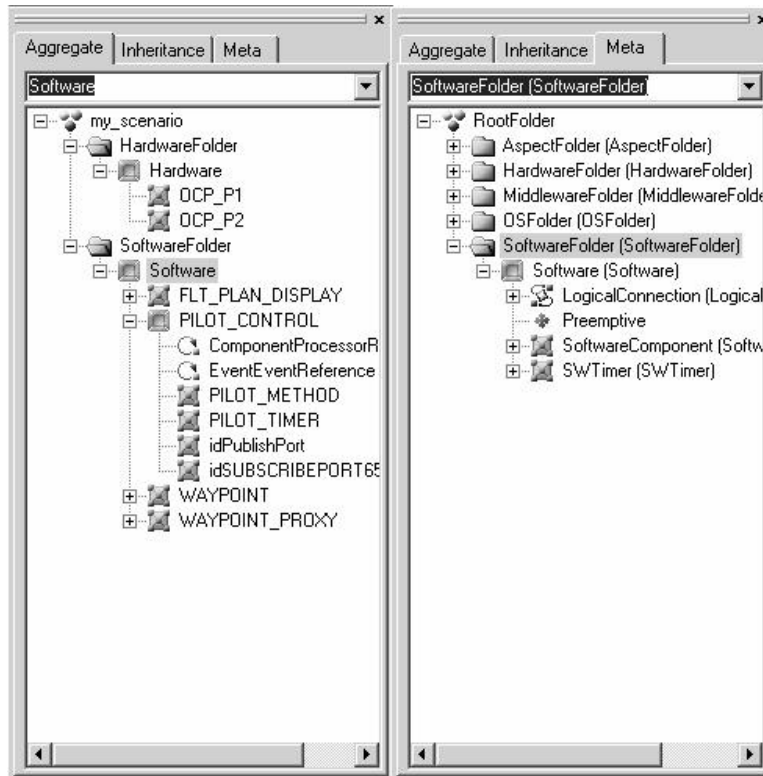
Figure 4-2

By default, all canvases open up maximized. By clicking on the restore-down button shown in Figure 4-1, all the current canvases that have been opened appear in the Model Editing Window as shown in Figure 4-2. Each sub-window has the name of the parent component in its title bar.

#### 4.1.2 Project Browser

The Project Browser has three different views as shown by the tabs in Figure 4-3. The *Aggregate* view displays the hierarchical structure of the product under development from the highest level of the project. For example, it can be seen in the figure that the Software layer has four components (PILOT\_CONTROL, WAYPOINT\_PROXY, WAYPOINT and FLT\_PLAN\_DISPLAY) and that the PILOT\_CONTROL component has six sub-components - 2 references (discussed later) and 4 software components.





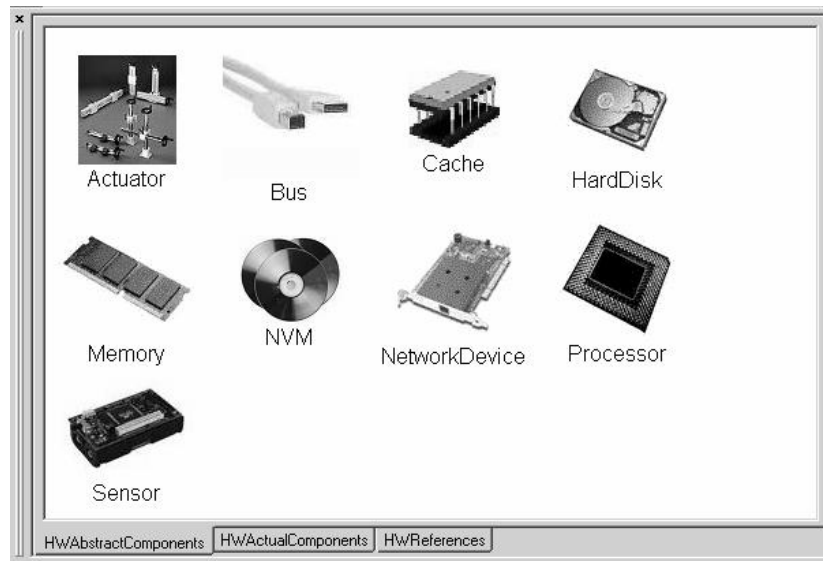
**Figure 4-3**

The Aggregate view also provides a convenient way of navigating between various levels of the product being designed. A user can open the canvas associated with some model layer or component by double-clicking on its name in the tree structure. For example, the PILOT\_CONTROL canvas was opened in Figure 4-2 by double clicking on its name in the aggregate view tree. When a particular level is chosen, all the available components for that level are displayed in the Component Browser. For example, Figure 4-2 shows the components that can be inserted into the PILOT\_CONTROL canvas. Finally, the aggregate view allows a user to rename components. Components can be renamed by clicking on the component's name and following the standard Windows file renaming procedure.

The *Inheritance* view is used to visualize the type inheritance hierarchy of the model currently selected in the Model Editing Window. Our example above doesn't have any inheritance. The *Meta* view shows the VEST modeling language at a glance. It displays the legally available array of Folders and objects that can be added to some level within the aggregate hierarchy. In VEST, as shown in Figure 4-3, at the highest level also known as the "RootFolder" level, we can add "Aspect", "Hardware", "Middleware", "OS" and "Software" Folders to a VEST project. Within each of these folders, we can add models. For example, in the Software folder, we can add a Software model. Within this model more parts can be added. For example, we can add Software Components and Software Timers to a Software model.

### 4.1.3 Component Browser

The Component Browser displays all the components available in a particular component library. In VEST, there are four component libraries: application, middleware, OS and hardware.



**Figure 4-4**

Figure 4-4 shows all the components available in the Hardware component library. Memory, Processor, Cache, Memory, etc are all the available hardware components that can be inserted into our design. A user can insert a component into a design by left-clicking on a component in the Component Browser and dragging it onto the design canvas. If a reference is dragged, a null reference is created because the target component is unspecified. References can be redirected at any time by dropping a new target on top of them (See Section 5.1.7).

There are two privilege levels with respect to library access. The first level is for users with only read access to the libraries. A system designer is an example of such a user. A system designer can use the predefined components in a library to build a system design but cannot modify the library in any way. The second level is for users with both read and write access to the libraries. A VEST library administrator is an example of such a user. A VEST library administrator can add components to the library.

### 4.1.4 Attribute Window

The Attribute Window displays all the attributes (reflective information) of a particular component when that component is highlighted. For example, the PILOT\_CONTROL component of Figure 4-1 is highlighted and its attributed are displayed in the Attribute Window. The component's name (which is also an attribute of the component) appears in a separate text box at the top of the attributes window. The name of the component can be changed here.

The Attributes Window has three tabs (see Figure 4-1) – one for the component’s attributes, one for the component’s preferences and another for the component’s properties. The attributes tab allows a user to set VEST specific attribute values for a component. The preference tab allows the user to define general preferences for a component such as color, icon name, etc. The properties tab lists constant properties of a component as defined by the VEST paradigm such as component type, kind, etc.

The common attributes of components include:

- WCET (worst case execution time)
- memory footprint
- data requirements
- interface assumptions
- importance
- initialization requirements
- environment requirements such as
  - must be able to disable interrupts
  - requires virtual memory
  - cannot block the code itself
  - preemption vs. non-preemption
- power requirements (if a hardware component)
- buffer rate

This list is illustrative and not comprehensive. It should give the reader a feel for the type of reflective information VEST stores for each component.

Whenever a component is added to a design, attributes that were designed with default values get initialized with those values. The other attributes have no values assigned to them. Default values appear grayed out in the Attribute Window. User-specified attribute values appear in black. Attributes (default and user-specified) can be modified in the Attribute Window. These values can come as single or multi-line text boxes, toggle buttons, combo boxes or color pickers. Values can be assigned to them using the standard Windows procedures for editing such data entry fields. Changes made to an attribute’s value are immediate.

#### **4.1.5 Menu Bar**

It contains commands for certain operations on the model. The menu bar in the VEST GUI environment contains the same options as the GME tool. Here is a brief overview of the commands available on the menu. For more information on any of these commands, please refer to the GME documentation.

##### **File:**

Contains the following project- and model-related commands:

- Create a new project
- Open, close, save or abort an existing project
- Import, export or update a project in XML format

- Print contents of active window
- Register paradigms and components
- Check or display model constraints
- Set GME-specific parameters
- Run model interpreters or plug-ins (paradigm-independent interpreters)
- Exit GME

#### **Edit:**

Contains the following editing commands

- Undo, redo up to last 10 operations
- Clear undo queue
- View/Edit current project properties
- Show parent, basetype or type of a model if applicable
- Copy, paste, delete, select all
- Cancel pending operation
- View/Edit current model preferences
- View/Edit current object's registry
- View/Edit synch aspects

#### **View:**

Allows the toggling on and off of the

- Toolbar, Status Bar
- Component Browser (called Part Browser in GME)
- Attribute Window (called Attribute Browser in GME)
- Project Browser (called Browser in GME)

#### **Window:**

- Cascade, Tile, Arrange Icons – standard Windows functions

#### **Help:**

- Contents – Shows online help contents GME
- Help – Shows context sensitive, user-specified help
- About – Standard Windows functionality

### **4.1.6 Tool Bar**



**Figure 4-5**

VEST interpreters

The toolbar contains icon shortcuts for certain editing functions that are available on the menu. VEST specific functionality appears in the *Components* section of the tool bar as shown in Figure 4-5. Located here are icons that implement interpreters. In our case, we have icons for aspect checks and prescriptive aspects. A developer can invoke an aspect check by clicking on a corresponding icon on the tool bar. For example, by clicking the

icon with an “S” in it in Figure 4-5, the user can invoke the real-time schedulability analysis aspect check. He can also apply a prescriptive aspect by invoking an aspect interpreter from an icon on the tool bar. More details about the checks available in VEST are in Section 4.2.

#### 4.1.7 Mode Bar

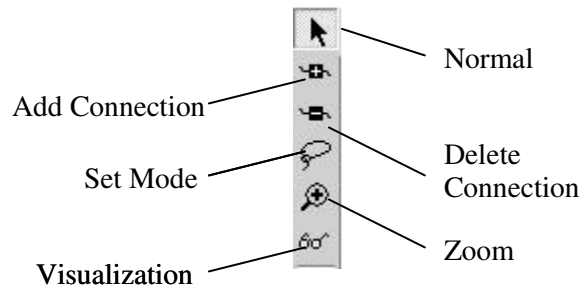


Figure 4-6

Contains buttons for selecting the editing mode. Each of the modes is described here briefly. Refer to the GME documentation for more detailed information.

Figure 4-6 indicates the buttons used to select the different editing modes. The editing modes are Normal, Add Connection, Delete Connection, Set Mode, Zoom Mode and Visualization.

**Normal Mode** - This mode is used to add, delete, move or copy parts within editing windows

- A component can be added by left-click-dragging the component onto a canvas from the component browser
- A component can be moved by left-click-dragging the component
  - This can be done across canvases also as long as the VEST modeling paradigm is not violated
- A component can be deleted or copied by right-clicking the component and choosing the appropriate option
  - A component can also be deleted also by selecting the component and hitting the “Delete” key on the keyboard
  - A component can also be copied by holding the CTRL-key down while left-click-dragging the component

**Add Connection Mode** – This mode allows connections to be made between components

- Connections are VEST-paradigm specific and will only be allowed between legal components
  - If you attempt to connect two incompatible components, GME will display an error message
- Connections are directional in nature

- Connections can be made by switching to add connection mode, left-clicking the source component and then left-clicking the destination component
- Connections can also be made in normal mode by right-clicking the source component, selecting “Connect” and then left-clicking the destination component

**Delete Connection Mode** – Existing connections between components can be removed in this mode

- Connections can be removed either by
  - left-clicking the connection
  - left-clicking the source/destination components

**Set Mode** – Can define sets and set members in this mode

**Zoom Mode** – Allows user to view a model at different levels of magnification

- Left-clicking anywhere in the model window zooms in and right-clicking zooms out

**Visualization Mode** – Allows single objects and collections of objects to be visually highlighted. This is useful for examining or analyzing large or complex models.

- Initially selecting this mode grays out all the components and connections
- Left-clicking any component toggles its visibility
  - Left-clicking a connection toggles the visibility of its end components
- Right-clicking a component will toggle the visibility of the component and the components at the ends of its connections

#### 4.1.8 Status Bar

The line at the bottom which shows status and error messages, current edit mode (e.g. EDIT, CONNECT, etc.), zoom factor, paradigm name (e.g. VEST), and current time.

#### 4.1.9 Title Bar

Indicates the currently loaded project.

### 4.2 Aspect Checks

One goal of VEST is to provide support for various types of dependency checking among components during the composition process. Dependency checks are invoked to establish certain properties of the composed system. This is a critical part of real-time embedded system design and implementation. Some dependency checks are simple and have been understood for a long time. We call these intra- and inter-component dependency checks. Other dependencies are very difficult and even insidious. We refer to these as crosscutting dependencies or *aspect checks*. Aspect checking is an explicit check across

components that exist in the current product configuration. We have identified many aspect checks that would help a developer avoid difficult to find errors when creating embedded systems from components. In many cases the important thing is identifying the check required and implementing it so that it is automatic. Although the implementation of some checks may be simple, when these checks are combined with all the other features of VEST, the result is a powerful tool. VEST currently supports four different aspect checks – buffer-size check, memory check, event check and an end-to-end real-time schedulability check. Each of these checks is described in the following sub-sections.

#### **4.2.1 Buffer-Size Check**

This check browses through a system and for each buffer, adds up its total buffer consumption based on the rate of execution of its consumers. Then it browses through the system, for each buffer, adds the total buffer production based on its rate of execution of its suppliers. The aspect check then compares each buffer's total production with its total consumption. If its total production is larger than total consumption, messages might be lost. The developer is informed. An illustration of the how to invoke and use the buffer-size check in VEST is given in Section 5.2.1.

#### **4.2.2 Memory Check**

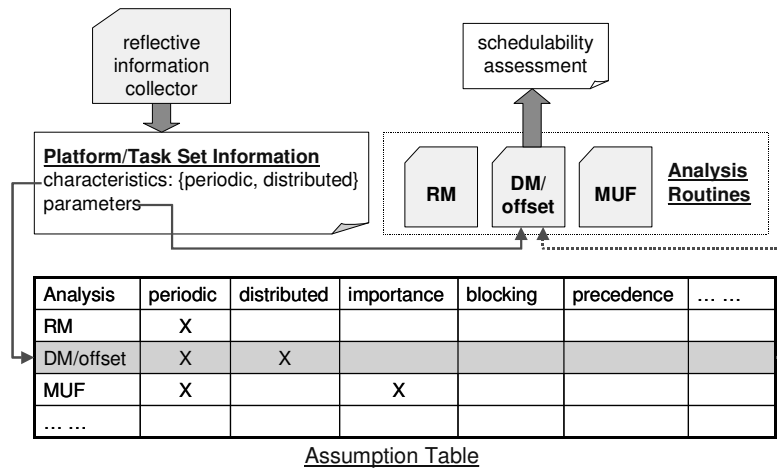
A memory footprint check is available in VEST. There are two parts to this check. The first part of the memory footprint check is concerned with main memory. It sums the memory needed by all the components in the system (including buffers since they are allocated from main memory), and all the available physical memory (RAM) provided by the hardware, and checks if there is enough physical memory in the system.

The second part of the memory check deals with NVRAM (e.g., EEPROM). For systems to function correctly, sufficient NVRAM for persistent components should be provided. Our check assures the developer that there is enough non-volatile memory to meet the system's requirement, or gives warning when not enough NVRAM is provided. An illustration of the memory footprint checks in VEST is given in Section 5.2.1.

#### **4.2.3 Event Check**

The event check is a simple check that iterates through all components and makes sure that every event supplier has an event consumer corresponding to it and every event consumer has an event supplier corresponding to it. Mismatches in the event channel are automatically identified. Also, circular event dependencies can be checked by going through the event channel. An illustration of how to use the event check is given in Section 5.2.3.

## 4.2.4 RT Scheduling



**Figure 4-7**

An important check for real-time embedded systems is the schedulability analysis that validates whether all tasks can make their deadlines. Note that while designing and implementing a system that most changes made will affect the real-time properties of the system. This makes real-time scheduling a global cross cutting dependency. While many different schedulability analysis techniques exist, they differ in their assumptions on the task set and none of the existing analysis is applicable to all real-time embedded systems. The compatibility between schedulability analyses and the characteristics of the designed system is a typical crosscutting dependency that is “hidden” from the designer. Using an incompatible analysis on a system can lead to timing violations even when the schedulability analysis itself is correct. To handle different types of embedded systems, VEST provides a flexible and extensible scheduling tool that provides aspect checks on the compatibility between existing schedulability analyses and the system. This tool (shown in Figure 4-7) is composed of a set of schedulability analysis routines, an assumption table, and a reflective information collector. The assumption table lists the assumptions of each schedulability analysis routine. The current list of assumptions includes:

- Periodic:** are all the tasks periodic?
- Distributed:** are any of the tasks distributed on multiple processors?
- Importance:** are important tasks protected in overload conditions?
- Blocking:** can low priority tasks block high priority tasks?
- Precedence:** are there precedence constraints among tasks?

For example, the assumptions of the Rate Monotonic analysis are that all tasks are periodic. The Rate Monotonic with Priority Ceiling protocol’s assumptions are (periodic, blocking). The VEST scheduling tool is extensible and new scheduling techniques can be added to the tool together with their assumptions. The current version of VEST can support the following schedulability analysis algorithms:

Single Processor:



- Rate Monotonic Analysis
- Maximum Urgency First
- Preemptive Threshold Scheduling
- Robust Scheduling
- Deadline Monotonic Scheduling with Phase Offset

Multi-Processor:

- Robust Scheduling
- Deadline Monotonic Scheduling with Phase Offset

Developers can assess the schedulability of the current design by running the scheduling tool from the GUI. The reflective information collector scans the software, hardware and network components of the design and produces a platform/task set information file that includes a list of the characteristics and the timing information of the task set. The tool selects an analysis whose assumptions match the characteristics of the system. This ensures that proper analysis and scheduling policy is applied. For example, for a system with all independent periodic tasks on a single processor, the checks listed under single processor above will be applied to the system. However, if the same task set is designed on a distributed platform, the Robust Scheduling or DM/Offset analysis will be applied.

Section 5.2.3 gives an illustration of the real-time schedulability analysis features available in VEST.

### 4.3 *Prescriptive Aspects*

Prescriptive aspects are *advice* that may be applied to a design. The advice is written in a simple VEST Prescriptive Aspect Language (VPAL). Prescriptive aspects are independent of programming languages because they apply to the system design, and the resultant new design can be implemented in any programming language. To change the system design, prescriptive aspects can adjust properties in the reflective information (e.g., change the priorities of a task or the replication levels of a software component). It can also add/delete components or interactions between components.

The new VPAL provided in this release has several enhancements over the previous version. First, the language explicitly separates the concerns of collection from operation. Collection is defined as gathering a set of components from a system design. Operation involves changing the design on previously gathered collections. Four key statements in the language, *Get*, *Set*, *Create* and *Delete*, enable this separation of concerns. The *Get* statement implements collection. Operation on collections are performed with the *Set*, *Create* and *Delete* statements. The *Set* statement adjusts the properties in the reflective information of the collection. The *Create* and *Delete* statements add and remove collections from the design. A second enhancement is the fact that collection can be based not only on the *properties* of the components but also on the *relationships* between the components.

Third, the new VPAL supports multi-line semantics. This means that each prescriptive aspect can contain multiple lines of instructions versus only a single line of instruction that was earlier supported. Third, the new VPAL supports grouping. A single, multi-line

prescriptive aspect is called a *simple* prescriptive aspect. Several simple prescriptive aspects can be grouped into a *compound* prescriptive aspect. Precedence constraints can also be specified among the elements of the group.

Suppose we want to apply the following prescriptive aspect to a distributed avionics system being designed in VEST:

```
change all display software components to use double buffering
```

Using the new VPAL, we could specify this prescriptive aspect as

```
GET SwComp = (CT == SoftwareComponent);  
GET DispComp = SWComp.(PN == componentType,  
                        PV == BM__DISPLAY_COMPONENT);  
SET DispComp.(PN == DoubleBuffered, PV = 1);
```

The prescriptive aspect interpreter will *get* all software components with type “BM\_\_DISPLAY\_COMPONENT” and *set* their double buffered flag to true. The complete BNF specification of the current version of VPAL is available in Appendix 4.

The developer applies a prescriptive aspect to a design by running a VPAL interpreter on its specification. The interpreter modifies the reflective information of design components. Since the code itself would no longer reflect the new design change, the interpreter marks the actual source code associated with that change as “*inconsistent and needing changes*” to meet the new design. Currently VEST does not support automatic code generation/modification, and the developer needs to implement the code change manually. Once the new code is created and linked to the component then the inconsistency indication is removed.

Prescriptive aspects should be general enough to be used in different products. VEST supports reusing prescriptive aspects by organizing them into the prescriptive aspect library. Prescriptive aspects will not be permitted into the prescriptive aspect library unless it meets with the approval of the system administrator. The requirements include sufficiently general, parameterized, complete English description, meaningful constraints specified, and relating to non-functional properties. The Prescriptive Aspect library will be implemented in a future version of VEST.

A description of how to use prescriptive aspects in VEST is given in Section 5.3.

## 5 Using VEST

### 5.1 *Building a System Design*

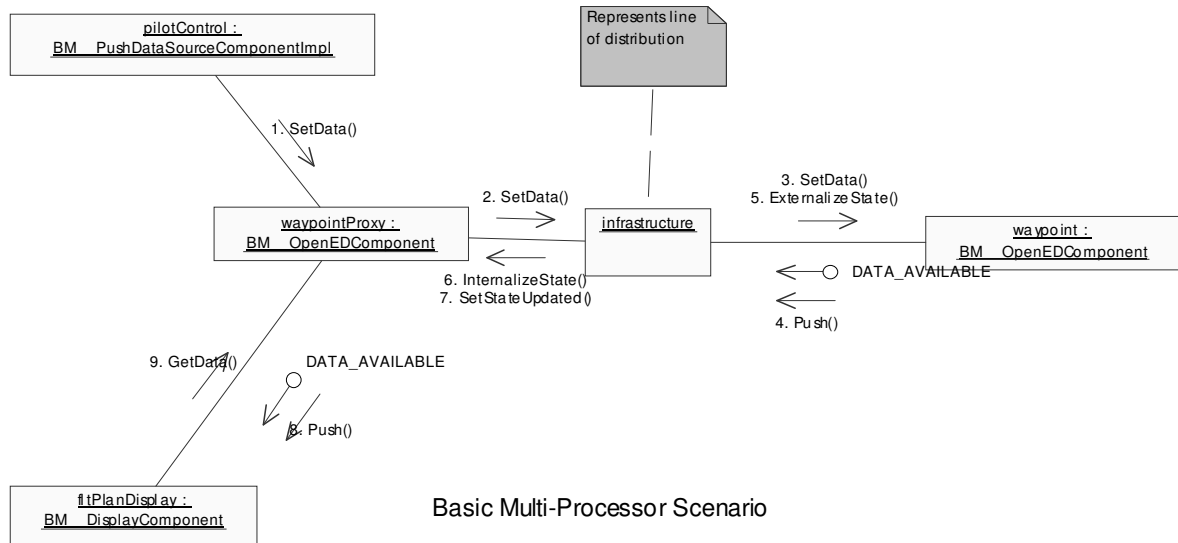
From the VEST GUI, a system developer can compose a distributed embedded system in the following way:

- 1) Design a product by choosing and combining components from the libraries.
- 2) Design the distributed systems hardware platform by choosing and combining components from the libraries.
- 3) Map software components to hardware and threads so that the active part of a composed system can be designed and analyzed. Only after this step can we truly do the real-time analysis since execution times are highly platform dependent.
- 4) Apply prescriptive aspects
- 5) Perform aspect checks and invoke (internal and off-the-shelf) analysis tools to analyze a configured system. If some checks fail, the developer may need to reconfigure or replace the actual components and repeat the checks.

#### **Exercise (Time Required: 1-2 hours)**

We present a simple tutorial that will take you through the steps outlined above in creating a simple system design. Following subsections will describe the various types of checks that we can perform on our design using VEST once it has been created. The system that we will design in this tutorial will be based on Boeing's Boldstroke middleware OEP. We will use Product Scenario 3.1 as the example. Please refer to [3] for more detailed information about the OEP platform and the product scenario. Note that the exercise may seem a little time-consuming and tedious at times. However if done properly, it will give the reader an excellent overview of the VEST tool and its capabilities.

This scenario is outlined in Figure 5-1. The scenario is initially triggered by an interval timeout that is consumed by the PILOT\_CONTROL. Upon receipt of this event, the PILOT\_CONTROL pushes data to the WAYPOINT\_PROXY via the Set operations in the proxy's facet. The WAYPOINT\_PROXY then forwards this call via the Infrastructure to the WAYPOINT. The WAYPOINT then updates its state and issues a Data Available event. That event causes the Replication Service to extract the state from the WAYPOINT and send it to the WAYPOINT\_PROXY. The WAYPOINT\_PROXY internalizes this state and issues its own Data Available event. The proxy's event is consumed by the FLT\_PLAN\_DISPLAY component that gets the data from the proxy and displays it.



**Figure 5-1**

You may find it useful to read Appendix 5 which describes how to construct Prism systems in VEST before proceeding. This exercise will take you through the steps required to model this scenario in VEST. After the system design is completed, we will perform checks on the design for robustness and schedulability.

Required parts of the exercise that are left to the reader to perform are indicated by the following symbol



### 5.1.1 Creating a new VEST project

First, we need to create a new project in VEST. To create a VEST project, do the following:

- Launch GME
- Select “New Project” under the File menu
- A “Select Paradigm” dialog box should appear
  - The Vest paradigm should be one of the options listed
  - If the Vest paradigm is not in the list, you did not successfully register the paradigm
  - Go back to the *Installation* section of this document and install the Vest paradigm into your system
- Select the Vest paradigm
- Click on the “Create New...” button
  - We are creating a new project

- A dialog box with 3 options on how to store the project will appear. Leave the default selection and click “Next”
  - The default selection “Create Project File” will store our project data in a file
- A dialog box asking you to name your project file will appear
  - Name the project Scenario3-1
- A new VEST project is created at this point. The name of the project (Scenario3-1) should appear at the top of the Project Browser window as shown in Figure 5-2.

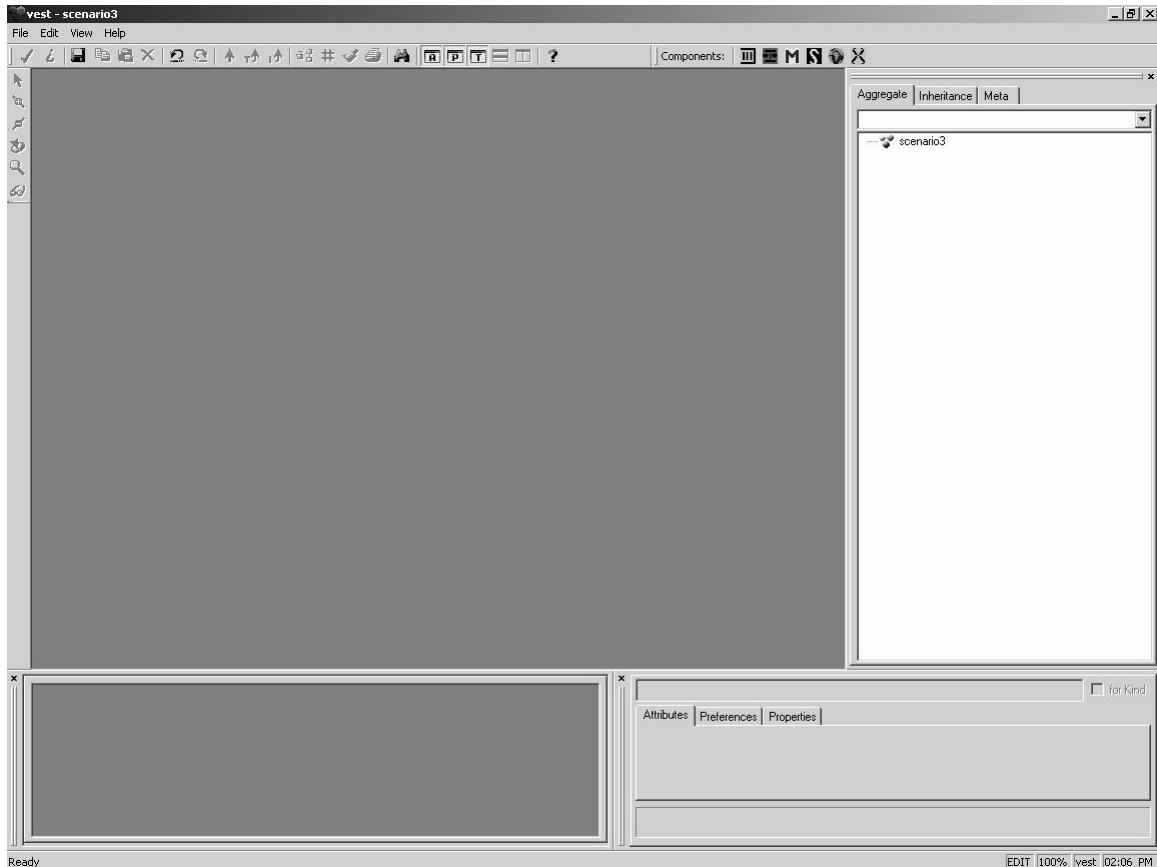


Figure 5-2

### 5.1.2 Creating a Model Layer

To design the software components of the system, we need to create a software model layer in VEST as follows:

- Right-click the project name (Scenario3.1) in the Project Browser window and select “Insert Folder -> Software Folder”.
  - VEST will create a new software layer folder and name it “NewSoftwareFolder” by default

- Rename this folder to “Software Folder” using standard Windows functionality
- Right-click on “Software Folder” and select “Insert Model -> Software”
  - VEST will create a new software model and name it “NewSoftware” by default
  - Double-click “NewSoftware”. Several things should happen.
    - § The Model Editing Window should display a blank canvas
    - § The Component Browser window should display two components namely “Software Component” and “SW Timer”
    - § The Attributes Windows should show the name of the model
- Change the name of the model to “Software” by editing the first text box in the Attributes Window



- Using this procedure, insert a Hardware model named “Hardware” into this project
- Follow Appendix 1 to attach the ACL library to your project. This will involve the following steps
  - Convert the Boeing ACL library into a VEST ACL library
  - Attach the library to your project in GME format
    - § This will add a new entry for the library under the project name in the project browser
    - § Expand this tree all the way down to the software level and you will eventually see the common components defined in the ACL in the Project Browser

### 5.1.3 Creating Components

Next, we will create the software and hardware components for the system. We will first create the software components of the system. Recall from the scenario description that the 4 software components are Pilot Control, Waypoint Proxy, Waypoint and Flight Plan Display. We will use predefined components from the ACL library to create the software components in our design. To add software components to this project, perform the following steps:

- Open the Software model
- Copy and paste the following components from the ACL library onto the Model Editing Window (henceforth, we shall simply call this the canvas)
  - 1 BM\_\_PUSH\_DATASRC\_COMPONENT
  - 2 BM\_\_OPEN\_ED\_COMPONENT
  - 1 BM\_\_DISPLAY\_COMPONENT
- Note that there may be several of these components that have corresponding components in the library with the same name except for upper-case characters. This comes from the Boeing ACL library itself. For the purposes of designing VEST systems, it does not matter which one you use. However, for consistency,

you may want to follow components that have names of the same format

- Rename the corresponding components in the above list to the following
  - PILOT\_CONTROL
  - WAYPOINT\_PROXY, WAYPOINT
  - FLT\_PLAN\_DISPLAY

This creates the four main software components of the design. We will perform other steps in the software layer but first, let's continue with defining the hardware components by following these steps:

- Open the Hardware model
- Click and drag a Processor component onto the canvas
- Name this component OCP\_P1



- Following the above steps, create the following components with the corresponding names
  - Processor OCP\_P2
  - Memory MEM\_P1 and MEM\_P2
  - NVM (Non-volatile Memory) NVM\_P1 and NVM\_P2

#### 5.1.4 Creating Relationships between Components


Next, we will define high-level relationships between the software and hardware component just created. From the scenario description, we can deduce the following relationships between our components:


- PILOT\_CONTROL contains WAYPOINT\_PROXY in its receptacle
- FLT\_PLAN\_DISPLAY contains WAYPOINT\_PROXY in its receptacle
- WAYPOINT\_PROXY invokes FLT\_PLAN\_DISPLAY by sending it a DATA\_AVAILABLE event
- WAYPOINT\_PROXY is a proxy of WAYPOINT


We can also define the following relationships in our HW layer

- OCP\_P1 has memory MEM\_P1 and NV-memory NVM\_P1
- OCP\_P2 has memory MEM\_P2 and NV-memory NVM\_P2

We can define these relationships in our model as follows:

- Switch to add connections mode by clicking on the following icon  on the mode bar. In this mode, you can add connections between components.
- Left-click on the PILOT\_CONTROL component and then left-click on the WAYPOINT\_PROXY component
  - This should create a connection between the two components

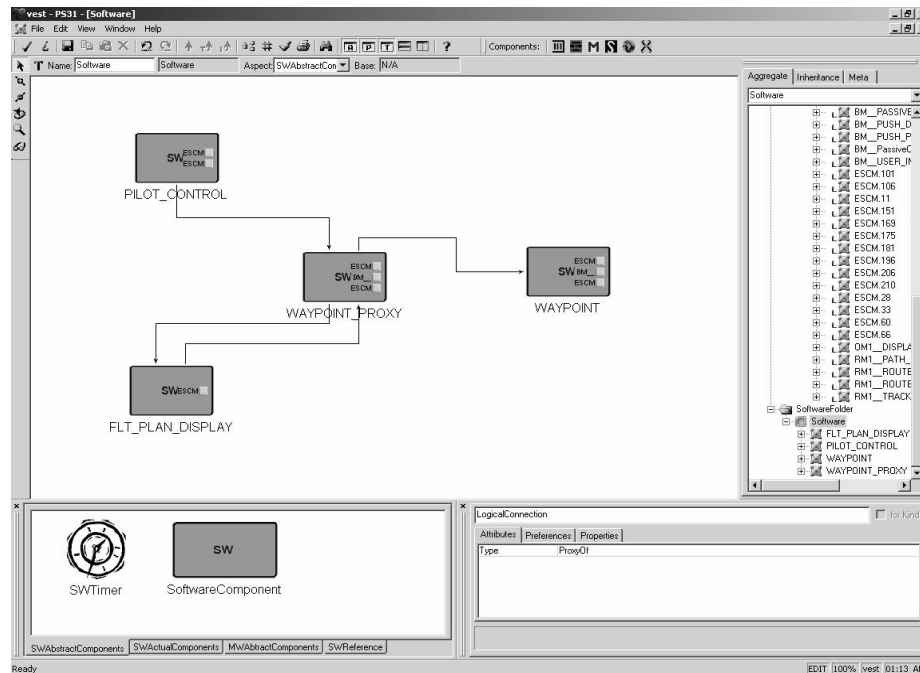
- To define the type of relationship, switch back to the select mode 
- Click on the connection and change the attribute “Type” to “ReceptacleContains”

If you make a mistake creating a connection, you can remove the connection in remove connection mode 



- Add the remaining connections defined above to the model

At this point, your software model should look like Figure 5-3 and your HW model should look like Figure 5-4.



**Figure 5-3**



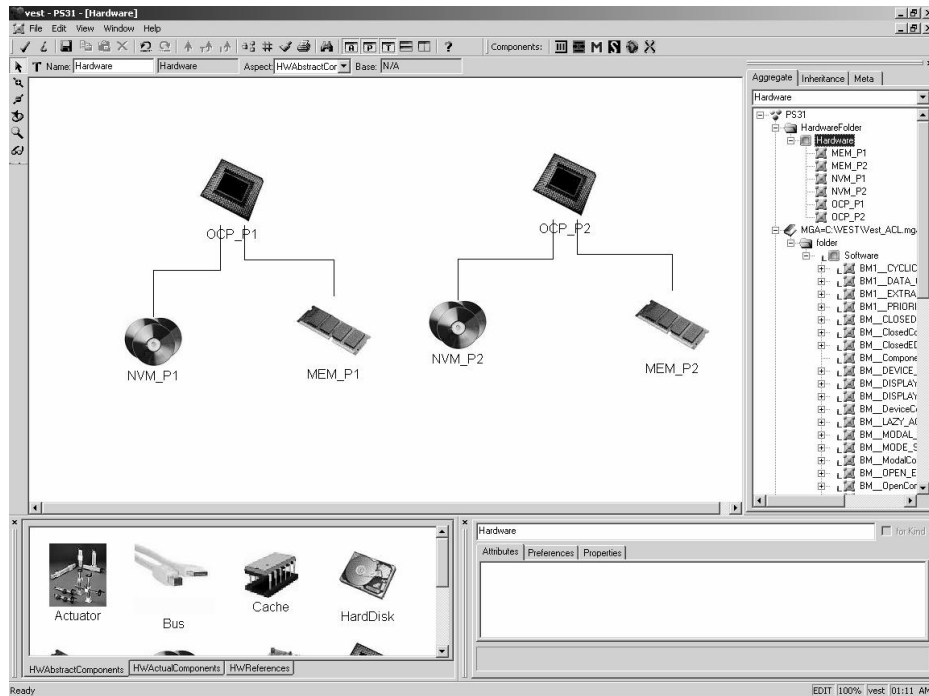


Figure 5-4

### 5.1.5 Setting Component Attribute Values

Next, we will set the attribute values of these components so that we can perform analyses on the system design. Follow these steps to define the Pilot Control software component:

- In the Software model, click on the PILOT\_CONTROL component
  - This selects the component
- In the Attributes window, set the values of the following attributes as indicated
  - PILOT\_CONTROL
    - § MemoryNeeded → 50
    - § NVMemoryNeeded → 25
    - § DoubleBuffered → True
    - § BufferSize → 10
    - § Relocatable → True



- As outlined above, define the remaining software components as indicated
  - WAYPOINT\_PROXY
    - § MemoryNeeded → 100
    - § NVMemoryNeeded → 150
    - § DoubleBuffered → True
    - § BufferSize → 20
    - § UpdateRate → 200
  - WAYPOINT

- § MemoryNeeded → 150
  - § NVMemoryNeeded → 100
  - § BufferSize → 10
  - § Relocatable → True
- FLT\_PLAN\_DISPLAY
  - § MemoryNeeded → 100
  - § NVMemoryNeeded → 100
  - § BufferSize → 10
  - § Relocatable → True
  - § UpdateRate → 50
- Switch to the Hardware model and make the following assignments
  - Memory\_P1
    - § Size → 300
  - Memory\_P2
    - § Size → 400
  - NVM\_P1
    - § Size → 350
  - NVM\_P2
    - § Size → 50

### 5.1.6 Creating Sub-Components

As mentioned in Appendix 5, all the real functionality of a software component such as its event channels is defined in a level below the highest model level. We can define the behavior of the PILOT\_CONTROL component by creating sub-components. To create sub-components of the Pilot Control component, follow these steps:

- Double-click on the Pilot Control component
- A new canvas will appear with the heading “PILOT\_CONTROL”
  - It should already contain 2 sub-components – 1 EventSupplied and 1 EventConsumable. This is because of the default definition of components of type BM\_PushDataSourceComponent
  - Rename the sub-components as follows
    - § EventSupplied → idPilotPublishPort1
    - § EventConsumable → idPilotSubscribePort1



Rename the following sub-components of the other components as indicated

- WAYPOINT\_PROXY
  - EventSupplied → idWPProxyPublishPort1
  - EventConsumable → idWPProxySubscribePort1
- WAYPOINT

- EventSupplied → idWPPublishPort1
- EventConsumable → idWPSubscribePort1
- FLT\_PLAN\_DISPLAY
  - EventConsumable → idDisplaySubscribePort1

In the next sub-section, we will create event channels but before we can do that, we need to learn how to create references in VEST.

### 5.1.7 Creating References

References allow a system designer to reference a component that is not on the current canvas. References are needed for the following reasons:

- Modeling large systems in a single canvas is sometimes infeasible and difficult to work with. References allow us to break up the system into smaller parts making the design easier to work with
- Making connections between sub-components of distributed components is only possible with references

We use references in this design for the second reason. To instantiate a reference in VEST in simple. Follow these steps to instantiate processor mappings for the software components:

- Double-click PILOT\_CONTROL in the Project Browser
  - This will open the PILOT\_CONTROL canvas
- Click and drag a ComponentProcessorReference component onto the canvas
- Restore-down this window
  - You may see several other open windows in the background. Close all of these windows
- Similarly, open the Hardware layer window
- Now, align the 2 windows so that OCP\_P1 and the ComponentProcessorReference do not overlap
- Click and drag the OCP\_P1 icon over the ComponentProcessorReference
  - The cursor icon should show a curved arrow when it is over the ComponentProcessorReference
- Release the mouse button
  - The ComponentProcessorReference icon should change into a Processor icon
- The reference is instantiated at this point



Using this technique, create processor mappings as indicated:

- WAYPOINT\_PROXY, FLT\_PLAN\_DISPLAY
  - OCP\_P1
- WAYPOINT

- OCP\_P2

### 5.1.8 Creating Event Channels

We will now define event channels in our design as specified in the product scenario by following these steps.

- Our event channel starts in the PILOT\_CONTROL component so open up that component
- Add the following components in the PILOT\_CONTROL component with the names and attributes indicated:
  - PilotTimer of type SWTimer
  - EventEventReference
- Set the following attribute values
  - PilotTimer
    - § Period -> 100
  - idPilotPublishPort1
    - § WCET → 9
- Next, connect the PilotTimer to idPilotPublishPort1 and this to the EventEventReference
- Redirect this reference to the idWPPProxySubscribePort1 component in the WAYPOINT\_PROXY component
- Simply stated, we have created a trigger for the SetData() call that is pushed from the PILOT\_CONTROL to the WAYPOINT\_PROXY component as described in this Product Scenario



Using the procedures described above, continue building this event channel following these steps:

- WAYPOINT\_PROXY
  - Create the following components
    - § idWPPProxyPublishPort2 of type EventSupplied
    - § idWPPProxyPublishPort3 of type EventSupplied
    - § idWPPProxySubscribePort2 of type EventConsumable
    - § idWPPProxySubscribePort3 of type EventConsumable
    - § EventEventReference1 of type EventEventReference
    - § EventEventReference2 of type EventEventReference
    - § GetData of type Method
  - Set the following attribute values
    - § idWPPProxyPublishPort1
      - WCET → 9
    - § idWPPProxyPublishPort2

- WCET → 10
  - MsgSize<sup>2</sup> → 5
- § idWPProxyPublishPort3
  - WCET → 8
- § GetData
  - WCET → 15
- Make the following connections
  - § idWPSubscribePort1 → idWPPublishPort1
  - § idWPProxyPublishPort1 → EventEventReference1
  - § idWPSubscribePort2 → idWPPublishPort2
  - § idWPProxyPublishPort2 → EventEventReference2
  - § idWPSubscribePort3 → idWPPublishPort3
  - § idWPProxyPublishPort3 → GetData
- WAYPOINT
  - Create the following components
    - § idWPSubscribePort1 of type EventConsumable
    - § SetData<sup>3</sup> of type Method
    - § EventEventReference
  - Set the following attribute values
    - § idWPPublishPort1
      - WCET → 19 (Includes the WCET of SetData – 9 without)
    - § SetData
      - WCET → 5
  - Make the following connections
    - § idWPSubscribePort1 → idWPPublishPort1
    - § idWPPublishPort1 → EventEventReference
- FLT\_PLAN\_DISPLAY
  - Create the following components
    - § idDisplayPublishPort1 of type EventSupplied
    - § EventEventReference
  - Set the following attribute values
    - § idDisplayPublishPort1
      - WCET → 12
  - Make the following connections
    - § idDisplaySubscribePort1 → idDisplayPublishPort1
    - § idDisplayPublishPort1 → EventEventReference

Instantiate the listed references for each of the following components:

---

<sup>2</sup> Indicates the size of a message in bytes sent from this producer

<sup>3</sup> The SetData method component is not used in the scheduling routine as explained in Appendix 5. It is put here for reasons of clarity

WAYPOINT\_PROXY:

- EventEventReference1 to idWPSubscribePort1 in WAYPOINT
- EventEventReference2 to idDisplaySubscribePort1 in FLT\_PLAN\_DISPLAY

WAYPOINT

- EventEventReference to idWPSubscribePort2 in WAYPOINT\_PROXY

FLT\_PLAN\_DISPLAY

- EventEventReference to idWPSubscribePort3 in WAYPOINT\_PROXY

CONGRATULATIONS! Your system design is now complete!

## 5.2 Using Aspect Checks

Using the system design that we created in Section 5.1, we will now invoke aspect checks on the design.

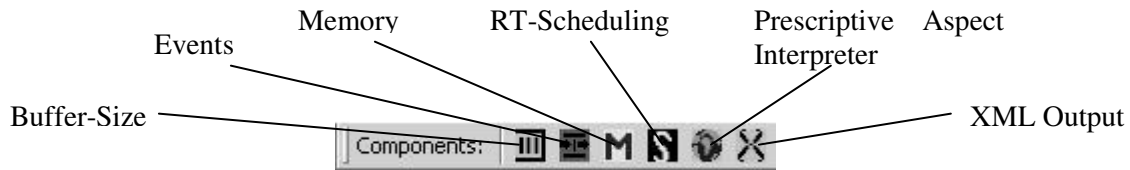


Figure 5-5

Figure 5-5 shows the icons for each of the checks on the VEST toolbar. The first check we will perform on our system design is the memory footprint check.

### 5.2.1 Memory Check

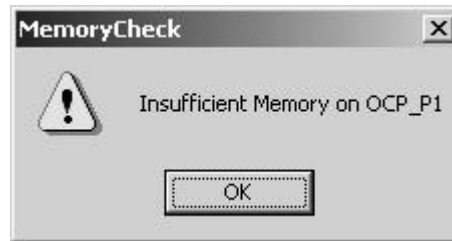
To recap, this is what we specified for the memory requirements (in bytes) of our software components and the available memory in hardware:

Software Component	MemoryNeeded	NVMemoryNeeded	DoubleBuffered	BufferSize
PILOT_CONTROL	50	25	True	10
WAYPOINT_PROXY	100	150	True	20
WAYPOINT	150	100	False	10
FLT_PLAN_DISPLAY	100	100	False	10

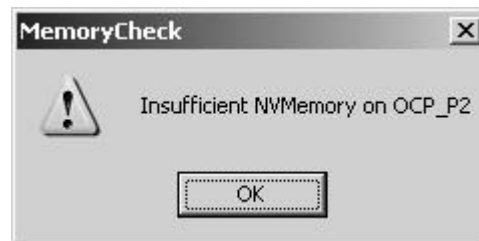
Hardware Component	Size
MEM_P1	300
NVM_P1	350
MEM_P2	400
NVM_P2	50

- To run the memory footprint, click on the Memory Check aspect icon
  - Based on the information we have entered, we need

- § 320 MB of memory and 275 MB of NV-memory for Processor 1
- § 160 MB of memory and 100 MB of NV-memory for Processor 2
- You should see the error messages shown in Figure 5-6 and Figure 5-7 because of the insufficient memory on processor OCP\_P1 and insufficient NVM on Processor OCP\_P2



**Figure 5-6**



**Figure 5-7**

- There are two ways to fix this problem
  - Increase the memory size on OCP\_P1
  - Reduce the memory requirements of the software components
- Let's say in this case that the system can only have 300 MB of memory for OCP\_P1. So we shall use the second option
  - We will make all buffers single-buffered
  - Change the DoubleBuffered attribute for PILOT\_CONTROL and WAYPOINT\_PROXY from True to False
  - Later we will see how we could have done these changes using a prescriptive aspect (Section 5.3)
- We will fix the second error by increasing the amount of non-volatile memory for OCP\_P2
  - Change the size of NVM\_P2 to 250 MB
- Re-run the memory check
- The memory check should successfully pass now

### 5.2.2 Buffer-size Check

For the buffer check, first verify that you have entered the following values for the attributes of the components listed below:

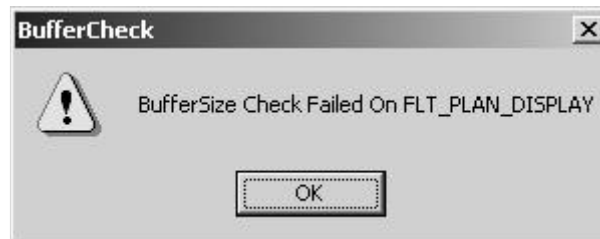
Software Component	BufferSize	UpdateRate
PILOT_CONTROL	10	0
WAYPOINT_PROXY	20	200

WAYPOINT	10	0
FLT_PLAN_DISPLAY	10	50

Sub-Component	Size of Msg in bytes
idWPProxyPublishPort2	5

Based on the scenario description, there is data flow between all components but for the sake of simplicity, we only define the data flow between WAYPOINT\_PROXY and FLT\_PLAN\_DISPLAY. The Waypoint Proxy component sends messages of size 5 bytes 200 times/sec. The Flight Plan Display component reads messages 50 times/sec. Between every read period of the Flight Plan Display component, the Waypoint Proxy has sends 4 messages. Therefore, the Flight Plan Display component needs a 20 byte buffer but it only has 10 bytes.

- To run the buffer-size check, click on the Buffer-Size aspect icon
- You will see some warnings that state that the UpdateRate attribute is not specified on the Pilot Control and Waypoint components
  - Ignore these and continue by clicking on “OK”
- The buffer-size check will fail and show the error message in Figure 5-8
  - This is for the reason specified above



**Figure 5-8**

- Update the buffer size of the Flight Plan component to 20 bytes
- Re-run the buffer size check
- The buffer-size check should pass this time
- Re-run the memory footprint check just to make sure that the changes you have made didn't affect anything
- The check should pass
  - If you had specified a larger buffer for the Flight Plan Display component, the memory check may have failed
  - You would have then had to update the attributes of the memory requirements of the system to make that check pass

When both the memory and buffer-size checks pass, you have the confidence that the system that you have designed has sufficient memory for its software components and that there will be no buffer overflows. This should illustrate to you the power and utility of the VEST tool.



### 5.2.3 Event Check

This is a very simple check. If you invoke the aspect check, it should give you a message for each event supplier that doesn't have a consumer and each consumer that doesn't have a supplier. In our design, you should see a message for a missing event supplier for a event consumer in PILOT\_CONTROL.

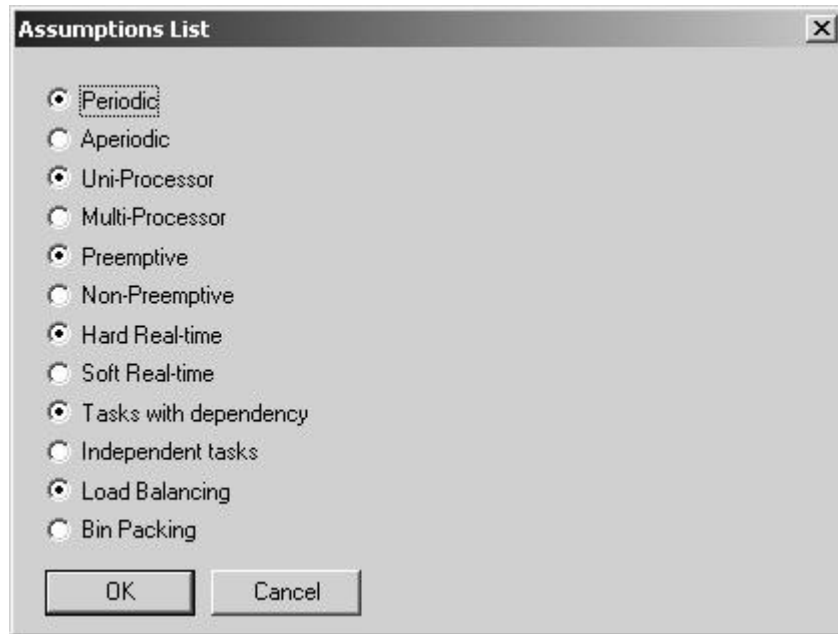
### 5.2.4 RT Scheduling Check

Here we present an example of how to perform schedulability checks on a system design. The schedulability of an algorithm depends on certain attributes of tasks in the system design. In VEST, a task can either be a Method or EventSupplied. The following are the task-level attributes that can affect the outcome of a schedulability analysis:

- Period
- WCET
- Offset
- Deadline
- Processor ID
- Relocatable

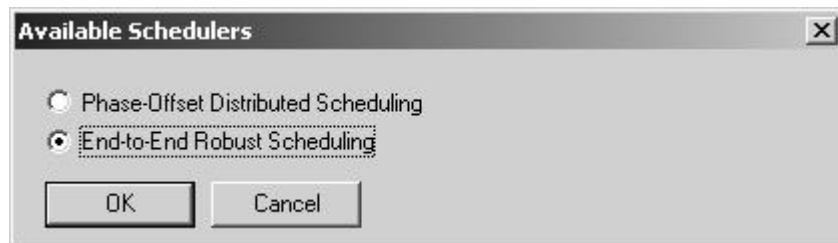
In the exercise that we perform, keep these attributes in mind. Follow these steps to schedulability analysis on your system design:

- Click on the RT-Scheduling check icon
- You should see the dialog in Figure 5-9
  - This is the Assumptions List of your design
  - This list represents the global-level attributes of your design
  - If this list accurately represents the assumptions you made about your design, then click "OK" to proceed
  - Otherwise, click "Cancel" to return to your design for modifications

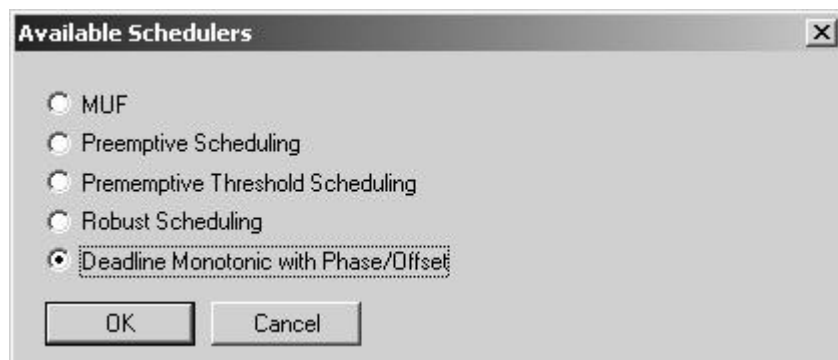


**Figure 5-9**

- Based on the Assumptions List, VEST will present the list of available scheduling algorithms that can be performed on your design. Since you have a multi-processor design, you should see the dialog in Figure 5-10
- If you had selected “Uni-processor” in your assumptions list, you would have gotten the available schedulers for multi-processor scenarios as shown in Figure 5-11



**Figure 5-10**



**Figure 5-11**

- Select “End-to-End Robust Scheduling” in the Available Schedulers list and click “OK”
- Based on the attribute values entered for the tasks in the system, our system design should pass the test and show the success dialog box in Figure 5-12. Click “OK”



**Figure 5-12**

- The output of the test is displayed next as shown below

```
Predefined cpu size: 2
Method Name PILOT_CONTROL.idPilotPublishPort1 Processor OCP_P1 Proc Num 0
Period 100 WCET 9
Method Name WAYPOINT_PROXY.idWPPProxyPublishPort1 Processor OCP_P1 Proc
Num 0 Period 100 WCET 9
Method Name WAYPOINT.idWPPublishPort1 Processor OCP_P2 Proc Num 1
Period 100 WCET 14
Method Name WAYPOINT_PROXY.idWPPProxyPublishPort2 Processor OCP_P1 Proc
Num 0 Period 100 WCET 10
Method Name FLT_PLAN_DISPLAY.idDisplayPublishPort1 Processor OCP_P1 Proc
Num 0 Period 100 WCET 12
Method Name WAYPOINT_PROXY.idWPPProxyPublishPort3 Processor OCP_P1 Proc
Num 0 Period 100 WCET 8
Method Name WAYPOINT_PROXY.GetData Processor OCP_P1 Proc Num 0
Period 100 WCET 15

Schedulability test on OCP_P2 passed.

Schedulability test on OCP_P1 passed.

The scaling factor is 1.061768 for robust end-to-end scheduling..
```

- We will now change an attribute in the system design so that this same test fails
- Select the “idPilotPublishPort1” component in PILOT\_CONTROL
  - Change its WCET from 200 to 400
- Re-run the steps outlined above

- Your scheduling test should fail because the tasks defined cannot be scheduled on a single processor. VEST should display the error message shown in Figure 5-13. Click “OK”
- You as the designer can try to change something in the design (eg. adding another processor) to make your test pass



Figure 5-13

### 5.3 Using Prescriptive Aspects

Here we present examples that illustrate how to create and execute prescriptive aspects in VEST that apply global advice to your system design. To run prescriptive aspects, you need to invoke the prescriptive aspect interpreter from the toolbar. This brings up the dialog shown in Figure 5-14. You have 2 options from here – the first to create/modify and execute a simple prescriptive aspect and the second to execute a compound prescriptive aspect. We will first look at creating and executing simple prescriptive aspects and later at compound prescriptive aspects.

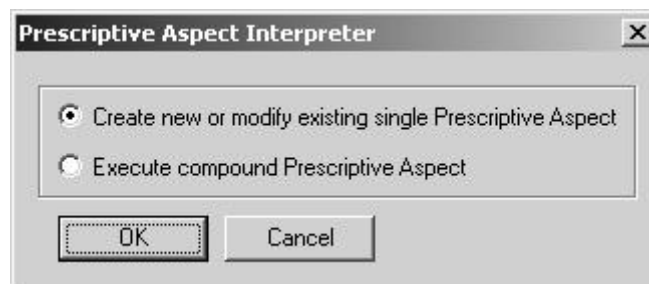
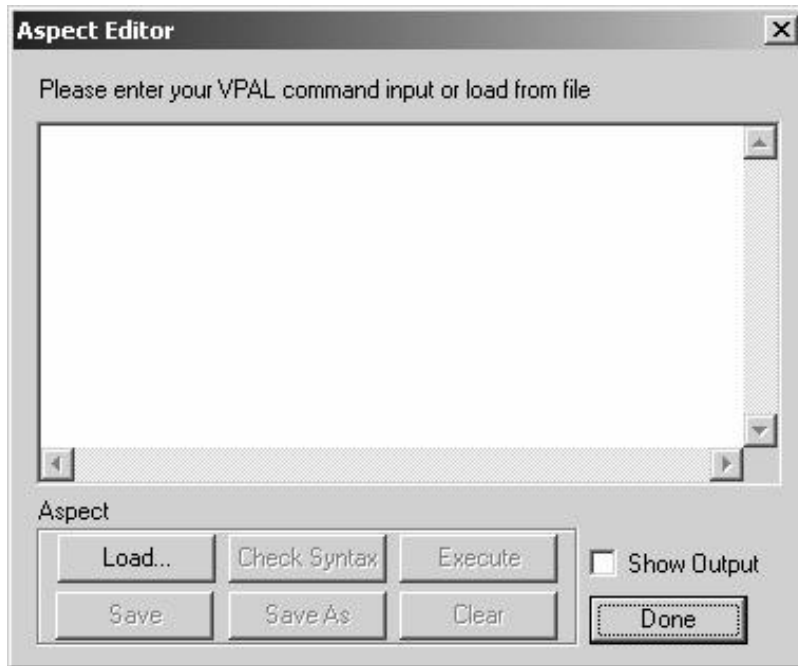


Figure 5-14

#### Simple Prescriptive Aspects

Suppose that as the designer, you decided to change the DoubleBuffered requirement of all software components in the system to true. Remember from the memory check exercise, we set this to false for all components. One solution would be to make the changes manually. Another method is to invoke a prescriptive aspect to do the work for you. We will create such a simple prescriptive aspect here to do precisely that.

- Select the first option in Figure 5-15 and click “OK”
  - This brings up the Aspect Editor shown in Figure 5-15



**Figure 5-15**

- The Aspect Editor is where you type in your prescriptive aspect commands
- There are several things you can do with the Aspect Editor outlined below
  - Enter Prescriptive Aspect input
    - § Manually by simply typing in input box
    - § From a file by selecting “Load...”
  - Check the syntax of the entered or loaded prescriptive aspect without executing the prescriptive aspect by selecting “Check Syntax”
  - Save the prescriptive aspect
    - § to a new file with “Save As”
    - § to the currently loaded file with “Save”
  - Clear the editor and start over with “Clear”
  - Execute the prescriptive aspect with “Execute”
    - § The “Show Output” checkbox (if checked) shows the contents of all sets defined in your prescriptive aspect when executed
  - Exit with “Done”
    - § Any unsaved work is lost!
- We will create a simple prescriptive aspect to change the double buffered assignment of all software components to true. Type in the following prescriptive aspect into the editor. Keep in mind that the prescriptive aspect parser is case-sensitive. You can find the complete BNF specification of the new VPAL language in Appendix 4

```
GET SWComp = (CT == SoftwareComponent);
SET SWComp.(PN = DoubleBuffered, PV = 1);
```

- The GET statement simply collects all components of type *SoftwareComponent* into set *SWComp*. The SET statement changes the property listed (DoubleBuffered) to the value listed (1 or true) for all elements in set *SWComp*.
- Check the syntax to ensure that your prescriptive aspect is syntactically correct
- Save the prescriptive aspect to file *my\_pa1.vpal*
- Execute the prescriptive aspect
  - If everything runs correctly, you should see the message box in Figure 5-16



**Figure 5-16**

- Check the DoubleBuffered property of all the software components in your design
  - They should all be set to “True” now



Using the procedure described above, create two more simple prescriptive aspects shown below and save them to the files listed. We will use these in create our compound prescriptive aspect in the next step.

*my\_pa2.vpal*

```
GET SWComp = (CT == SoftwareComponent);
GET DispComp = SWComp.(PN == componentType, PV == BM__DISPLAY_COMPONENT);
SET DispComp.(PN = MemoryNeeded, PV = PV + 50);
```

*my\_pa3.vpal*

```
GET SWComp = (CT == SoftwareComponent);
GET DispComp = SWComp.(PN == componentType, PV == BM__DISPLAY_COMPONENT);
SET DispComp.(PN = MemoryNeeded, PV = PV * 2);
```

Both of these aspects select the display software components from the design (FLT\_PLAN\_DISPLAY in our case) and make changes to their memory needed requirements. The first aspect adds 50 to it while the second doubles it.

### Compound Prescriptive Aspects

Compound prescriptive aspects are defined in the Aspect folder of a project. Each model layer within this folder represents one compound prescriptive. A component prescriptive aspect is composed of one or more simple prescriptive aspects called sub-aspects. Each

sub-aspect is associated with a file that contains the code for that simple prescriptive aspect. We will create a compound prescriptive aspect with the 2 sub-aspects that we created in the last exercise.

- Create an Aspect Model layer in your project called “AspectFolder”
- Insert an aspect model into this folder called “Memory Aspect”
- Open this model
- Click and drag two components of type *AspectComponent* into this model
  - An *AspectComponent* represents a single sub-aspect
- Name these two sub-aspects *MemAdd50* and *MemDouble*
- Connect *MemAdd50* to *MemDouble*
  - We specify a precedence constraint in our compound aspect this way
  - *MemAdd50* will execute before *MemDouble*
  - § The reverse order would produce different results!
- Now invoke the Prescriptive aspect interpreter and select option 2 of Figure 5-14
- This brings up the compound prescriptive aspect chooser as shown in Figure 5-17
  - Since we have only 1 compound prescriptive aspect define, we have only 1 entry listed

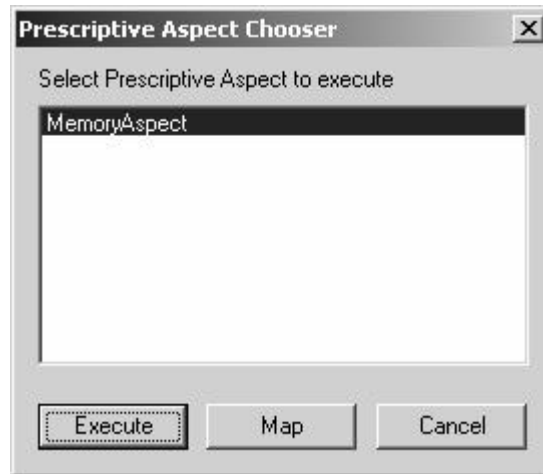
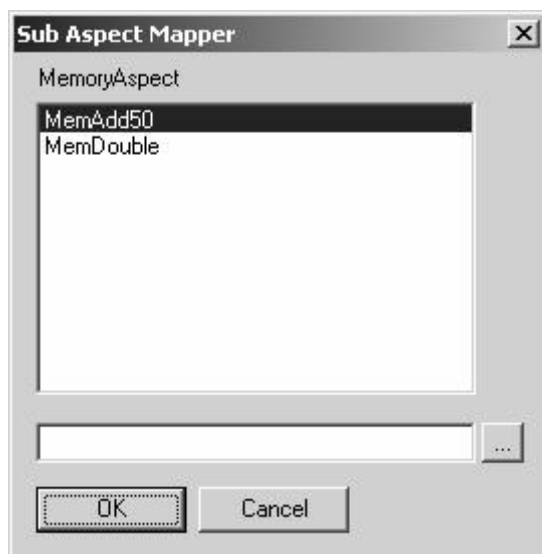


Figure 5-17

- Select “Map”
- This brings up a sub-aspect mapper shown in Figure 5-18
  - You map your sub-aspects to the files that contain their instructions here
  - Map *MemAdd50* to *my\_pa2.vpal* that you saved earlier
  - Map *MemDouble* to *my\_pa3.vpal*
  - Click “OK”



**Figure 5-18**

- Now that your compound aspect is mapped, it can now be executed
  - Click “OK” to execute the *Memory Aspect*
- Your aspect should execute successfully
  - FLT\_PLAN\_DISPLAY’s memory needed should be 300  $((100 + 50) * 2)$



## 6 References

- [1] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis, "VEST: An Aspect-Based Composition Tool for Real-Time Systems," *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, Washington DC, May 2003.
- [2] Akos Ledeczki, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle and Peter Volgyesi, "The Generic Modeling Environment," *Published in the Proceedings of WISP'2001*, Budapest, Hungary, May 2001.
- [3] "Product Scenario Description Document for the Weapon System Open Experimental Platform" *The Boeing Company, P.O. Box 516, St. Louis, MO 63166*
- [4] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis, "VEST: An Aspect-Based Real-Time Composition Tool," *Technical Report*, Department of Computer Science, University of Virginia, June 2003.

## 7 Appendix 1: ACL to VEST mapping

VEST includes a mapping tool that converts the Boeing Boldstroke middleware ACL components to GME-readable components. These ACL components can then be used as VEST components in VEST projects. The ACL to VEST mapper is available at [Installation Directory]/ACLMapper. The ACL mapper is written in Java and can be invoked by the following command

```
java ACLMapper3 <input_file> <output_file>
```

The input file is the ACL library file. The ACL library is stored as an XML file. Included in the ACLMapper directory is a copy of this library named *OEP.xml*.

When you produce the output VEST library XML file, it can be used by inserting it into a VEST project. There are two ways to doing this

- Attaching the library in GME format
- Importing the library in XML format

You can convert the XML library file outputted from the mapper into GME project format. The steps required to create a GME-format library are the same as those required to create a VEST project from XML format described in Section 3.2.4.1.

### Attaching a library in GME format

To attach a GME-format library to your VEST project, follow these steps:

- Right-click on the project name in the Project Browser window
- Select “Attach Library...” from the menu
  - The dialog box in Figure 7-1 will appear

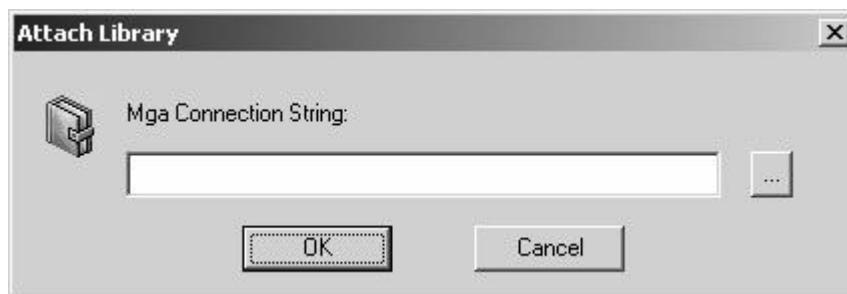


Figure 7-1

- If you know the location of the GME library, then type it in and click “OK”
- If not, you can browse for it by clicking the button with “...”

### Importing the library in XML format

To attach a XML format library to your VEST project, follow these steps:

- With your VEST project open, select “Import XML...” from the File menu
  - A File Selector dialog box will appear
- Select the XML file that contains the ACL library in VEST format
- Click “Open”
  - GME will read in the XML data and show a success dialog box
- The root of the project will change to that of the library name

To add the library components to your VEST project, use the standard GME click-and-drag operations.

## 8 Appendix 2: VEST to XML configuration mapping

VEST includes a mapper that converts the a VEST system design into a Prism XML configuration file. The XML configuration file can then be used to execute the system design on the Boeing Boldstroke OEP platform. The XML file is generated by an interpreter that can be invoked from the toolbar.

Running the XML configuration mapper is very simple. Follow these steps:

- Click on the XML Output interpreter icon
  - A File Selector dialog box will appear
- Select the location where you would like to store the output file
- Click “OK”

Once your output file is generated, you can view its contents using an XML compatible editor or browser.

If you have access to the Boeing OEP, you can use the XML file to build and decipher the system.

## 9 Appendix 3: RT Scheduling API

To allow third-party developers incorporate their own scheduling algorithms into the VEST tool, VEST 3.0 provides a RT Scheduling API. There are many unnecessary low-level details in the GME API. With the help of a high-level interface, third-party developers only need to worry about these high-level semantics for scheduling.

Whenever schedulability analysis is performed on a system design, VEST sweeps the model and generates a calling-graph. The *calling-graph* is a data structure that represents the relationship of the tasks in the system design. Third-party developers can incorporate their scheduling algorithms into VEST by manipulating this calling-graph. The high-level VEST scheduling API provides an interface to the calling-graph.

A *task* is defined as an execution unit. It has properties of period, execution time, start time (offset) and deadline. A collection of tasks is called a *task set*. The reason for using task sets is that there could be multiple starting tasks and ending tasks. Also, it is possible for a parent task to have multiple children and for a child task to have multiple parents.

The calling-graph is represented by a class called *callinggraph* in VEST. It provides the following interface:

```
Vest_taskset get_calling_graph()
    Get the set of starting tasks
```

The initial construction of the calling graph for Prism systems takes place as follows. In a Prism system, every thread is triggered by a timer. The calling graph is constructed by following the path of the timer. The *callinggraph* class contains a task set which includes references to all the timers in the system.

A task is represented by a *task* class in VEST. A task set is represented by a STL set called *taskset*. The task class includes, in addition to the above mentioned properties of period, execution time, start time and deadline, two task sets called *children* and *parents*. These task sets reflect the precedence of execution among the tasks in the system. The task class provides the following interface:

- void set\_period(period)  
*set the period of the task*
- int get\_period()  
*get the period of the task*
- void set\_wcet(wcet)  
*set the wcet of the task*
- int get\_wcet()  
*get the wcet of the task*

- void set\_offset(offset)  
*set the offset of the task*
- int get\_offset()  
*get the offset of the task*
- void set\_deadline(deadline)  
*set the deadline of the task*
- int get\_deadline()  
*get the deadline of the task*
- void set\_processor(processor)  
*set the processor id of the task*
- int get\_processor()  
*get the processor id of the task*
- vest\_taskset get\_parent()  
*get the set of parent tasks*
- vest\_taskset get\_children()  
*get the set of children tasks*
- int set\_child(task)  
*set task passed in to be a child of THIS*
- int remove\_child(task)  
*remove the parent-child relationship between THIS and the task passed in*
- int create\_task()
- int remove\_task(task)

When the developer wants to design a new scheduling algorithm using VEST, he can first initiate the calling graph, which collects the necessary scheduling information from the model. Then, the developer uses the methods provided by the task class to visit the calling graph.

The implementation of calling graph is available in callinggraph.h, callinggraph.cpp. All scheduling algorithms (for example, rma.h, rma.cpp) in VEST have used this interface.

## 10 Appendix 4: VPAL BNF Specification

```

<statement_list>      =      <statement_list> <statement> ';'

<statement>           =      <get_statement>
                          |      <set_statement>
                          |      <create_statement>
                          |      <delete_statement>

<get_statement>       =      GET <Variable> '=' <get_expression>
                          |      GET <Variable> '=' <Variable> '.' '('
                              <get_expression> ')'

<set_statement>       =      SET <Variable> '.' '(' <set_expression> ')'

<create_statement>    =      CREATE <create_expression>

<delete_statement>    =      PDELETE <delete_expression>           // Unimplemented

<create_expression>   =      Variable '=' '(' FolderType ',' STRING ',' CT '='
                              STRING ',' CN '=' ComponentName ')'
                          |      Variable '=' Variable '[' Relation_Type ',' STRING
                              ']' Variable

<delete_expression>   =      <Variable>                               // Unimplemented
                          |      <Variable> '.' <Relation>           // Unimplemented

<set_expression>      =      '(' <set_expression> ')'
                          |      <ObjectType> '=' <Value> ',' <set_expression>
                          |      <ObjectType> '=' <Value>

<get_expression>     =      '(' <get_expression> ')'
                          |      <get_expression> AND <get_expression>
                          |      <get_expression> OR <get_expression>
                          |      NOT <get_expression>
                          |      <ObjectType> '==' <Value>
                          |      <ObjectType> '==' <Value> ','
                          |      <ObjectType> '==' <Value>
                          |      <Variable> <Relation> <Variable>

<Relation>           =      '[' <Function_Type> ',' <Mapping_Type> '='
                              <Relation_Type> ',' STRING ']'

<Mapping_Type>       =      $DR                               // Direct Relation
                          |      $IR                               // Indirect Relation

<Relation_Type>      =      $CONT                               // Containment
                          |      $REF                               // Reference

```

		\$CONN	// Connection
<Function_Type>	=	\$ONEONE	// One-to-one
		\$ONEMANY	// One-to-many
		\$MANYTOONE	// Many-to-one
		\$MANYMANY	// Many-to-many
<ObjectType>	=	CT	// Component type
		CN	// Component name
		PN	// Property Name
		PV	// Property Value
<Variable>	=	`&' STRING	// Return variable
		STRING	
<Value>	=	INTEGER	// Integers
		ComponentName	// For component names
		Path	// To accept paths
		StringSequence	// To accept sequences
		True   TRUE   true	// Boolean
		False   FALSE   false	// Boolean
		`*'`	// For wildcards
		<scalar_exp>	// Arithmetic expressions
<ComponentName>	=	STRING	
		STRING '.' INTEGER	
<Path>	=	STRING '/' Path	
		STRING	
StringSequence	=	STRING StringSequence	
		STRING	
<scalar_exp>	=	<scalar_exp> "+" <scalar_exp>	
		<scalar_exp> "-" <scalar_exp>	
		<scalar_exp> "*" <scalar_exp>	
		<scalar_exp> "/" <scalar_exp>	
		"-" <scalar_exp>	
		"+" <scalar_exp>	
		"(" <scalar_exp> ")"	
		<ObjectType>	
		INTEGER	



## 11 Appendix 5: Designing Prism Systems in VEST

Due to the incompatibility of the VEST metamodel with the Prism architecture, there are certain assumptions that a designer should make when designing Prism systems in VEST. In VEST, the software, hardware, OS and middleware layers of the system design are specified separately. This discussion focuses on the software and hardware layers.

### 11.1 *Hardware Layer*

The hardware layer has only one level of abstraction. Designing the hardware layer is fairly simple. Each of the components in this layer represents an actual hardware component. Connections between components represent a concrete relationship. For example, Figure 11-1 represents a hardware design in VEST with three components – Processor, Memory and Non-volatile Memory (NVM). The connections here imply that components M1 and NVM 1 are associated with Processor P1.

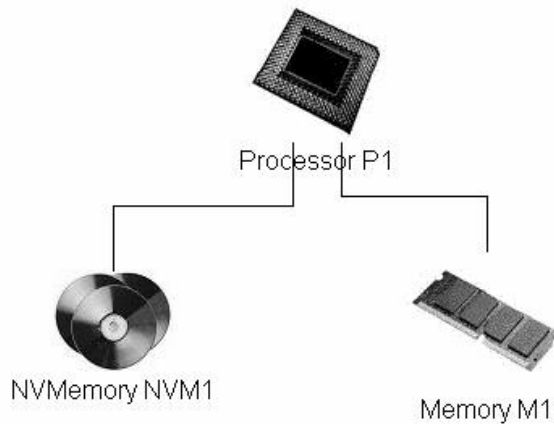
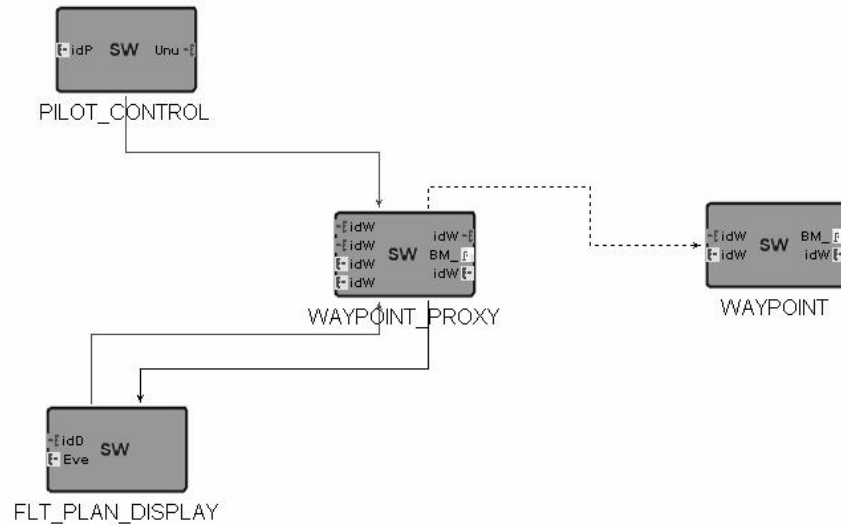


Figure 11-1

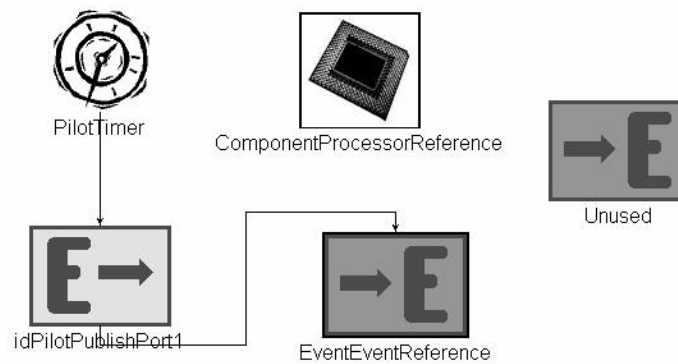
### 11.2 *Software Layer*

The software layer, on the other hand, has two levels of abstraction. The highest level (called Level 1 henceforth) of abstraction is for representing functional software components. The second level (Level 2) of abstraction defines the real functionality of the components in Level 1 such as event channels, whether the component is triggered by a timer, etc. Connections between components in Level 1 represent relationships between functional software components. Connections between components in Level 2 define the internal operation and characteristics of the component in Level 1. For example, Figure 11-2 shows a Level 1 software design that represents Product Scenario 3.1 [3]. The dashed line between WAYPOINT\_PROXY and WAYPOINT represents a *ProxyOf* relationship (Note that this is not automatic on the VEST GUI; it was manually done for demonstration purposes). Figure 11-3 shows a Level 2 design of the subcomponents of the PILOT\_CONTROL component of Level 1. This design specifies for PILOT\_CONTROL

its processor assignment, event channels and that it is timer triggered. A detailed discussion of how each of these is constructed in VEST follows. It is important that the software layer is represented as described here as all the analyses in VEST work on the basis of these assumptions.



**Figure 11-2**



**Figure 11-3**

Before proceeding with the detailed descriptions, it would be useful to familiarize yourself with Product Scenario 3.1. The discussion below assumes you have an understanding of this product scenario.

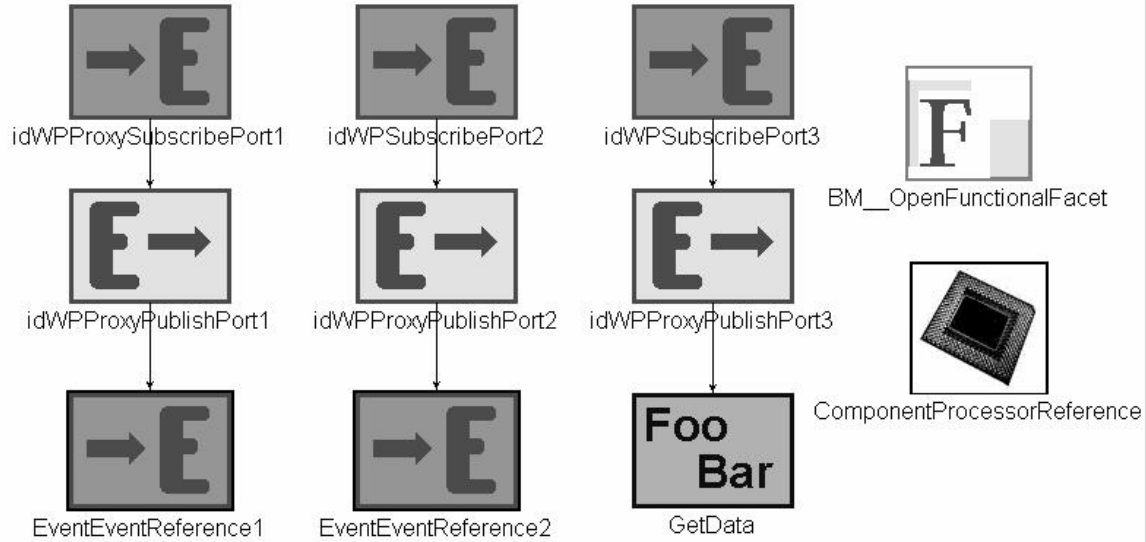


Figure 11-4

### 11.2.1 Event Channels

An event channel consists of a chain of event supplier-consumer-supplier links. An event channel is triggered by a timer and represents one thread of execution in the system. An event supplier in VEST is represented by the “EventSupplied” component in the software layer. Event consumers are represented by the “EventConsumable” component. Event suppliers and consumers within the same functional software component can be connected directly. However, if an event supplier and consumer pair lie in different software components, they need to be linked using an *EventEventReference*. The event channel and its associated methods constitute the task graph of a system. For example, in Figure 11-3, event supplier *idPilotPublishPort1* in the PILOT\_CONTROL component is connected to event supplier *idWPSSubscribePort1* in the WAYPOINT\_PROXY component (see Figure 11-4) through its connection to the *EventEventReference* component.

*EventConsumable* components either trigger other event supplier-consumer links or connect to a Method<sup>4</sup> component. For example, in the WAYPOINT\_PROXY component (see Figure 11-4), event consumer *idWPPProxySubscribePort1* triggers event supplier *idWPPProxyPublishPort1* which connects to an event consumer in WAYPOINT through an *EventEventReference*. In the same component, event consumer

<sup>4</sup> Only event suppliers at the end of an event thread can connect to a method. If a method needs to be invoked in the middle of an event channel, then its WCET should be added to the WCET of event supplier that invokes it. This is a temporary fix and the next version of VEST will allow methods in the middle of event channels

*idWPProxySubscribePort3* triggers event supplier *idWPProxyPublishPort4* which invokes method *GetData*.

### 11.2.2 Receptacles/Facets

A Receptacle relationship is specified in Level 1 of a software design. A connection of type *ReceptacleContains* between 2 software components implies that the first component contains the second component in its receptacle. For example, in Figure 11-1, the connections from PILOT\_CONTROL and FLT\_PLAN\_DISPLAY to WAYPOINT\_PROXY are of type *ReceptacleContains* and implies that both PILOT\_CONTROL and FLT\_PLAN\_DISPLAY contain WAYPOINT\_PROXY in their receptacles.

### 11.2.3 Processor Mappings

Mapping a software component to a processor is fairly simple. A Level 1 software component is mapped to a processor using a *ComponentProcessorReference* defined in Level 2 of that component. For example, Figure 11-3 contains a *ComponentProcessorReference* for the PILOT\_CONTROL component. Double-clicking on this reference takes you to the processor component that PILOT\_CONTROL is mapped to in the hardware layer.

### 11.2.4 Specifying Timeouts

Specifying a timeout interval for some event channel in a Prism design involves defining only one SWTimer component. The SWTimer component is defined in Level 2 of the software component that is triggered by a timer. The timer should be connected to an event supplier component that is the start of an event channel. For example, in Figure 11-3, timer *PilotTimer* is connected to event supplier *idPublishPort1* which is the start of an event channel in Product Scenario 3.1. The *period* attribute of the SWTimer determines the timeout interval for the event channel.