

Intelligent Machine Architecture 1.0 AgentBuilder Reference Manual

ROBERTO OLIVARES*

*Vanderbilt University, Department Of Electrical Engineering & Computer Science, Nashville, TN.

Intelligent Robotics Laboratory – Internal Technical Document, May 22nd, 2003.

1. INTRODUCTION

AgentBuilder is an application used to build *intelligent agents* from software objects called *components*. Initially, components are developed and compiled with Visual C++ into DLL form. AgentBuilder then provides developers with a user interface to select components, insert them into agents, initialize them, configure them, connect them to other components, then activate and debug them. In general, AgentBuilder should be thought of as a development tool designed for refining experimental approaches to robot control. More information on the underlying IMA system and IMA ideology can be found in the *IMA 1.0 Introduction & System Overview* manual.

2. USER INTERFACE OVERVIEW

In Figure 1, we see AgentBuilder running on Windows NT. The application's title bar displays the agent file currently being displayed. The agent's name on the component network is its file name without the *.GAS (Generic Agent Script)* extension. Beneath the title bar is the menu bar, which contains the *File, Edit, Managers, Agent, View, Project, Window, and Help* menus. Beneath the menu bar is the application toolbar, which contains various IMA-specific buttons.

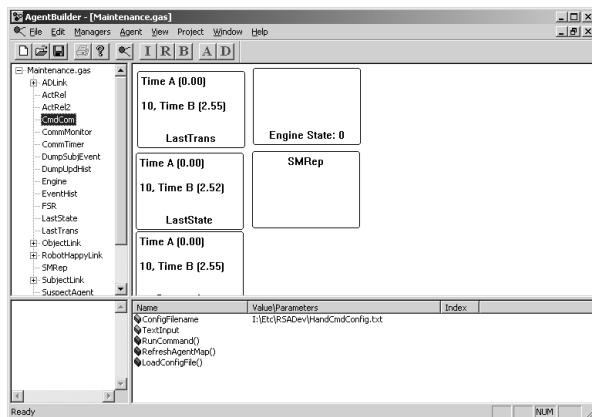


Figure 1. AgentBuilder running on Windows NT.

The left side of the AgentBuilder window contains the *Agent Tree View (ATV)* that displays the current agent and its child components. The right side of the window contains the *Manager View (MV)* that displays the currently connected managers. The bottom right panel contains the *Component View (CV)* which displays the default properties and methods for the component currently selected in

the ATV. The bottom left panel provides as a temporary notepad for developers.

3. MENUS

AgentBuilder contains six primary menus. Each of these menus contains items related to a specific aspect of agent management:

- **File** – Operations on the current agent file.
- **Edit** – Operations on the selected component.
- **Managers** – Operations on component managers.
- **Agent** – Operations on the Agent.
- **Window** – List of agents currently loaded.
- **Help** – Accesses help topics for AgentBuilder

4. TOOLBAR

The AgentBuilder toolbar contains eleven buttons (see Figure 1). The first five of these contain the following buttons (left to right): *New Agent File, Open Agent File, Save Agent File, Print, and Help*. The next six buttons are IMA-specific and relate to agent management (left to right): *Insert Component* (red node icon), *Initialize Agent, Register Agent, Bind Agent, Activate Agent, and Deactivate Agent*.

5. THE AGENT TREE VIEW

The ATV displays the child components in the current agent. When a new agent is created, the filename it is saved as is used as its agent name on the component network. When an agent file is opened, its filename is used as the agent name. Each component in the agent can be right-clicked to access a context menu containing various useful menu items from the *Edit* menu, including *Insert Component* and *Invoke Properties*. A component can be created in the agent by using either the context menu, *Edit → Insert Component*, or the insert component button. A component can be removed from an agent by selecting the component name and pushing the delete key, or by selecting *Delete Component* from the context menu. It is important to remember that many menu actions in AgentBuilder are performed on the component currently selected in the ATV.

6. CREATING A NEW AGENT

When AgentBuilder starts, it automatically creates a new agent for you. However, at any point additional new

agents can be created by selecting *File* → *New* or pushing the *New Agent* button on the application toolbar. The result will be a new child window with an empty agent tree view. It is often useful to immediately save the agent to a more descriptive filename, such as *CameraAgent*, instead of using the default name of *Agent1*. Components can then be inserted into the agent. Saving agents is covered in greater detail in Section 12.

7. INSERTING COMPONENTS

In AgentBuilder, a component can be created in an agent by using the agent’s context menu in the ATV, the *Edit* → *Insert Component* menu, or the *insert component button (ICB)* on the toolbar (see Figure 1). After selecting to insert a component, the *Insert Component Dialog (ICD)* is displayed on the screen (see Figure 2). This dialog can take a few moments to load, depending on the number of registered IMA components on the machine. The registered components are then displayed in the *IMA Components Listbox (ICL)* in the center of the dialog. If the *Preview Components* checkbox is enabled, the properties and methods of the currently highlighted IMA component are displayed in the bottom panel.

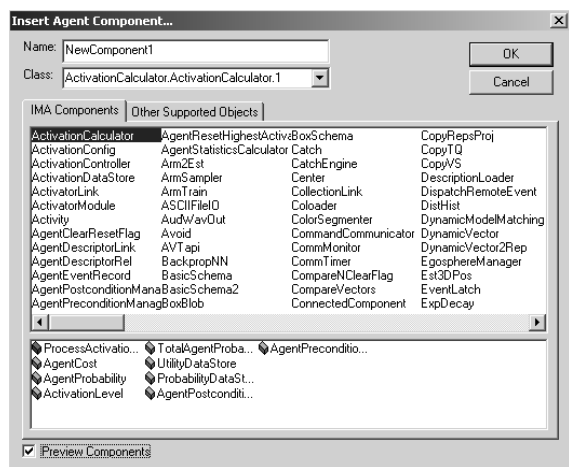


Figure 2. The Insert Component Dialog.

The class for the new component can be selected from the listbox or can be typed in to the *Class* textbox near the top of the dialog. This text corresponds to the *Program Identifier (PID)* of the IMA component you wish to create in the agent. Additionally, the name for the component can be specified in the *Name* textbox at the top of the dialog. When the name and class for the new component have been specified, the *OK* button creates the specified component in the agent.

In IMA 1.0, the PID for an IMA component is typically of the form *projectname.classname.1* and is specified by the developer as the “class name” when a new IMA class is generated in their IMA C++ project. An IMA component will not be shown in the ICD unless the component is properly registered by Visual Studio or by using *RegSvr32.exe*. Proper registration can be checked by searching through the registry for the class PID under “Automation Objects” in *OLEView32.exe*. Components

can be removed from an agent by first selecting the component, then selecting *Edit* → *Delete Component* from the application menu.

8. CONFIGURING COMPONENTS

Properties. Once a component is inserted, it is often useful to configure it by changing its properties or methods to new values. An arm controller could, for example, have its properties changed to change its pressure bias or to have it communicate with different agents. Viewing and changing properties can be accomplished through the component view or through the *Component Properties Dialog (CPD)*.

To change a property value in the component view, you must click the mouse on the value displayed for the property (denoted by teal blocks), enter a new value, and then push enter or click on another property value. The value can then be refreshed by clicking on the value. Alternatively, selecting *Edit* → *Component Properties* displays the CPD (see Figure 3) and allows a property name on the component to be specified. The existing value can then be retrieved via the *Fetch* button, or a new value can be entered and set via the *Apply* button. Both methods of viewing and changing properties work through the *IDispatch.Invoke(...)* functionality provided by COM. In order for properties to be accessible in this manner, they must be provided by the object’s default interface.

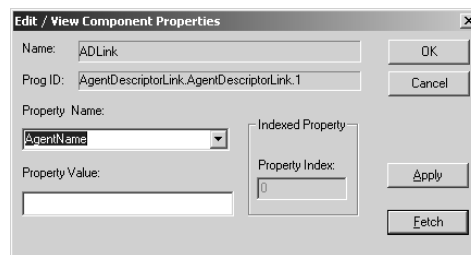


Figure 3. The Component Properties Dialog.

Methods. In addition to properties, methods on an object can be called by selecting *Edit* → *Invoke Components* from the application menu. This displays the *Invoke Method Dialog (IMD)*, which allows a method name to be selected and values for each parameter to be specified. The *Invoke* button can then be used to execute the function call. In order for the methods to be callable, they must be supplied by the component’s default interface. Calling methods using the IMD provides a convenient way of testing component functionality.

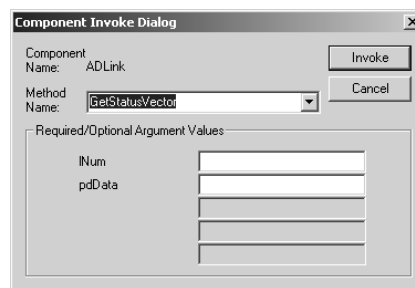


Figure 4. The Invoke Method dialog.

9. COMPONENT BINDINGS

Overview. An IMA binding is a named connection between two components. A component may *publish* multiple bindings (in the form of string properties) on its default interface. When one of these properties is changed to the path of an existing component on the network (i.e. *Agent1\Camera1*), the binding's *target* is said to be specified. At runtime, the path of a binding's target can be *resolved* by the AgentLocator into an object pointer. Once the pointer is obtained, methods and properties on the target can be called by the binding's publisher. In IMA 1.0, control agents contain bindings to representations (such as angular velocities) in the arm agent. These bindings allow them to communicate and transfer information.

Specifying Binding Targets. To change the target of a binding in AgentBuilder, the host component must first be selected in the ATV. Once selected, the properties for the component are listed in the component view. The developer must then select the text property that corresponds to the binding of interest (e.g. "*CameraPath*") and change its value to the path of an existing component (e.g. "*Agent1\Camera1*").

Registering & Binding Agents. Binding targets aren't resolved until the agent is *registered* and *bound*. Agent registration can be thought of as putting one's name and phone number in a phonebook. Agent binding can be thought of as looking up the phone numbers for a list of people. When an agent is registered, it enters the paths and pointers of its components into the AgentLocator database. When an agent is bound, each of its components queries the locator for a pointer to its binding targets. When those components are activated later on, their code accesses the resolved pointers.

More information on how to implement bindings can be found in the *IMA 1.0 Programmer's Guide*. To register and bind agents in AgentBuilder, the *Register* and *Bind* buttons on the toolbar can be used, or the equivalent commands can be selected from the agent menu.

10. INSERTING MANAGERS

Creating Managers. After components have been inserted into the agent, it is often useful to display a graphical representation of their data for debugging purposes. This task is performed by managers, objects written in Visual C++ that display information about a source component. Managers can display the numbers in a vector, the image coming from a camera, or the controls for a state machine. Many IMA classes already have managers written for them, such as the VectorSignal, TextQueue, and StateMachineEngine. Managers, like components, must first be developed, compiled, and registered to be useable from AgentBuilder.

To insert a manager into an agent, developers must first click on the Manager View panel in AgentBuilder. Selecting *Managers* → *Insert OLE Object...* from the application

menu will then display the *Insert Object Dialog* (see Figure 6).

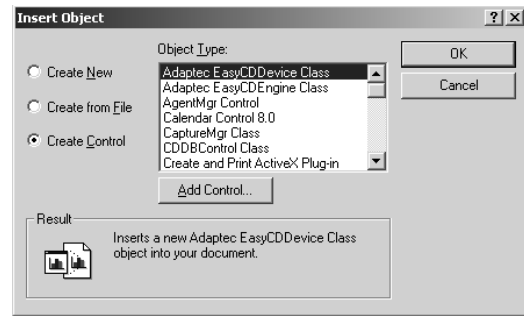


Figure 6. The Insert Object Dialog.

To display the list of available IMA managers, the *Create Control* option button must first be selected. Next, the manager's name must be selected from the *Object Type* listbox. In IMA, the manager for a component usually has the name of the component with "*Mgr*" appended. In Figure 6, we see the *CaptureMgr* class on the first page. Many of the objects listed in the dialog will not be IMA managers, so the selection must be performed carefully. After selecting a manager and pushing the *OK* button, two exceptions will be displayed. These are perfectly normal. Pushing ignore twice will then lead to the *Insert Manager Dialog* (see Figure 7).

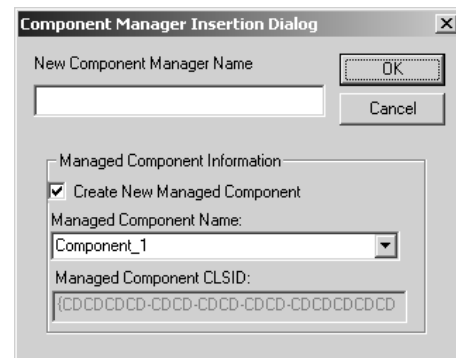


Figure 7. The Insert Manager Dialog.

At this point, a name for the manager should be typed into the *New Component Manager Name* field. If a source component for the manager already exists in the agent, then the *Create New Managed Component* checkbox should be unchecked and the name of the component should be entered into the *Managed Component Name* field. At this point, the *OK* button will configure the new manager. If the source component doesn't already exist, then the managed component's CLSID must be specified.

Resizing Managers. Once a manager has been inserted, it will appear in the Manager View. The size and position of the manager is prespecified, so it is useful to move and size the manager after it is created. Moving the manager is accomplished by clicking outside the manager so that it is not highlighted. Then, clicking and dragging the manager to a new position will move it. The manager cannot be moved while it has the focus (is highlighted). To

resize the manager, click the manager, then right click and drag a diagonal line outside the manager. The position and dimensions of the dragged line will be assigned to the manager. Due to a quirk in AgentBuilder, to see the moved manager will require clicking on a blank area of the Manager View.

11. ACTIVATING AGENTS

Overview. After components have been inserted into an agent, configured, and any relevant managers inserted, the agent is nearly ready to be activated. When an agent is activated, the engine components contained in the agent are told to begin their thread of execution, thereby making function calls on other components. Clearly, this process should not be started unless the components are properly initialized, configured, registered, and bound. For this reason, AgentBuilder provides the Initialize, Register, and Bind buttons on its application toolbar. After these have been performed, the agent can be activated or deactivated using the *A* and *D* toolbar buttons.

Initialization. After components have been inserted and configured, they should be initialized to allocate resources and reset their starting values. The *Initialize* toolbar button will cause the agent to call *IComponent.Initialize(...)* on each child component.

Registration. Before the bindings of an agent can be resolved into pointers, their bind targets must be registered in the AgentLocator. The *Register* toolbar button submits the path and pointer of each child component to the AgentLocator so that it can be located. If an agent is registered, and then new components are added, it must be registered again to reflect those new components. An agent that is not registered will not be accessible to other agents at bind time.

Binding. Once all agents on the network are registered, the process of binding them can occur. Pushing the *Bind* toolbar button will cause each component in the agent to attempt to resolve its bindings into a pointer. If the target components are not registered, this process will fail without any signal to the user. It is up to developers to verify that the binding process has been performed correctly, or the system will fail to function properly when it is activated.

Activation. Once agents have been built, initialized, configured, registered, and bound, they can be activated using the *Activate* toolbar button. The order in which components are activated matters, and it is up to the developer to make sure they are activated properly. If any required agents are not on the network, not registered, or improperly configured, errors may occur after activation.

Deactivation. Once an agent has been activated, it can be deactivated by pushing the *Deactivate* toolbar button. This will stop the thread of execution of all engines within the component. This can be used to effectively pause or stop an agent. Agents should be deactivated before they are closed.

12. LOADING AND SAVING AGENTS

Generic Agent Script Files. When *File* → *Open...* or *File* → *Save As...* are selected from the application menu, AgentBuilder will attempt to load or save information about the names and types of components found in the agent (see Figure 8). These AgentBuilder files also store relevant manager information. The filename of the saved .GAS file will become the name of the agent when it is opened. An agent loaded only from a .GAS file will have component properties set to their defaults.

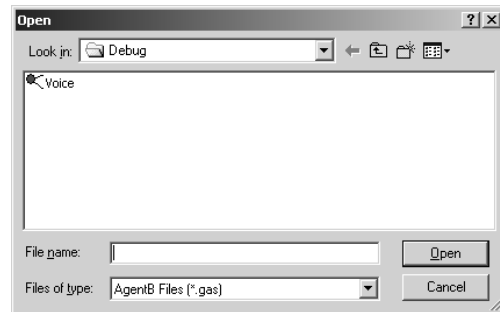


Figure 8. The Open AgentBuilder File Dialog.

Agent Files. Because the .GAS file format only saves the names and types of components in an agent, a second file is necessary to save the state of components. The state of a component includes the values of its properties and any recorded data. State data is stored in agent files with the .AGT extension. To make loading state data easy, selecting *Agent* → *Setup Agent File* from the application menu allows developers to specify which agent file will be used by default when the .GAS file is opened. Once the default agent file has been specified, selecting *Agent* → *Load Agent File* or *Agent* → *Save Agent File* from the menu will automatically load or save data to that file.

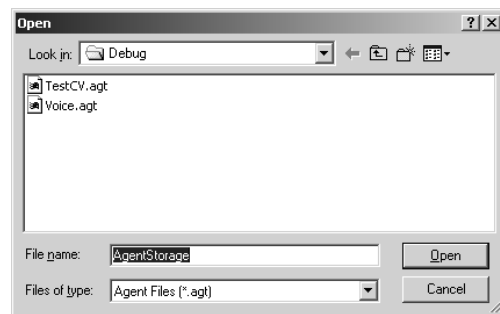


Figure 9. The Open Agent File Dialog.

Developers will typically want to create their agents, initialize them, configure them, and then save the agent file so their changes are recorded. Selecting *Agent* → *Import Storage* will load the information in a specified agent file without changing the default agent file specified in the .GAS archive.

13. WORKING WITH MULTIPLE AGENTS

AgentBuilder is a multiple-document interface (MDI) application built using Visual C++ and MFC. Its MDI interface style allows multiple agents to be hosted inside of one AgentBuilder window, each with its own screen. Agents within the main window can be cycled through by pressing CTRL+TAB, or selecting them from the Window menu.

14. REPRESENTATION PROXIES

Specifying Representation Proxies. Representation proxies are a special type of binding supported by representations. By selecting *Edit* → *Set Representation Proxy* from the application menu, a remote representation of the same type can be set to *cache* (mimic) the data of the selected representation. This is often done to provide a local copy (the *proxy*) of remote data (the *source*) for performance or reliability purposes. When a proxy is set for a representation, the target component will receive a data update whenever the source representation is changed. Because of this update process, proxies can slow down their respective agents if not used carefully.

To configure a representation as a proxy, it must first be selected in the ATV. Next, selecting *Edit* → *Set Representation Proxy* will display the *Proxy Setup Dialog (PSD)* shown in Figure 10. To configure the proxy, the *Set Proxy Mode* checkbox must be set and the source component's path must be specified in the *Remote Source Component Name* textbox. This component must be registered on the network to be accessible. Additionally, the path of the proxy itself should be specified in *Proxy Tag String* field. Any component-specific flags should be then entered in the *Proxy Subscription Flags* field and the *OK* button pressed. Each source component can support multiple proxies.

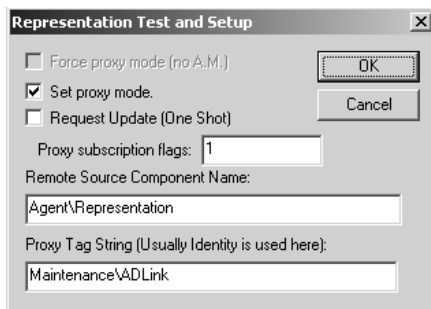


Figure 10. The Set Representation Proxy Dialog.

Once a proxy has been configured, it is useful to test the connection to make sure it is receiving updates from the source. This can be performed by adding managers to the source and proxy components, then changing the data in the source component and looking for an equivalent change in the proxy component. Alternatively, an update can be forced by selecting *Edit* → *Rep. Update Test* from the application menu.

APPENDIX A: FILE LOCATIONS

IMA 1.0 component and manager files (DLLs) should be stored in either the *C:\IMA* or *D:\IMA* directory. The AgentBuilder program itself is stored on the server, typically on the *I:* drive. If the *I:* drive is not mapped, it can be mapped by right-clicking on my computer and selecting *Map Network Drive*. The *I:* drive should be mapped to the *\\IRLSERVER\IMA* server share. The AgentBuilder program is stored in the *I:\AgentBuilder\Source\Debug* directory. The executable filename is *AgentBuilder.exe*.