

# Intelligent Machine Architecture 1.0

## Introduction & System Overview

ROBERTO OLIVARES\*

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

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

### 1. INTRODUCTION

Humanoid and mobile robots are just a few examples of intelligent machines that require sophisticated software to transform sensory information into purposeful actions. When writing this type of large-scale, complex software, developers benefit from domain-specific guidelines that promote code reuse and integration. The *Intelligent Machine Architecture (IMA)* was designed to provide these guidelines and is currently used to control the ISAC [Peters et al., 2000], Helpmate [Kawamura et al., 2000], and Scooter [Wilkes et al., 2002] robots at the Vanderbilt University Intelligent Robotics Laboratory. IMA was also developed to address *software integration* and *scalability* issues in intelligent machines [Bagchi et al., 1992]. Software integration is the process of combining software to extend the functionality of the system. Software scalability is a measure of how well a particular piece of software allows for future integration [Schach, 2002].

### 2. BACKGROUND

At the time of IMA's development, intelligent robotics research was being strongly influenced by three popular artificial intelligence (AI) approaches: *agent-based systems* [Minsky et al., 1986], *behavior-based robotics* [Brooks, 1986], and *reactive control* [Arkin, 1987]. Each of these approaches had been proposed nearly a decade earlier in an effort to work around the scalability issues of traditional symbolic reasoning systems [Nilson et al., 1971]. A common theme underlying each of these new approaches was the integrative nature of their design—that increasingly robust control and behavior could be achieved by adding additional, well-defined components to the overall system. This ideology, which we refer to as *integrative robotics*, was quickly found to be critically dependent on software design that supported long-term code integration and scalability. At the time, however, there were no robotics-related software integration tools available.

During the late 1980s, when integrative robotics was gaining precedence, a similar transformation was occurring in the mainstream commercial software industry. *Object-Oriented Programming (OOP)* was becoming widely accepted and rapidly phasing out older

structured programming methodologies for new projects [Booch, 1994]. The commercial sector's migration to OOP generated a need for integration tools and design methods, such as *Object Linking and Embedding (OLE)*, that could leverage reuse across multiple distinct OOP software projects [Brockschmidt, 1995]. This commercial trend eventually resulted in Microsoft's development of the *Component Object Model (COM)* as a tool for managing OOP-based software integration issues [Stafford et al., 1995]. At nearly the same time, the *Common Object Request Broker Architecture (CORBA)* was gaining popularity as a rival to COM, especially for Linux development [Duddy et al., 1996]. Concurrent with the release of COM and CORBA, there was a temporary push towards *architecture driven design* [Garlan et al., 1996] which emphasized establishment of a set of object-oriented guidelines to manage scalability issues.

On the hardware side of things, the processing requirements for commercial software were also continuing to grow due to the networking and internet revolution. The need for increased low-cost computing power led to the resurgence of *distributed computing platforms* to pool computational resources across networked computers [Mullender, 1993]. Ultimately, it was the coincidental availability of these commercial software tools and the academic need for them in robotics that led to the emergence of the original IMA. This version of IMA, referred to in this document as IMA 1.0, combined contemporary integrative robotics approaches of the mid-1980s with OOP-based software development, COM-based software integration, a distributed computing infrastructure, and architecture driven design.

Parallel work in software integration and scalability was being performed by other organizations during IMA's development. iRobot's Mobility platform, used in consumer-oriented robots, was originally based on IMA [iRobot Corp., 2002]. IBM's AgentBuilding and learning Environment (ABLE) provided a strict agent-based approach to designing large-scale autonomous software systems [Bigus et al., 2002]. Research in Model Integrated Computing (MIC) addressed scalability and integration at the modeling level [Sztipano-vits et al., 1997]. And, concurrently, Carnegie Mellon's Software Engineering Institute (SEI) established the now-popular Capability Maturity Model (CMM)

for managing the overall software process [S.E.I., 2000].

In the original work, IMA 1.0 was described as “a set of organizing principles and fundamental components that help designers manage the complexity of building robot control software that supports intelligent action” [Pack, 1998]. True to this description, a second version of IMA, referred to as IMA 2.0, was then developed to provide expanded language support, rapid component development, a simplified programming interface, and a new user interface [Olivares, 2000]. At this point, IMA 1.0 had been implemented on the ISAC and Helpmate robots, so it was decided that IMA 2.0 would be implemented on the Helpmate robot and used as a testbed for architecture innovation. The adoption of IMA 2.0 on mobile robots was successful, and it was further used on the Scooter and Skeeter robots. However, while IMA 2.0 addressed many problems with IMA 1.0 software development, it did not resolve all of them. [Olivares, 2001]. This led to the development of IMA 2.5, the latest refinement to the IMA software architecture [Olivares, 2003].

### 3. SYSTEM COMPATIBILITY

Each version of IMA comes with its own set of unique binary and source-code files. The result is that IMA 1.0 is not directly compatible with IMA 2.0 or IMA 2.5. Objects written in IMA 1.0 require significant changes to be made IMA 2.0 compatible, but, fortunately, these changes are systematic. The incompatibilities across versions stem primarily from COM object *Interface Identifier (IID)* and *Program Identifier (PID)* conflicts, as well as various programming model and remoting service changes. IMA 1.0 can coexist with IMA 2.0 and 2.5, but the later versions cannot coexist with each other on the same machine. More information on how to port IMA 1.0 objects to 2.0 can be found in the IMA 1.0 programmers guide or the IMA 2.5 thesis [Olivares, 2003].

### 4. PROGRAMMING REQUIREMENTS

IMA 1.0 development requires developers to have basic competency in 4 main areas:

- 1) Object Oriented Programming (OOP)
- 2) Visual C++ Technologies (C++, ATL, MFC)
- 3) Microsoft’s Component Object Model (COM)
- 4) IMA’s Programming Paradigm

The first three topics are programming areas outside the scope of this document. Developers are encouraged to refer to appropriate references for C++ and OOP [Stroustrup, 2000], Microsoft Visual C++ Technologies [Swanke, 2000], and COM/DCOM [Box, 1997].

The IMA programming paradigm will be reviewed in this document, but a full description can be found in the original work. We strongly suggest that new developers review the definitions section (Appendix A) for any unfamiliar terminology before continuing. A brief review of COM is provided in Appendix B.

### 5. TARGET AUDIENCE

IMA 1.0 is targeted for the academic development environment, where a group of developers generate IMA implementations to explore the software solution space. Throughout this document, the term *IMA implementation* denotes a robot control system consisting of 1) a version of the IMA software platform and 2) a set of developer-written software objects for that platform. The term *developer* denotes a person who develops IMA 1.0 software objects (typically graduate students). The term *end-user* denotes a person that interacts with, or uses a specific IMA 2.5 implementation on a robot—such as a soldier or civilian. While it is acknowledged that in most research scenarios developers will actually be end-users, this document will refer to the parties separately.

### 6. DEVELOPMENT PHASES

The process of using IMA 1.0 to control a robot falls into three phases: *installation*, *implementation*, and *execution*. During installation, the binary files (.EXE, .DLL, .OCX) required to run IMA 1.0 are copied to each computer on a LAN. The files and applications are then registered, and the appropriate DCOM servers and protocols are configured. After installation, the computer is ready to develop IMA components and host them in the runtime environment.

During the implementation phase, the components necessary to control the robot are written, compiled, debugged, and organized into agents—usually by multiple developers. The implementation phase can be further subdivided into *development*, *deployment*, *configuration*, and *activation phases*. During the development phase, components are written using Visual C++ and then compiled to DLL files. During the deployment phase, these DLLs are copied and registered onto any computers that will host or communicate with them. During the configuration phase, the components are instantiated inside of an agent using the Agent-Builder application and configured to communicate with other components on the network through *bindings*. Once the components on a network are configured, the agents may be activated to begin the execution phase. During this phase, components on the network interact to control the robot.

## 7. LAYOUT & TERMINOLOGY

IMA is a complex piece of software consisting of multiple subsystems. It is therefore useful to think of IMA in terms of an analogy to Java [Hopson et al., 1996]. IMA, like Java, consists of both a runtime environment and a set of development tools. In both cases, software is written for the runtime environment using the development tools. However, detailed knowledge of the underlying technologies are not required by the developer. To program in Java, for example, a developer does not need to know how interpreted code execution or just-in-time compiling are achieved. Java, like IMA, also supports transparent distributed computing via its Enterprise JavaBeans and Remote Procedure Call (RPC) functionalities. The primary difference between Java and IMA is that Java also provides a language to develop in, while IMA uses existing languages.

The IMA system consists of a *software platform* and *programming paradigm*. The software platform consists of a set of binaries that provide a *runtime environment* and *development tools* for software objects known as *components*. The programming paradigm consists of guidelines for writing these components at the code level, as well as the guidelines for grouping them into *agents*<sup>1</sup> within the runtime environment. Ultimately, groups of these software agents cooperate to control the robot [Maes, 1994; Marcenal, 1997].

## 8. PROGRAMMING PARADIGM

The IMA 1.0 programming paradigm describes how agents should interact and is the default approach for programming the control layer. It consists of the hybrid actor-agent paradigm described in the original work [Pack, 1997]. The high-level types of these agents and their organization within the agent network are defined by the *agent taxonomy*. The types of components and their organization within agents is defined by the *component taxonomy*.

**Agent Taxonomy.** Agents are labeled differently depending on the types of components they contain and the tasks they perform. At the highest level, they are categorized into *Atomic Agents* and *Compound Agents*. Atomic agents do not contain other agents, while compound agents host and/or control child agents. Agents are further classified into *Hardware/Resource (HR) Agents*, *Skill/Behavior (SB) Agents*, *Environment (EV) Agents*, *Sequencer (SE) Agents*, or *Multi-Type (MT) Agents*. HR agents interface to sensors or actuators on the robot. SB agents encapsulate basic behaviors or skills that the robot

software can execute. EV agents abstract and encapsulate the environment. SE agents coordinate a sequence of operations across other agents. MT agents combine the functionalities of two or more agent types.

**Component Taxonomy.** The IMA 1.0 component taxonomy defines five different types of component: *mechanisms, engines, representations, links, and relationships*. These objects are grouped hierarchically to create software agents. Software agents such as AgentBuilder implement the IAgentManager interface to provide a hierarchical container for other components.

**Mechanisms.** Mechanism components expose functions and data that other components in an agent can use to perform a task. IMA 1.0 mechanisms are COM objects that implement the IComponent and IMechanism interfaces. The IMechanism interface allows member functions to be called according to an integer numbering scheme. This allows dynamic programming at runtime to define which functions on the mechanism are called. Most components that IMA 1.0 developers will write will be mechanism components that encapsulate algorithms.

**Engines.** Engine components (alternatively referred to as sequencers or policies) serve as event dispatchers. Engines contain their own thread and internal logic. The resulting algorithm is used to call data processing functions on mechanisms and control the behavior of agents. The StateMachine Engine (SME), for example, maintains a configurable list of mechanism paths. When activated, the SME goes through its list of mechanisms and calls a specified function on each one. In this manner, the StateMachine engine “drives” the code found in mechanism components; hence the name “engine.” Without engines, the IMA 1.0 agent network would be a passive collection of function libraries with no thread of execution. Multiple engines allow multiple algorithms to function within the network simultaneously.

**Representations.** Representation components hold data that needs to be shared or replicated across the component network. In this sense, they are complementary to mechanisms, which hold functions that operate on the data in representations. The IRepresentation interface provides transparent proxy functionality, allowing the data in a local representation to be reflected in a proxy located on another machine or in another agent. This serves to protect the data from corruption and to isolate agent crashes (preventing the domino effect of exceptions). Representations define unique interfaces to the data they encapsulate and the corresponding mechanisms must be written to manipulate data according to those guidelines. The standard representations used in IMA 1.0 are the VectorSignal and TextQueue.

---

<sup>1</sup> An IMA agent is a collection of software components that interact with each other to provide functionality.

**Links.** Links encapsulate the connections between components in a system. A MotionLink, for example, contains three representations and automatically coordinates their proxies when given the path of another Agent’s MotionLink. In a sense, Links plug into each other, and act as “plugs” on Agents.

**Relationships.** Relationships encapsulate multiple links to provide pre-packaged functionality. The arbitration relationship, for example, encapsulates MotionLinks.

## 9. SOFTWARE PLATFORM

The IMA 1.0 software platform consists of *distributing*, *control*, and *application* layers. The bottommost layer, the distributing layer, transparently provides operating system abstraction, manages the layout of agents across the network, and provides the communication protocols for them. The central layer, the control layer, is a hierarchy of agents and components (some developer-written, some provided with IMA 1.0) that collectively control the robot. The topmost layer, the application layer, provides a graphical user interface to the other two layers.

**Distributing Layer.** The distributing layer allows components to interact transparently across machines and processes to control the robot. It consists of a *remoting model* that defines the overall structure of the component network, *remoting interfaces* that implement that structure in software, and *remoting services* that handle requests to create, access, and modify resources on the network. The distributing layer is designed to make the complex process of writing distributed software as simple and transparent as possible.

**Remoting Model.** The remoting model defines how components are created and organized across the LAN, as well as how they locate and interact with each other. The remoting model organizes components hierarchically into *locators*, *containers*, and *components*. Locators are the topmost nodes in the hierarchy. Their only children are containers. Containers may contain both components and other containers, but container nesting is not encouraged. IMA 1.0 developers will primarily write components.

The organized collection of locators, containers, and components is known collectively as the *component network*. The organization of the component network is designed to provide an intuitive naming convention following the form <Locator>\<Container>\<Component>. The Locator/Container/Component structure also loosely defines process boundaries for the component network. The container field denotes the agent the component is running in and is associated with a specific AgentBuilder window.

**Remoting Interfaces.** Distribution of IMA 1.0 components across a LAN is accomplished through three

three core interfaces: IComponent, IComponentContainer, and ILocator. Implementing the IComponent interface allows objects to be instantiated and distributed by the IMA 1.0 system. Implementing the IComponentContainer interface allows objects to provide hierarchical containment of other components. The ILocator interface is used to obtain pointers to remote components. Base classes and templates are provided for implementing these interfaces behind the scenes.

**Remoting Services.** The remoting services handle the majority of the work that allows components to locate and interact with each other. These services also implement the remoting model’s hierarchical structure through their control over component instantiation and paths. By exercising this control, the remoting services also provide the back end to applications that wish to display or modify the component network. The remoting services in IMA 1.0 consist only of the AgentLocator service.

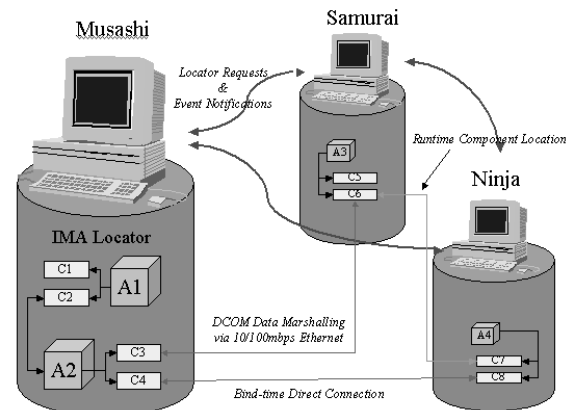


Figure 1. IMA 1.0 Locator and component network.

The AgentLocator service (a.k.a. “Locator”) is the single most important piece of software in the IMA system. One computer on the network provides the Locator object, which contains the master list of components addressable on the network. This master list contains the path of each component and its DCOM pointer. Typically, a component will query the Locator for a pointer to another component (which may be on another machine) so that they can interact in some way. These locating requests are handled by the Locator, as are requests to register and unregister components from the database during component instantiation and termination (see Figure 1). The Locator thus provides a single-object interface to a distributed network of objects.

## 10. CONTROL LAYER

The control layer consists of the hierarchy of agents and components that collectively control the robot. This layer is written almost exclusively by developers

using the IMA programming paradigm. The software in the control layer relies on the distributing layer to implement communication protocols between components. Developers are referred to the IMA 1.0 developer's guide for more information on writing components and agents.

## 11. APPLICATION LAYER

The application layer provides a means of configuring the underlying agent network. Application layer software is often targeted for the developer or for the end-user. In general, this software consists of *AgentBuilder*, *managers*, and *end-user interfaces*. For developers, the AgentBuilder application provides generic setup, configuration, and debugging of the agent network. Managers are ActiveX controls stored in DLLs, that are created within in AgentBuilder to help configure and visualize components. The application layer also contain end-user interfaces provided for the robot. Typically, laboratory researchers will want to use the developer interfaces (AgentBuilder and managers) to prototype, test, and debug robots, then develop mission-specific or task-specific end-user interfaces for the customer.

## 12. AGENTBUILDER

The AgentBuilder application provides developers with the primary interface for organizing their compiled components into intelligent agents. Each AgentBuilder window can contain multiple agents. Develop-

ers can add and remove components from agents, modify the properties on components, or call component methods. Modifying certain properties on a component can change which other components on the network it communicates with—these properties are know as bindings.

Figure 2 shows a typical AgentBuilder window running an agent named Maintenance. The various components contained within the agent are displayed on the left. Properties and methods for the selected component are displayed in the bottom panel. The managers associated with components in the agent are displayed on the right. The toolbar displays the initialize, register, bind, and activate buttons that control the agent activation process.

## REFERENCES

R.C. Arkin, "Motor schema-based navigation for a mobile robot: An approach to programming by behavior", Proceedings of the IEEE Conference on Robotics and Automation, Raleigh, NC, pp. 264-271, 1987.

S. Bagchi, K. Kawamura, "An architecture for a distributed object-oriented robotic system" , In IEEE/RSJ International Conference on Intelligent Robots and Systems, 2:711-716, July 1992.

J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills III, Y. Diao, "ABLE: A toolkit for building multiagent autonomic systems", IBM Systems Journal, Volume 41, Number 3, 2002.

G. Booch. *Object-Oriented Analysis and Design*. Addison-Wesley, 1994.

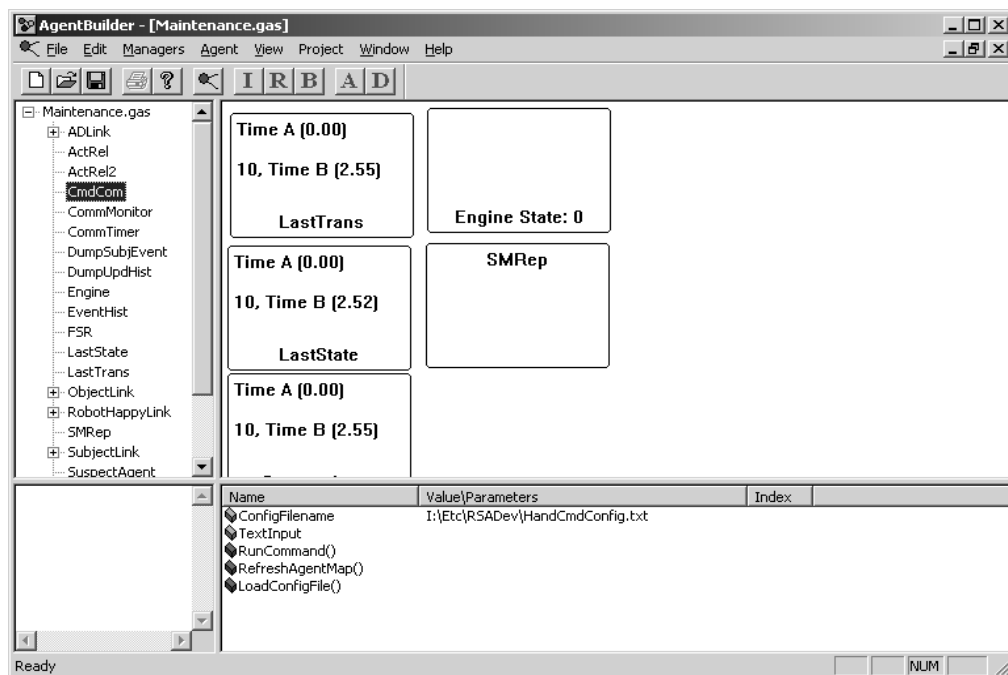


Figure 2. The IMA 1.0 AgentBuilder application.

D. Box, *Essential COM*. Addison-Wesley, 1997.

C. Brockschmidt. *Inside OLE: Second Edition*. Microsoft Press, 1995.

R. Brooks, "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation*, 2:14-23, 1986.

K. Duddy, Z. Yang, "CORBA: A platform for distributed object computing", *ACM Operating Systems Review*, 30(2):4-31, 1996.

D. Garlan, M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

K.C. Hopson, S.E. Ingram, *Developing Professional Java Applets*, SAM Publishing, 1996.

iRobot Corporation. "Mobility Information", <http://www.irobot.com/rwi/p10.asp>, 2002.

K. Kawamura, D.M. Wilkes, A.B. Koku, T. Keskinpala, "Perception-Based Navigation for Mobile Robots", *Proceedings of Multi-Robot Systems Workshop*, Washington D.C, March 18-20, 2002.

P. Maes, "Modeling adaptive autonomous agents", In *Artificial Life*, 1:135-162, 1994.

P. Marcenac, "The multiagent approach", *IEEE Potentials*, 16(1):19-22, 1997.

M. Minsky. *The Society of Mind*. Simon and Schuster, 1986.

S. Mullender. *Distributed Systems*. Addison-Wesley, 1993.

H.J. Nilsson, R.E. Fikes. "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial Intelligence*, 2(3-4):189-208, 1971.

R. Olivares, "IMA2: Vanderbilt Undergraduate Summer Research Program", Center for Intelligent Systems Internal Document, Vanderbilt University, 2000.

R. Olivares, "November 2001 Research Report," Center for Intelligent Systems Internal Document, Vanderbilt University, 2001.

R. Olivares, "IMA 2.5: A Revised Development Environment And Software Architecture," M.S. Thesis, 2003.

R.T. Pack, "IMA: The Intelligent Machine Architecture," Ph.D. Dissertation, Vanderbilt University, 1998.

R.A. Peters II, K. Kawamura, D.M. Wilkes, W.A. Alford, T.E. Rogers, "ISAC: Foundations in Human-Humanoid Interaction", *IEEE Intelligent Systems*, 2000.

S.R. Schach. *Object Oriented and Classical Software Engineering*. McGraw Hill Publishing, 2002.

Software Engineering Institute (SEI), "Capability Maturity Model Integration (CMMI): Frequently Asked Questions (FAQ) v1.3", Carnegie Mellon University, 2000.

P. Stafford, J. Powell. *COM: A Model Problem Solver*, Microsoft Developer Network, 1995.

J. Swanke, *COM Programming by Example: Using MFC, ActiveX, ATL, ADO, and COM+*, CMP Books, 2000.

B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 2000.

J. Sztipanovits, G. Karsai: "Model-Integrated Computing", *IEEE Computer*, pp. 110-112, April, 1997.

D. M. Wilkes, K. Kawamura, S. Suksakulchai, S. Thongchai, "Mobile Robot Localization using an Electronic Compass for Corridor Environment", *Proceedings 2000 IEEE International Conference on Systems, Man and Cybernetics*, Nashville, Tennessee, USA, October 8-11, 2000.

## APPENDIX A: DEFINITIONS

- **PID** – Program Identifier string identifying a COM object. Ex: MyDLL.MyObject
- **IID** – Interface Identifier string unique to a given interface. Ex: {594531-876242-1876274-243529}
- **GUID** – A string format used in the Windows registry to describe IIDs and CLSIDs, among other things. Ex: {298733-298730-23984848-129291}
- **System** – A combination of hardware and software.
- **Architecture** – A plan for building a set of related systems.
- **Binary file** – An compiled program in executable form. (Ex: DLLs and .EXEs)
- **Source Reuse** – Reusing functionality by sharing source code files (Ex: .CPPs and .Hs)
- **Binary Reuse** – Reusing functionality by sharing binary files.
- **OOP** – Object Oriented Programming.
- **COM** – Component Object Model, an extension to OOP that allows binary reuse.
- **DCOM** – Distributed COM allows binary reuse to operate across machines via RPC.
- **Marshalling** – Transmitting function parameters and return values across some communication medium. (Ex: RPC, TCP/IP, UDP)
- **RPC** – Remote Procedure Calls allow function parameters and return values to be transmitted and executed on a remote machine via TCP/IP marshalling.
- **In-Process** – Within a program's memory space.
- **Out-of-process** – Outside of a program's memory space.
- **Interface** – A named collection of functions implemented by an object implements to allow binary reuse.
- **IDL** – Interface Definition Language used to specify COM interface wrappers for OOP objects and libraries.
- **TLB** – Typelibrary file containing IDL information in binary form.
- **Binding** – Resolving a variable or function name to a machine-usable memory address.
- **Early binding** – Resolving a function address at compile time.

- **Late binding** – Resolving a function address at run-time using a variable, textual name.
- **Integration** – The act of adding new functionality to an existing system.
- **Scalability** – A measure of how well a system responds to integration.
- **Specification** – Written description of what a system is supposed to accomplish.
- **Design** – Written description of how a system will be engineered to achieve its specifications.
- **Instantiation** – Allocating memory for an instance of an object.
- **Registry** – The Windows program information database.
- **Registration** – Committing COM information for a binary file to the Registry. In IMA, registration is

IMA 1.0 takes advantage of COM to make the task of writing distributed software as transparent as possible to the developer. Without using COM, IMA developers would have to resort to using a proprietary method for writing distributed code (most likely involving TCP/IP coding) and managing late-binding to system objects. This would certainly reduce productivity and scalability for developers, as has been seen in previous blackboard architectures [Bagchi et al., 1992]. By taking advantage of COM, IMA allows developers to write distributed, run-time connectable code without their having to know almost anything about these two fields of software development. By using a public technology like COM, IMA system developers are also freed from having to develop and refine their own proprietary technology to provide these features.

## APPENDIX B: COM AND DCOM

The IMA 1.0 software platform is built on COM<sup>2</sup> technology. When used across machines, COM is referred to as *Distributed COM (DCOM)* for short. In plain English, COM lets you write objects that can 1) interact normally even if they are running on different machines, and 2) that can be plugged together at run-time instead of needing to be compiled together. The elegance of this approach is that developers do not have to write any additional code to enable these functionalities; it is completely taken care of by the compiler.

To provide a more technical description, COM is a set of programming guidelines that allows objects to be compiled and interoperate across process boundaries without referencing the original source code. Before COM, this functionality could only be achieved by referencing the original source code (header files) or through calling multiple DLL functions through a hand-written “wrapper” class. Both of these approaches were inconvenient and inelegant. By using COM and DCOM, a developer writes objects that can be transparently plugged into code that is already running—even if that code is on another machine. This functionality is achieved by COM compatible compilers, which place header-like information in the resulting binary files. The COM runtime also transparently directs function calls on objects to their appropriate process via *Microsoft Remote Procedure Calls (RPC)*. These two supervisory functions of the COM runtime do not impose any significant penalty on performance.

---

<sup>2</sup> It is important to note that the term “component” as found in the acronym COM is not exactly equivalent to the term “component” as found in IMA terminology. A COM component is simply an object that implements the IUnknown interface [Stafford, 1995]. An IMA component is an object that implements both IUnknown and the IMA-specific IComponent interface.