A revised version of the Intelligent Machine Architecture (IMA) software architecture and development environment has been implemented. The IMA 2.5 software architecture provides support for distributed concurrent programming using integrative robotics approaches. These approaches combine agent-based, behavior-based, and reactive algorithms to provide control for intelligent machines. In IMA 2.5, control algorithms are encapsulated within intelligent agents—collections of objects distributed across a network. Intelligent agents interact with each other according to guidelines specified in the IMA 2.5 programming paradigm. The software architecture and programming paradigm are accompanied by the IMA 2.5 development environment, a suite of tools for constructing, debugging, and managing intelligent agents.

The combined IMA 2.5 software architecture, programming paradigm, and development environment builds upon and refines features found in the original work. Presented within the body of this document are a detailed introduction, system overview, software development reference, and agent development reference for IMA 2.5 developers. Also presented are the results of preliminary implementation on the ISAC humanoid robot and a summary of conclusions from six years of IMA development. This latter information, intended for management and future architecture-level developers, addresses important issues found in the IMA architecture, software development process, and software engineering environment.
THE INTELLIGENT MACHINE ARCHITECTURE VERSION 2.5:
A REVISED DEVELOPMENT ENVIRONMENT
AND SOFTWARE ARCHITECTURE

By
Roberto E. Olivares

Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
in
Electrical Engineering

May, 2003

Nashville, Tennessee
To my parents

and colleagues.
This work could not have been accomplished without the many contributions of the graduate students and faculty of the Intelligent Robotics Laboratory. I thank Professor Mitch Wilkes for his steadfast support in the research of robotic development tools, whose guidance and insight have led me down a road less traveled, Professor Kazuhiko Kawamura for his incredible scrutiny of all things robotic, past and present, Professor Alan Peters for his endless hope and faith in our endeavors, Professor Steven Schach for his helpful comments, and Todd Pack for his original work.

I also thank Anthony Alford, Mark Cambron, Atit Srikaew, Jian Peng, Surachai Suksakulchai, Tamara Rogers, Kim Hambuchen, Ed Brown, Carlotta Johnson, Ali Sekmen, Bugra Koku, Jane Wilkes, Daniel Wilkes, and Flo Fottrell for their friendship and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Background</td>
<td>1</td>
</tr>
<tr>
<td>Layout And Terminology</td>
<td>3</td>
</tr>
<tr>
<td>Target Audience</td>
<td>3</td>
</tr>
<tr>
<td>Installation And Implementation</td>
<td>4</td>
</tr>
<tr>
<td>COM And DCOM</td>
<td>4</td>
</tr>
<tr>
<td>Design Elements &amp; Requirements</td>
<td>5</td>
</tr>
<tr>
<td>Design Goals</td>
<td>7</td>
</tr>
<tr>
<td>Feature Comparison</td>
<td>8</td>
</tr>
<tr>
<td>System Compatibility</td>
<td>10</td>
</tr>
<tr>
<td>Implementation Results</td>
<td>10</td>
</tr>
<tr>
<td>II. SYSTEM OVERVIEW</td>
<td>12</td>
</tr>
<tr>
<td>Distributing Layer</td>
<td>12</td>
</tr>
<tr>
<td>Remoting Model</td>
<td>12</td>
</tr>
<tr>
<td>Remoting Interfaces</td>
<td>13</td>
</tr>
<tr>
<td>Remoting Services</td>
<td>13</td>
</tr>
<tr>
<td>Locator Service</td>
<td>14</td>
</tr>
<tr>
<td>Information Service</td>
<td>14</td>
</tr>
<tr>
<td>Event Service</td>
<td>14</td>
</tr>
<tr>
<td>Application Layer</td>
<td>15</td>
</tr>
<tr>
<td>Distributed Agent Designer</td>
<td>15</td>
</tr>
<tr>
<td>ManagerBook</td>
<td>15</td>
</tr>
<tr>
<td>Command Console</td>
<td>16</td>
</tr>
<tr>
<td>Control Layer</td>
<td>16</td>
</tr>
<tr>
<td>Agent Model</td>
<td>17</td>
</tr>
<tr>
<td>Agent Structure</td>
<td>17</td>
</tr>
<tr>
<td>Agent Taxonomy</td>
<td>18</td>
</tr>
<tr>
<td>III. WRITING COMPONENTS</td>
<td>19</td>
</tr>
<tr>
<td>Component Development Task List</td>
<td>19</td>
</tr>
<tr>
<td>Creating IMA 2.5 Projects With Visual Basic</td>
<td>20</td>
</tr>
<tr>
<td>Visual Basic Templates</td>
<td>21</td>
</tr>
<tr>
<td>Creating IMA 2.5 Projects With Visual C++ 6.0</td>
<td>21</td>
</tr>
<tr>
<td>Component Icons</td>
<td>23</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>23</td>
</tr>
<tr>
<td>Source Code Conventions</td>
<td>24</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Feature comparison across IMA versions</td>
<td>9</td>
</tr>
<tr>
<td>2.</td>
<td>IMA 2.5 core project and file naming conventions</td>
<td>25</td>
</tr>
<tr>
<td>3.</td>
<td>Information about the intended use of each IMA 2.5 core directory</td>
<td>25</td>
</tr>
<tr>
<td>4.</td>
<td>Information about the intended use of each IMA 2.5 system subdirectory</td>
<td>26</td>
</tr>
<tr>
<td>5.</td>
<td>Registration and unregistration command lines for various system files</td>
<td>26</td>
</tr>
<tr>
<td>6.</td>
<td>IMA 2.5 console command description</td>
<td>38</td>
</tr>
<tr>
<td>7.</td>
<td>IMA 2.5 console command variations</td>
<td>38</td>
</tr>
<tr>
<td>8.</td>
<td>Binary files associated with IMA 2.5 framework</td>
<td>39</td>
</tr>
<tr>
<td>9.</td>
<td>Software engineering environment for three IMA software projects</td>
<td>61</td>
</tr>
<tr>
<td>10.</td>
<td>Software engineering details for three IMA software projects</td>
<td>63</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.</td>
<td>Screenshot of the IMA 2.5 Distributed Agent Designer user interface</td>
<td>31</td>
</tr>
<tr>
<td>2.</td>
<td>The insert component dialog box</td>
<td>32</td>
</tr>
<tr>
<td>3.</td>
<td>Bindings dialog in DAD</td>
<td>34</td>
</tr>
<tr>
<td>4.</td>
<td>Save components dialog in DAD</td>
<td>35</td>
</tr>
<tr>
<td>5.</td>
<td>Screenshot of the IMA 2.5 Console application</td>
<td>37</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

This work provides continued investigation of a development environment and software architecture for intelligent machines. 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. This thesis describes IMA 2.5, a new version of IMA aimed at further increasing the productivity and stability of the original system.

Background

IMA was first developed to address *software integration* and *scalability* issues in intelligent machines [Pack, 1997]. Software integration refers to the act of combining individual *software components* in such a way that they work together despite having been developed separately. Software scalability is a measure of how well a particular piece of software allows for future integration. 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 posited 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, architecture driven design, and a distributed computing infrastructure.

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, hereafter referred to as IMA 2.0, was then developed to provide rapid component development, expanded language support, an intuitive user interface, simplified programming interface, and background agent processing [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 quite successful, and was used on the Scooter and Skeeter robots as well. However, while IMA 2.0 addressed many issues of IMA 1.0 software development, it raised its own set of code reusability is-
sues and did not completely solve the complexity issues of IMA programming [Olivares, 2001]. This led to the development of IMA 2.5, the latest refinement to the IMA software architecture described herein.

**Layout And Terminology**

IMA is a complex piece of software that performs many functions and consists of many individual parts and systems. 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, but explicit knowledge of the technologies that make the runtime environment work are not required by the developer. To program in Java, for example, one does not need to know the internals of interpreted code execution or just-in-time compiling. Additionally, Java supports transparent distributed computing in a manner similar to IMA via its RPC functionalities and Enterprise JavaBeans. While the analogy between Java and IMA runs a long way, the primary difference between the two is that Java also provides a language, whereas IMA uses existing languages for development. IMA also focuses on guiding object interaction, which Java does not.

Using IMA-specific terminology, at the highest level IMA 2.5 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 within the runtime environment. Ultimately, groups of these software agents then cooperate to control the robot [Maes, 1994; Marcenal, 1997].

**Target Audience**

IMA 2.5 is targeted for academic, government, or commercial development environments where a group of developers aims to deliver an implementation to an end-user. Throughout this document, the term developer thus denotes a person who develops IMA 2.5 compatible software objects, typically a university or government researcher. 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 end-user de-

---

1 An IMA agent is a collection of software components that interact with each other to provide functionality.
notes 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.

**Installation And Implementation**

The process of using IMA 2.5 to control a robot falls into two phases: *installation* and *implementation*. During installation, the binaries (.EXE, .DLL, .OCX files) required to run IMA 2.5 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 components and host them in the runtime environment.

During the implementation phase, the components necessary to control the robot are written, compiled, tested, and then organized into agents. The implementation phase can be further subdivided into *development*, *deployment*, *configuration*, and *activation phases*. During the development phase, components are written using a programming language (such as Visual Basic or 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 IMA 2.5’s *Distributed Agent Designer (DAD)* software and configured to communicate with other components on the network. Once the component is configured, it or another component within its agent may be activated to begin its interaction with other components. When all components and agents are configured and activated, the implementation is complete.

**COM And DCOM**

The IMA 2.5 software platform is built on COM\(^2\) 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 runtime in-

---

\(^2\) 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.
stead 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.

IMA 2.5 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.

**Design Elements & Requirements**

IMA is first and foremost a software development product. Without adequate means for developing component software, the IMA runtime environment and programming paradigm are irrelevant. IMA developers spend the largest amount of their time writing, testing, and debugging their components; not designing agents or solving performance issues. Because of this, it is appropriate to evaluate the architecture using measures for a software development product rather than on the formality of its programming paradigm or the performance of its distributing code. Successful software development tools share certain common design elements. It is these
features that should be the benchmark of any version of IMA. Successful software development tools tend to: 1) maximize productivity, 2) aid the development of expertise, 3) provide intuitive approaches to problem solving, 4) make knowledge easily available 5) avoid getting in the way of the task (ease of use), 6) trivialize or automate routine tasks, and 7) their behavior is predictable [Schach, 2002].

A development environment like IMA should provide support for the languages that developers are most comfortable with. This serves both to reduce the learning curve for new developers and maximize productivity of experienced developers. This is particularly important at our laboratory, where most developers are electrical engineers with a limited programming background—typically just a few years of C programming. Clearly, designing the IMA programming paradigm around the presumption of experienced C++ programmers is a suboptimal choice given this environment. Following this line of reasoning, support for Visual Basic and Java were added to IMA 2.5 to avoid hampering developers with the more complex requirements of C++ and LINUX. Language support is a critical design choice. If the focus of IMA development was to fill in a well defined design specification as efficiently as possible, C++ on LINUX would perhaps have been a better choice for some situations. But the focus of IMA is building, testing, modifying, and rebuilding whole systems. Building these systems is further complicated by the constraint that, due to the grant-based nature of research funding, IMA must also support generating a variety of smaller component software very quickly. The emphasis on modification, experimentation, and fast implementation found in IMA is difficult to achieve with a complex development environment—especially given the developer experience level and turnover rates characteristic of universities. If developers in this environment are spending 80% of their time discovering how to write the code necessary to implement and test just a small portion of their system, they are being prevented from focusing on their research: how to intelligently control robots.

Appropriate choice of language also greatly aids development of expertise, since developers can focus on mastering design issues rather than working to understand their code. The ease with which developers can understand and visualize their code directly affects the complexity of the development process. Developers should be able to understand their code completely and be able to perform every common task quickly and reliably without resorting to reference material or workarounds. Choice of language contributes to this, since code in a non-preferred language can hide knowledge from the developer. Poor design choices in applications and framework can also thwart developers in finding important bits of knowledge, especially when working with
components provided by other people. Care should also be taken not to get in the way of the developer, or to make routine tasks irritating. Previous versions of IMA paid a price in this area because they were developed with C++ and required significant developer resources for even small user interface enhancements. The debugging process in previous versions of IMA was also unpleasant due to the high time requirements of compiling these large C++ projects. Lack of edit and continue\(^3\) features for debugging these C++ projects slowed the debugging process as well. In addition to not interfering with the developer’s task, factors such as a solid programming paradigm and a stable, responsive system are also important for a development environment. If the software platform is unstable or unpredictable fault diagnosis becomes time consuming and inconsistent, ultimately making the software development process impossible to manage.

**Design Goals**

With these design elements in mind, the IMA 2.5 system was created to resolve architectural issues and bugs that had been raised by the IMA 2.0 architecture and to solidify architectural features that had proven effective in previous versions of IMA. The result is a bug-fixed, performance-oriented, and polished release based on the strongest features of IMA 1.0 and IMA 2.0. Additionally, some new features were added to address multi-user development since these could now be supported by the overhauled applications and infrastructure. The key design goals from IMA 2.0—rapid component development, an intuitive user interface, and background processing—were also adopted in this release, but more emphasis was placed on refining the architecture than innovating new features. Heavy emphasis was placed on managing standards and system complexity, since many problems in the 2.0 version had arisen from developers having too much freedom (or lack of direction) in how to tackle certain key tasks. Ultimately, this refinement process focused development on software platform scalability, performance, stability, and management, as well as the refinement of programming paradigm guidelines.

---

\(^3\) Edit and continue functionality allows changes made in source code to be made to an executing program while debugging, without requiring a complete program termination and recompile.
Feature Comparison

Table 1 provides a feature comparison across IMA versions 1.0, 2.0, and 2.5. We can see that each version clearly adds functionality to the previous version. While IMA 2.5 lists a near equivalent increase in features as between IMA 1.0 and 2.0, the net difference between versions is actually smaller and denoted by the half point increase in version number. There are drastic changes between IMA 1.0 and 2.0, whereas the primary changes to IMA 2.5 are hidden from the developer. We can see from the table that IMA 1.0 provides the core functionality of a distributed infrastructure, core code components, and an agent design and debugging environment. Originally, this agent development environment was named *GenericShell* because it was meant to be a rapid prototype and eventually replaced. Unfortunately, the complexity of the original software led to it simply being modified and renamed as *AgentBuilder*.

IMA 2.0 broke compatibility with IMA 1.0 and added additional language support through compliance to Microsoft’s *OLE Automation* standard. This allowed control of IMA 2.0 objects from compilers such as Visual Basic (VB), Visual J++, and Java, as well as from scripting clients such as VBScript, JavaScript, and Visual Basic for Applications (VBA). The overall IMA programming approach was simplified to aid learnability and maintainability. Addition of background processing support allowed IMA 2.5 agents to run in the background of a Windows NT machine, without requiring a logged-on user or a running AgentBuilder window. IMA 2.0 also provided a completely new version of the agent development environment, named *AgentBuilder 2000*. Additional improvements were made in agent-level fault tolerance through expanded control over process space allocation for agents (memory protection) and in remote administration, which allowed one machine to control agent instantiation on remote machines.
Table 1. Feature comparison across IMA versions.

<table>
<thead>
<tr>
<th>Feature</th>
<th>IMA 1.0</th>
<th>IMA 2.0</th>
<th>IMA 2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Infrastructure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Supporting Base Classes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Agent Design Environment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Agent Debugging Environment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Property and Method Viewing</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Simplified Programming Model</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Agent-Level Fault Tolerance</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Remote Administration</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Background Processing</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>End-user GUI support</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Locator</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Separate Design / Debug / Admin Environments</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Component Registrar and Icons</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Event Services</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Distributed Information Services</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Automatic Binding System</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Automatic Documentation System</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Network Visualization Tools</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Command Console</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Languages Supported</td>
<td>Visual C++</td>
<td>Visual C++</td>
<td>Visual C++</td>
</tr>
<tr>
<td></td>
<td>Visual Basic</td>
<td>Visual Basic</td>
<td>Visual Basic</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td></td>
<td>VS.NET</td>
<td>VS.NET</td>
<td>VS.NET</td>
</tr>
</tbody>
</table>

IMA 2.5, while not directly backwards compatible with IMA 2.0, enhanced and corrected certain features of the previous version. Separate agent design (Distributed Agent Designer), debugging (ManagerBook), and remote administration (IMA2 Console) applications were developed to provide increased reliability and a more pleasant user interface. A component registrar database was added allowing the list of available components to be immediately browsed on the local machine, complete with unique graphical icons. IMA 2.5 enhanced the performance and stability of the agent locator service by removing its information and event services into different applications. GUI clients could now subscribe to IMA system information and events without fear of bogging down the performance of the underlying component network. The bindings system of IMA 2.0 was completely revamped to provide a unified interface to managing component interaction. Lastly, the new automatic documentation system in IMA 2.5 allows developers to query and submit information about a component.
or agent to a central repository with a simple push of the F1 key from Distributed Agent Designer (DAD)—the IMA 2.5 equivalent of AgentBuilder.

**System Compatibility**

As stated earlier, IMA 2.5 is not directly compatible with IMA 1.0 or IMA 2.0. However, IMA 2.0 objects can usually be recompiled with minor changes to make them IMA 2.5 compatible. IMA 1.0 objects require more significant changes to be made IMA 2.5 compatible, but these changes are systematic. These incompatibilities are due primarily to COM object Interface Identifier (IID) and Program Identifier (PID) conflicts, as well as various programming model and remoting service changes. More information on how to port IMA objects to IMA 2.5 can be found in the appendix.

**Implementation Results**

IMA 2.5 was implemented on the ISAC humanoid robot for testing purposes, and will most likely replace IMA 2.0 on the Helpmate and Scooter mobile robots. During its testing, IMA 2.5 showed a variety of improvements in both software platform and development environment compared to IMA 1.0 and 2.0. Refinement of core interface standards made code simpler and less bug prone, providing gains for both component and architecture developers. These refinements also enhanced support for the Visual C++ and Visual Studio.NET development of IMA 2.5 components. Changes to the programming paradigm cleared up remaining paradigm ambiguities and provided firm direction for all aspects of component coding. The most significant of these changes were to the bindings system and component taxonomy definition. Among other results was a more scalable infrastructure, now composed of distributed and separate Locator, Information, and Event Services to provide higher component communication performance. The new services also allowed more data about the agent network to be collected and distributed, with applications in system status monitoring and evaluation. Additional separation of administration, agent design, and agent debugging functionality into multiple programs like DAD, ManagerBook, and Console further increased system stability and ease of management. Focusing each of these applications to a single task decreased their learning curve and increased their utility.

The rest of this document will provide a more detailed explanation of the architecture, the changes made from the previous versions, an explanation of how to program the architecture, and an explanation of how to use
the provided development tools. A final chapter will discuss the conclusions that have been drawn from a 6 year case study of the IMA implementation on the Helpmate, Scooter, and ISAC systems.
CHAPTER II

SYSTEM OVERVIEW

The IMA 2.5 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 2.5) that collectively control the robot. The topmost layer, the application layer, provides a graphical user interface to the other two layers. In this chapter, we provide an overview of each of these layers, the functions they perform, and the software they comprise.

Distributing Layer

The IMA 2.5 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 IMA 2.5 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, thereby extending the hierarchy downward. IMA 2.5 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 IMA 2.5 naming convention is,
for the most part, exactly like the Universal Naming Convention (UNC) for Windows NT networks. This Locator\Container\Component structure also loosely defines process boundaries for the component network. Locators indicate which physical machine a component is executing on, then containers define which process space within that machine a specific component resides on.

A common topic of confusion is between the overall IMA 2.5 network structure and the details of its implementation. The remoting model describes the structure seen by an external observer viewing the IMA 2.5 network. Internally, the actual distinction between Locators, Containers, and Components is quite different. This will be discussed in the next section. The important thing to remember is that the IMA 2.5 distributing layer implements a full-fledged hierarchy. This is different from IMA-1, where hierarchy support is limited to a network three-levels deep.

**Remoting Interfaces**

Distribution of IMA 2.5 components across a LAN is accomplished through three core interfaces found in the IMA2.TLB file: IComponent, IContainer, and ILocator. Implementing the IComponent interface allows objects to be instantiated and distributed by the IMA 2.5 system. Implementing the IContainer interface allows objects to provide hierarchical containment of other components. The ILocator interface is used to obtain pointers to remote components.

Most IMA 2.5 component developers will not need to interact with any of these interfaces as they are primarily used by the IMA 2.5 distributing services (Locator, Information, and Event services) and application layer (AgentBuilder, DAD) to create and connect components together. Base classes and templates are provided to developers for implementing these interfaces behind the scenes.

**Remoting Services**

The IMA 2.5 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 the component network or allow its modification. If any soft-
ware wishes to query or alter the structure of the IMA 2.5 component network, it must ultimately go through the remoting services.

Locator Service

The IMA 2.5 Locator service is the single most important piece of software in the IMA system. There is one locator for each computer on the network, and each Locator contains the master list of components registered on that particular machine. This master list contains the path of each component and its pointer as described in the next chapter. Typically, a component executing on a computer will query their Locator for a pointer to another component (which may be on another machine) so that they can interact in some way. These requests are handled by the Locator, as are requests to register and unregister components from the database during component instantiation and termination. Additionally, each Locator contains a list of the other Locators it is connected to. If a component path is not found within the queried Locator, the search is extended to remote Locators until the component is found. The Locator thus provides a single-object interface to a distributed network of objects while avoiding a centralized repository.

Information Service

The IMA 2.5 information service was originally a part of the Locator service but was later separated for performance reasons. The information service maintains a variety of information about the Locator and components running on the local machine. Primarily, the information service provides a complete list of the local components and their type information. It also maintains a list of remote Locators the machine is connected to and other statistical information. The information service is primarily accessed by GUI clients in the application layer that wish to display information about the network. By handling these requests, the information service takes considerable network and processing load off of the Locator Service.

Event Service

Rather than polling the IMA 2.5 Locator or information services, many GUI applications instead subscribe to IMA 2.5 event system through the event service. This service notifies clients when a component is registered or unregistered from the network. Events are also broadcast when a Locator is added or removed from the re-
mote Locators list. By broadcasting these events, clients are notified of any changes in network topology. The event service was originally part of the IMA 2.0 Locator Service but was removed due to the stability and performance issues posed by a large volume of event subscribers.

**Application Layer**

The IMA 2.5 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. For developers, the AgentBuilder and ManagerBook applications allow generic setup, configuration, and debugging of the agent network. For end-users, the IMA 2.5 application layer would also contain any custom graphical user interfaces provided for the robot. Typically, laboratory researchers will want to use the developer interface to prototype, test, and debug robots, then develop mission or task-specific end-user interfaces once robot development is nearly complete.

**Distributed Agent Designer**

The IMA 2.5 Distributed Agent Designer (DAD) application serves the specific function of configuring components on the IMA 2.5 network. This is similar to the AgentBuilder 2000 application, but DAD was written specifically for the IMA 2.5 system. Components with the IMA 2.0 system cannot natively interoperate with those written for IMA 2.5. However, recompiling IMA 2.0 components for IMA 2.5 is not difficult. Unlike AgentBuilder 2000, DAD does not support loading managers for debugging components. That functionality is assigned to the ManagerBook application.

**ManagerBook**

The ManagerBook application was designed to off-load graphics processing from the AgentBuilder 2000 application. Traditionally, the GenericShell application (for IMA 1.0) provided a process space and a thread for the hosted agent and its managers\(^4\). However, this was undesirable for multiple reasons. First and foremost, increasing the number of components and managers used in a single GenericShell decreased the performance of the agent. This was partially from having many components running on the same thread, but also due to the amount of processor power necessary to rapidly update the display information in managers. To mitigate this

\(^4\) Managers are ActiveX controls written to interact with and display the data contained in IMA components.
problem, GenericShell allocated separate agent and user interface threads. This was found to help in the short term, but managers could still slow agents down if precautions were not taken. These performance penalties were undesirable and eliminated by moving manager execution into the separate ManagerBook program.

ManagerBook consists of a menu bar, a tool bar, and a manager area. The manager area contains multiple tabs that act as pages. Each page can contain multiple IMA 2.5 managers that the user drags onto the area from either AgentBuilder 2000 or from an AgentView within ManagerBook. The managers are displayed in sizeable windows within the page. The user can give each page’s tab a unique name, thus being able to quickly thumb through the pages in the book to display a given set of data about the network. This feature addresses the problem of having too many AgentBuilder windows open to support managers. In IMA 1.0 and 2.0, this problem made configuring and debugging multiple agents problematic.

Command Console

Because graphical user interfaces are complex, they tend to introduce new faults into processes that are relatively fault free without them. When trying to debug architecture-level issues in IMA, it is useful to have direct access, without a complex GUI, to low-level distributing layer information and services. Thus, the IMA 2.5 Command Console was developed using Visual Basic .NET. The command console provides a text based interface similar to the MS-DOS command line. Through this interface, IMA 2.5 specific commands can be issued and system information displayed. The program is found in the \C:\IMA2\System\Framework\IMA2_Console.exe file. Commands for the application can be accessed by typing help and pushing enter.

Control Layer

The IMA 2.5 distributing layer provides the scaffolding and services that allow developers to write distributed, object-oriented software that can be dynamically modified at runtime. This distributed programming implementation is completely separate from programming paradigms for robotic control and does not inherently provide any intelligence to the robot. The actual software that controls the behavior of the robot is contained within the IMA 2.5 control layer. This is where developer code is inserted. It allows various AI control algorithms (e.g. behavior-based, agent-based, subsumption-based, schema-based) to be implemented. The distribu-
ing layer allows these algorithms to be distributed among networked computers, and the application layer allows the developer an interface to their organization and parameters.

**Agent Model**

The IMA 2.5 *agent model* describes how agents should interact and is the default approach for programming within the control layer. It consists of the hybrid actor-agent paradigm described in the original work [Pack, 1998]. The high-level types of these agents and their organization within the IMA 2.5 agent network are defined by the IMA 2.5 *agent taxonomy*. The types of components and their organization within agents is defined by the IMA 2.5 *agent structure*.

**Agent Structure**

The IMA 2.5 agent structure has three different types of component, *agent components*, *engine components*, and *mechanism components*, that are grouped hierarchically to create software agents. A software agent consists of an agent component that may contain other engine, mechanism, and agent components within itself. Agent components implement the IContainer and IComponent interfaces to provide a hierarchical container for other components. Conceptually, an agent is simply a collection of components, but IMA 2.5 provides specific Agent components to implement this collection in code. IMA developers should not need to write new agent components. The IMA2.AgentEXE.Agent class provides a separate process space for its child components and is the default agent used in DAD. The IMA2.AgentDLL.Agent class uses its parent container process space to host its children components, and is useful for debugging components within the DAD environment or for avoiding performance issues with cross-process marshalling.

Mechanism components expose functions and data that other components can use to complete a task. IMA 2.5 mechanisms are just normal COM objects that implement the basic IComponent interface. IMA 1.0 and 2.0 mechanisms, in contrast, implement additional interfaces. The need for these older interfaces was supplanted by the call-by-name (late binding) functionality in IMA 2.5 inherited through OLE Automation compliance. Mechanism functions can now be referenced at runtime by providing a textual function name instead of having

---

5 Many of the IMA specific interfaces and components are completely defined in Chapter V.
to use an ambiguous integer number. Most components that IMA 2.5 developers will write will be mechanism components that encapsulate algorithms or data.

Engine components (alternatively referred to as sequencers) are somewhat different from mechanism components. They serve as event dispatchers and contain their own thread. Engines use a threaded function to periodically call data processing functions on other mechanisms or components according to some internal logic. The `IMA2_SimpleComponents.SimpleEngine` class, for example, maintains a configurable list of mechanism paths. When the SimpleEngine is activated (its enabled property is set to true), it goes through its list of mechanisms and calls a specified function on each one. In this manner, engines “drive” the code found in mechanisms in a particular order; hence the name “engine.” Without engines, the IMA 2.5 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. Another well-known engine is the `SimpleIMA2.StateMachine` engine which allows the user or developer to specify a state machine algorithm to guide function calls on mechanisms within the network.

Agent Taxonomy

Agents are labeled differently depending on the types of components they comprise 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.
CHAPTER III

WRITING COMPONENTS

This chapter serves as a programmer’s guide to writing IMA 2.5 components. A basic component development task list is provided to guide development efforts through the coding, build, and debug phases. Individual Visual C++ and Visual Basic walkthroughs are also given for creating new IMA 2.5 component projects. Tools for registering components are discussed, as are common registration and dependency problems. The chapter finishes with a discussion of how to use the IMA 2.5 bindings system.

Component Development Task List

To introduce software into the IMA 2.5 network, one must first code an IMA 2.5 compatible component then compile it to a DLL. Once the component is in DLL form, DAD can create the component in a parent agent on the network. The following steps will make the required component DLL:

Coding Phase:

1. Create a new COM project (typically a DLL).
2. Name the project following the naming convention: IMA2_CameraLib, for example.
3. Set the project description following the naming convention: (IMA2 Components) – Camera Library.
4. Reference the IMA 2.5 typelibrary in C:\IMA2\System\IMA2.TLB.
5. Add a new COM object.
6. Implement IComponent through inheritance (VC) or cut and paste (VB).
7. Specify your own interfaces.
8. Implement your own interfaces.
9. Fill in IComponent event handlers.
Build Phase:

11. Compile the project.

12. Register the project (usually automatic).

13. Make sure a copy of the DLL is in the C:\IMA2\Components directory so it is visible in DAD.

14. For convenience, an icon file for each component within the project should be placed in the C:\IMA2\System\Icons directory.

Debug Phase:

15. Set C:\IMA2\System\Framework\IMA2_DAD.EXE as the debug application for the IDE.

16. VC++ users must uncheck the “Create component in parent process” when inserting a component in DAD.

Creating IMA 2.5 Projects With Visual Basic

This section provides a walkthrough of creating an IMA 2.5 compatible COM project (known as an ActiveX DLL in VB) that contains an IMA 2.5 component. The full process is given first as reference, but can be significantly simplified by using the provided VB template for IMA 2.5 components. Installing and using the template feature is discussed in the next section.

Project creation steps:

1. Select ActiveX DLL project from the New Project dialog box. This dialog normally appears when you start Visual Basic, however you can alternatively go to File → New Project menu to make it appear.

2. Rename your project to something descriptive yet associated with IMA2, like “IMA2_CameraLib”

3. Reference the IMA2.TLB file by going to Project → References and selecting (IMA2 Base) – Base Interfaces.

4. Select Project → Properties → General → Program Description and enter something useful like (IMA2 Components) Camera Library. Following this suggested naming standard will be useful in numerous places, including the references and components dialogs in VB, Microsoft OLE Viewer, and DAD.
5. Rename the provided Class1 to something useful, like Camera, and then copy the code found in 
   C:\IMA2\Source\Framework\BaseClasses\ComponentImpl.cls file into your component to implement basic 
   IComponent functionality.

6. To debug your component, first compile it to the C:\IMA2\System\Components directory, then set the Project \ Properties \ Debugging \ Start Program option to point to IMA2_DAD.exe located in the 
   C:\IMA2\System\Framework directory.

7. You may execute the project in debug mode and insert your component into DAD from within VB, or you 
   can shut down VB, execute DAD, and insert your component normally.

### Visual Basic Templates

Installing the IMA 2.5 Visual Basic templates makes creating new components easy. To install the templates, copy the files found in the C:\IMA2\Base Classes\VB Templates directory to Visual Basic’s Templates\Projects subdirectory. For Visual Studio 6.0, this is typically the C:\Program Files\Microsoft Visual Studio\VB98\Template\Projects directory. Once the templates are installed, selecting File \ New Project \ New IMA2 Component in Visual Basic will create a new IMA 2.5 component project for you.

### Creating IMA 2.5 Projects With Visual C++ 6.0

Support for COM is less transparent in Visual C++ than it is in Visual Basic, since VC++ exposes more of 
COM’s inner workings to the developer. Because of this, IMA 2.5 developers considering Visual C++ should 
familiarize themselves with basic COM programming in Visual C++ before attempting to write complex IMA 
2.5 components. Primarily, the proper use of IDL to describe automation-compliant (and therefore VB compli-
ant) interfaces can be confusing to a beginner, as can the rules for proper allocation, deallocation, and referenc-
ing of interface pointers, variants, return values, and wide character (BSTR) strings.

While there are certainly more pitfalls in using Visual C++ for IMA 2.5 coding, there are advantages as 
well. Compiled Visual Basic 6.0 execution speed compares with C++ for most common tasks, but operations on 
large data sets or multiple mathematical operators still tends to perform faster in C++. Developers may also 
want to consider C++ for coding hardware components or vision components that require low-level memory
manipulation or API calls. Visual C++ also offers finer control over threading, device drivers, and memory allocation than Visual Basic.

Because most of today’s robotics research groups are multidisciplinary, Visual Basic 6.0 was considered the development platform of choice because of its ease of use, with Visual C++ support added later for those needing a deeper level of control. Due to this initial Visual Basic target, some of the Visual C++ code may seem indirect at times, but in all cases due attention has been given to presenting a user friendly C++ interface. The base classes provided with the IMA 2.5 framework handle nearly all the basic underlying functionality of a blank IMA 2.5 component and provide a few other protected functions to help the developer accomplish common tasks that would become tedious otherwise.

The following set of steps guides us through using Visual C++ to create a simple IMA 2.5 compliant project named IMA2_CameraLib that contains one IMA 2.5 component named Camera:

1. Select File → New → Projects → ATL COM AppWizard, fill in the project name (something descriptive like “IMA2_CameraLib”), select the project path and then push OK.
2. On the next page, make sure “Server Type” is set to “Dynamic Link Library (DLL)” and that everything else is blank.
3. Now that the project is created, add a new COM object by selecting Insert → New ATL Object → Simple Object, then supply the name for your class (ex. “Camera”) in the “Short Name” field and push OK.
4. In your object's .H file, add the following line of code after your other #include files:

   ```
   #include "c:\Ima2\Source\Framework\BaseClasses\ComponentImpl.h"
   ```

5. Add the following code at the end of your class inheritance list (remember to put a comma at the end of the line before it):

   ```
   public CComponentImpl
   ```

6. Add the following code in your COM_INTERFACE_ENTRY map:

   ```
   COM_INTERFACE_ENTRY(IComponent)
   ```

7. In the IDL file find the version(1.0) line, and change the next line to something more descriptive, like:

   ```
   helpstring("(IMA2 Components) - Camera Library")
   ```

8. Find the Library section and add the following after the importlib("stdole2.tlb") line:
importlib("c:\ima2\system\framework\ima2.tlb");

9. Find the coclass section and add the following after the [default] interface ICamera; line:

   interface IComponent;

10. After you compile your project, a copy of the resulting DLL file must be put into the C:\IMA2\System\Components directory to make it visible to DAD. You can automate this by adding the following line to the end of the commands in Project Æ Settings Æ Custom Build Æ Commands:

   copy $(TargetPath) C:\IMA2\System\Components\ 

11. Visual C++ can only let you debug code running in its currently attached process, so to debug your component you'll have to set the Project Æ Settings Æ Debug Æ Executable for Debug Session field to: C:\IMA2\System\Framework\IMA2_DAD.EXE

   If you want to debug an IMA 2.5 component written in VC++ 6.0, you must uncheck the “Create component in parent’s process” checkbox when inserting the component into an agent in DAD. This allows the component to be created in the IMA2_DAD process which VC++ 6.0 is currently attached to. Note that when you close the IMA2_DAD window that you have spawned, the object will be deallocated and become invalid. If the component is not properly created, VC++ breakpoints will appear not to work properly.

   **Component Icons**

   The IMA 2.5 application layer now supports icons for registered components. These icons further simplify the task of identifying and inserting new components. Once your component is compiled and registered, Visual Studio 6.0 can be used to make a 16x16 icon file (.ICO) that represents your component. Once the icon is complete, the filename must be changed to the component’s full PID with a .ICO appended at the end. So, for a component with a PID of IMA2_CameraLib.Camera.1 you would have to name the file IMA2_CameraLib.1.ico and place it in the C:\IMA2\System\Icons directory.

   **Naming Conventions**

   Naming conventions are very important in an environment with many developers developing multiple objects for mass use. If naming conventions are not used, the function of objects can become obscured, descrip-
tions of how to use those objects can become hard to specify, and more specific knowledge about each compo-
nent is necessary to understand an agent.

Source Code Conventions

The following conventions are used for identifiers within IMA 2.5 projects:

- **Class names** are not preceded by any prefix.
- **Structure names** are not preceded by any prefix.
- **Default interfaces** for a class are not preceded by a prefix.
- **Interfaces** implemented across projects are denoted by the prefix “I”:
  
  Ex: IComponent, ICameraImage.

- **Enumerations** are preceded by a context-specific prefix:
  
  Ex: LE_Register which stands for “Locator Event” of type Register.

Project Naming Conventions

The following conventions are used for IMA 2.5 project identifiers. It is strongly recommended that these
be followed, as some features of DAD expect them:

- **Project names** follow the form IMA2_<name>Lib.
  
  Ex: IMA2_CameraLib.

- **Project binaries** follow the form IMA2_<name>Lib.<extension>.
  
  Ex: IMA2_CameraLib.DLL.

- **Component project descriptions** follow the form (IMA2 Components) - <description> Library.
  
  Ex: (IMA2 Components) – Camera Library.

- **Manager project descriptions** follow the form (IMA2 Managers) - <description> Managers.
  
  Ex: (IMA2 Managers) – Camera Managers.
The following table provides examples of these conventions as applied to IMA 2.5 core projects:

**Table 2. IMA 2.5 core project and file naming conventions.**

<table>
<thead>
<tr>
<th>Category</th>
<th>Filename</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>IMA2.TLB</td>
<td>(IMA2 Base) - Base Interfaces</td>
</tr>
<tr>
<td></td>
<td>IMA2_Bindings.DLL</td>
<td>(IMA2 Base) - Bindings Collection</td>
</tr>
<tr>
<td></td>
<td>IMA2_Helper.DLL</td>
<td>(IMA2 Base) - Helper Library</td>
</tr>
<tr>
<td></td>
<td>IMA2_Events.EXE</td>
<td>(IMA2 Base) - System Event Service</td>
</tr>
<tr>
<td></td>
<td>IMA2_Info.EXE</td>
<td>(IMA2 Base) - System Information Service</td>
</tr>
<tr>
<td></td>
<td>IMA2_Locator.EXE</td>
<td>(IMA2 Base) – System Locator</td>
</tr>
<tr>
<td>Component</td>
<td>IMA2_Agent.EXE</td>
<td>(IMA2 Components) - Agent</td>
</tr>
<tr>
<td></td>
<td>IMA2_StateMachine.DLL</td>
<td>(IMA2 Components) - State Machine</td>
</tr>
<tr>
<td>Manager</td>
<td>IMA2_StateMachine.OCX</td>
<td>(IMA2 Managers) - State Machine Manager</td>
</tr>
</tbody>
</table>

**Directory Conventions**

The IMA 2.5 system is designed to run out of the `C:\IMA2` directory. Developers should add components, managers, and icons to the provided subdirectory within `C:\IMA2`. In general, developers should not modify anything within the `C:\IMA2\System` directory. The following provide general guidelines for directory usage:

- Components (.DLL, .EXE) should go in the `C:\IMA2\Components` directory.
- Managers (.OCX) should go in the `C:\IMA2\Managers` directory.
- Icons (.ICO) should go in the `C:\IMA2\Icons` directory.

**Table 3. Information about the intended use of each IMA 2.5 core directory.**

<table>
<thead>
<tr>
<th>IMA 2.5 Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\IMA2</td>
<td>Root directory, contains registration batch files and other directories.</td>
</tr>
<tr>
<td>C:\IMA2\Components</td>
<td>Contains IMA 2.5 components. Developers should add new component files here.</td>
</tr>
<tr>
<td>C:\IMA2\Configuration</td>
<td>Contains configuration files for machine-specific components. Developers should code their components to put machine-specific configuration information into this directory.</td>
</tr>
<tr>
<td>C:\IMA2\Documentation</td>
<td>Contains IMA 2.5 system help files and FAQs. Developer generated documentation should NOT be put here, but be put on the server instead.</td>
</tr>
<tr>
<td>C:\IMA2\Icons</td>
<td>Contains icons for IMA 2.5 components. Developers should add their new icons here.</td>
</tr>
<tr>
<td>C:\IMA2\Managers</td>
<td>Contains IMA 2.5 managers. Developers should add their new managers here.</td>
</tr>
<tr>
<td>C:\IMA2\Source</td>
<td>Contains the source code for IMA 2.5 framework.</td>
</tr>
<tr>
<td>C:\IMA2\System</td>
<td>Contains IMA 2.5 system specific directories and binaries. Developers should generally NOT modify these directories.</td>
</tr>
</tbody>
</table>
Table 4. Information about the intended use of each IMA 2.5 system subdirectory.

<table>
<thead>
<tr>
<th>IMA 2.5 System Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\IMA2\System\Applications</td>
<td>Contains additional third party applications used by the framework.</td>
</tr>
<tr>
<td>C:\IMA2\System\Framework</td>
<td>Contains the core binaries needed to run IMA 2.5, including DAD, Manager-Book, Console, Locator Service, Information Service, Event Service.</td>
</tr>
<tr>
<td>C:\IMA2\System\Help</td>
<td>Contains the component and agent documentation that is stored online within DAD.</td>
</tr>
<tr>
<td>C:\IMA2\System\Mappings</td>
<td>Contains the mappings of components to managers generated by DAD.</td>
</tr>
<tr>
<td>C:\IMA2\System\Third Party</td>
<td>Contains miscellaneous third-party binaries used by the framework.</td>
</tr>
</tbody>
</table>

Registering Components

Registration is the process by which one of the programs listed below loads a COM server into memory, then instructs the server to write its COM registration information to the registry. Visual Studio 6.0 and Visual Basic 6.0 will automatically register components upon a successful build, however there are times when manually registering or unregistering a component are necessary. The following lines can be used to register or unregister varying types of component servers, assuming you have power user or administrator permissions:

Table 5. Registration and unregistration command lines for various system files.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Registration Command</th>
<th>Unregistration Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DLL, .OCX</td>
<td>regsvr32 &lt;file path&gt;</td>
<td>regsvr32 &lt;file path&gt; /u</td>
</tr>
<tr>
<td>.TLB</td>
<td>regtlib &lt;file path&gt;</td>
<td>regtlib &lt;file path&gt; /u</td>
</tr>
<tr>
<td>.EXE</td>
<td>&lt;file path&gt; /regserver</td>
<td>&lt;file path&gt; /regserver</td>
</tr>
</tbody>
</table>

It is important to remember that registering a component server performs a dependency file check before allowing a successful registration (transfer of COM settings to the registry). Therefore, many mysterious registration failures can be attributed to wrong dependency versions or the operating system’s inability to locate the files. The best way to solve these problems are to double check all dependency versions and CLSIDs, and to make sure that all required files (i.e. Intel Image Processing Library, Computer Vision Library, Signal Processing Library) files are accessible through the OS search path. For an example of how to register multiple files, see the C:\IMA2\System\IMA2_Register.bat file.
Checking COM Registration And Dependencies

Microsoft Visual Studio ships with two very helpful programs: oleview32.exe and depends.exe. Microsoft OLE Viewer lists registered IMA2 components under the Automation node and lists IMA2 managers under the Controls node. If a component is displayed here, its registration information is properly entered into the registry. Double clicking the component’s entry will attempt to create the component. This may fail for a variety of reasons. During the object creation process, security settings are verified (server launch permissions), file dependencies are double checked, and the constructor for the class is called. A failure in any of these parts may cause object instantiation to fail, and therefore likely fail in DAD or other development tools. If the component registers properly, but does not instantiate properly, the constructor or dependency versions are probably the culprit.

To check dependencies, the Depends.exe program provided with Visual Studio should be used to look for red nodes or red procedures that could not be matched. For constructor problems, VC or VB must be used to step through constructor execution to locate the cause of the problem.

Checking Cross-Machine Registration

Checking cross-machine registration is a difficult task. Typically, if the components servers are properly registered and installed on two separate IMA2 machines, remote instantiation through DAD should run smoothly. To check the COM remoting functionality independently of IMA2, a remote instantiation through OLE Viewer is useful. This can be performed by right-clicking the component’s node and selecting the Create Instance On… option, then specifying the machine name to attempt the instantiation on. In order for this to be successful, a local instantiation of the objects on their respective machines must first be valid. Next, the two machines must have a functioning trust relationship (such as being on the same domain) to verify identities (logon names and passwords). Once the identities have been verified, the accounts being used to launch the server must have the appropriate security permissions. This can usually be checked using the DCOMCnfg.exe program and browsing through both default, user-specific, and server-specific permissions. As you can see, remoting objects is all about security considerations once they are confirmed to be functioning locally.
Publishing And Using Bindings

In order for agents to function properly, the components that they consist of must be able to communicate with one another. When a *source component* wishes to communicate with another component, it *publishes* a binding. At runtime, the binding is *configured* to point to an existing *target component*. Once configured, the source component attempts to connect to the target by *resolving* the binding into a pointer. Once resolved, the two components continuously interact until the binding is *released* by the source component. Bindings information is contained within the component and made available through the IComponent interface. Components are responsible for interacting with their internally stored bindings collection to publish bindings.

A good example of this process is a camera component. The camera component would publish a binding named “Output Image”, for example, by calling the IConfigureBindings.Add() function on its bindings collection member variable in its OnConstruct() function. At runtime, the developer would use DAD to then double-click the binding and connect it to an Image component existing on the network. Whenever a function on the Camera component was called, it would then use IBindings.Item() with the binding name to resolve a pointer to that Image component. When the camera component is done using the pointer, it releases it.

It is important to note that any use of bindings should assume the default condition: that the binding has not been resolved. It is also very important to release pointers obtained through bindings or COM reference counting will keep components alive longer than necessary.

**Sample Visual Basic code publishing two bindings:**

```vbnet
Private Sub OnConstruct()
    Dim BI As IMA2_BindingInfo

    'Prepare the binding information structure.
    BI.PIDs = Array("IMA2_CameraLib.Image")
    BI.IIDs = Array("ICameraImage")
    BI.BindType = BT_Default

    'Publish the "Input Image" binding.
    BI.Name = "Input Image"
    Call m_Bindings.Settings.Add(BI)

    'Publish the "Output Image" binding.
    BI.Name = "Output Image"
    Call m_Bindings.Settings.Add(BI)

End Sub
```
Sample Visual Basic code resolving two bindings:
'Public interface function that copies input image to the output image.
Public Sub CopyImage()

'Declare binding pointers.
Dim InputImage As ICameraImage
Dim OutputImage As ICameraImage

'Attempt to resolve bindings to input and output images.
Set InputImage = m_Bindings.Pointers("Input Image")
Set OutputImage = m_Bindings.Pointers("Output Image")

'Confirm that pointers are valid.
If InputImage Is Nothing Or OutputImage Is Nothing Then Exit Sub

'Perform operation on both bind targets.
Call OutputImage.SetData(InputImage.GetData)

End Sub
Distributed Agent Designer (DAD) is a tool for creating and debugging agents. In IMA 2.5, agents consist of hierarchically grouped COM components that work together to achieve some function. Components are added to an agent component which provides a unique, protected memory space and thread for components to run on. Building an agent thus involves creating the agent component somewhere within the IMA 2.5 network, then creating child components within it. Once the agent is built, the basic properties, methods, and bindings of each component within the agent can be inspected or modified from within the program. DAD is also an administrative tool, and it provides information about the status of IMA 2.5 network, allows remote configuration of agents running on other machines, and allows agents to be loaded from and saved to files.

**Differences From Versions 1.0 And 2.0**

DAD differs from the AgentBuilder program in a few important ways. The AgentBuilder program provided both a user interface space and memory space for components and their managers. Due to performance, usability, and fragility issues, the user interface space was taken out of DAD and put into a separate program named ManagerBook. The ManagerBook program now serves the specific role of organizing and running managers that provide a custom interface to IMA 2.5 components. DAD is specifically designed to organize and administer agents on the network, somewhat like the Explorer program allows files and programs to be administered in Windows. DAD is also remarkably similar to the Mobility Object Manager (MOM) interface for the Mobility software architecture [iRobot, 2000].

**Starting DAD**

To start DAD, push the Start button, then Programs \IMA 2.5 \Distributed Agent Designer. Alternatively, the \IMA2_DAD.EXE file is located in C:IMA2\System\Framework directory.
User Interface Overview

The primary interface to the IMA 2.5 network is through the agent tree view on the left hand side of DAD. The agent tree view can display either the entire network or specific agents within it. Right clicking on an item within the agent tree view displays the operations allowable on the object. Right clicking on the *IMA2 Network* icon, for example, provides the option of connecting new machines into the IMA 2.5 network. Right clicking on the on a machine icon (e.g. “Kangaroo” in Figure 1) allows top level IMA 2.5 agents to be created. Right clicking on an agent component (e.g. “Ag1” in Figure 1) then allows creating child components within the agent (e.g. “Sampler” in Figure 1). The path of the currently selected component is displayed just above the agent tree view.

*Figure 1. Screenshot of the IMA 2.5 Distributed Agent Designer user interface.*

The agent list view displays the sub items within the currently selected agent tree view control. When the IMA 2.5 network is selected in the agent tree view, for example, the agent list view displays the Locators currently connected to the network. When a Locator is selected, the agent tree view displays the top level agents within that Locator. When an agent is selected, the agent list view displays the components within that agent. When a component is selected, the agent list view displays the component’s properties, methods, and bindings.

The component view displays basic information about the component selected in the agent tree view. The name of the component is displayed, as well as its class identifier (PID), the machine the component is running on, etc.
on, and the process and thread that is executing it. If it’s a sequencer component, the state (active or deacti-
vated) is also displayed.

Creating And Naming Agents

To create and name an agent, first right click on the computer icon with the local machine’s name. Select
New $\rightarrow$ Agent to create the default IMA 2.5 agent component. Type in the name you wish to give your agent
and push enter when finished. The editing box will not disappear until a valid name is entered or the escape key
is pushed to cancel the action.

Creating Child Components

To create child components, right click on the agent icon in the agent tree view. Select New $\rightarrow$ Component
to bring up the insert component dialog. Select the “New” tab to browse the component libraries (DLLs) cur-
rently registered on this machine. Double click the library you wish to browse to show its components. Double
click a component or push the OK button to create it in your agent. Alternatively you can push the backspace
key or the button with the yellow folder and an up arrow to return to the library list. Type in the name you wish
to give your component and push enter when finished. The editing box will not disappear until a valid name is
entered or the escape key is pushed to cancel the action.

Figure 2. The insert component dialog box.
Removing Components

To remove components, right click on the component icon in the agent tree view. Select Delete. This will remove the component and any other child components from the network. Make sure that the agents accessing this component are deactivated before removing the component to prevent complications.

Viewing And Setting Properties

DAD allows the properties and methods of a component to be viewed and configured in the agent list view. To view the properties and methods supported by an object’s default interface, select the component in the agent tree view. Properties are shown with a hand icon and methods with a little flying box. The type and values of properties and methods are listed in the columns of the agent list view. To refresh property values, push the F5 key while an item in the agent list view is selected. To edit a property, double-click it in the agent list view and enter a new value in the supplied dialog box. Only properties and methods with basic data types are supported in DAD. If more advanced configuration is necessary a component manager should be created and used in ManagerBook program.

Calling Methods

To call a method on the component, double-click the method name and enter its parameter list in the space provided. Only properties and methods with basic data types are supported in DAD. If more advanced configuration is necessary a component manager should be created and used in ManagerBook program.
Configuring Bindings

To view the bindings on a component, select the component in the agent tree view. Bindings are denoted by a handshaking icon in the agent list view, and are listed with their name and current value. To configure a particular binding, double click it. The Configure Bindings dialog box is shown where you can select components from the network to bind to. If the binding supports multiple targets (a binding collection), then multiple components may be selected. If a binding is not desired it may be deleted from the bindings list view on the right hand side of the dialog. Push OK to save your changes or Cancel to exit without changing the property.

![Figure 3. Bindings dialog in DAD.](image)

Activating Agents And Sequencer Components

To activate the sequencer components in an agent, right click on the agent in the agent tree view and select Activate. To deactivate the agent, select the Activate option again. If the agent is activated, the Activated menu option will have a checkmark next to it. The same steps may be applied directly to the sequencer component in the agent. Activated sequencer components have theirs names shown in the agent tree view in green letters, deactivated ones are shown in red.
Saving Agents

To save an agent, right click on the agent’s icon. Select Save As to bring up the Save Component dialog. You may select the components in the tree view and then rename or delete them. Selecting a component will show its basic information. Push OK to confirm the list of components you wish to save to a file and to bring up the file select dialog. Select the file or directory you wish to save the agent to, or type in the name of a new component file. Push the Save button to save the file.

![Figure 4. Save components dialog in DAD.](image)

Loading Agents

Right click on the agent or locator you wish to load the agent into. Select New → Component to bring up the insert component dialog. Select the “Existing” tab and push the button in the “Look In” text box to select a file to preview. Once the file is selected, you may rename or remove components from the tree view before loading the agent. Selecting a component will show its basic information. Push OK to load the agent.

Accessing Documentation And Help Files

IMA 2.5 provides two types of online documentation to developers: class-specific and agent-specific. Both types of documentation can be accessed from DAD by selecting a component in the agent tree view and pushing the F1 key. Documentation is stored in rich text format (RTF) and can include formatting found in most word processors. The DAD help system allows documentation files to be both read and modified. Modifications are automatically saved to the server while older copies are archived. This help system allows IMA 2.5 developers
to document and revise component information together through one central interface. It also allows developers to immediately access the information from a central repository, hopefully making documentation a simpler team effort.

When the DAD help window is accessed, both class specific and agent specific documentation is provided. Class specific documentation pertains to a specific type of component, such as the instructions and documentation for using the IMA 2.5 state machine (IMA2_SMCompStateMachine) class. The second type of help provided is agent-specific help. This is documentation that pertains to a specific component within a specific agent. This documentation is transparently loaded and stored directly to the network server and acts as a general help repository for component classes. Agent specific documentation can be thought of as notes on how specific components are configured within an agent, as well as notes on how to use the agents themselves.

Configuring Locators

In order to create or access components on a remote machine in the network, the machine running DAD must have the appropriate connections. In most cases, connecting to remote locators is handled automatically by the Locator when it is first connected, however new machines may be added to the network once the Locators are already running which requires manual configuration. To connect a remote Locator to the current machine using the Connect Locators dialog box, select Tools → Locators from the DAD menu. A list of machines on the network is then provided and you may select one and push the connect button to establish a connection.

IMA 2.5 Console

The IMA 2.5 command console dumps information directly from the Locator, Information, and Event services to the console, where it can be reviewed without the ambiguities of a user interface. It was designed for performing low-level maintenance and debugging tasks on the both the application (Distributed Agent Designer, ManagerBook) and distributing layers (Locator, Information, and Event services). At times the network may become improperly configured or unstable, at which point the console also provides a bare-bones interface to configuring and managing components. Basic commands include displaying network status, creating and registering components, and shutting down the local machine’s IMA services. To exit the program, the “exit” command is used.
Starting The Console

To start the console, Select Start button → Programs → IMA 2.5 → IMA2 Console. Alternatively, the IMA2_CONSOLE.EXE file is located in “C:\IMA2\System\Framework” directory.

Console Interface Overview

The IMA 2.5 command console works exactly like a MS-DOS or LINUX command console. The prompt displays the network path of the local machine and Locator. In figure 1, we see this is the “kangaroo” machine. At any time, the “help” command provides a list of the supported commands and a brief description. For more detailed information, you can type “help [command]” for more information.

Figure 5. Screenshot of the IMA 2.5 Console application.

Since IMA 2.5 Console is an MS-DOS application, any output in its window can be captured to the Windows clipboard and saved to a file. To capture screen output use the [alt + space] key combination to access the console’s window menu, then select Edit → Mark. Click and drag the mouse over the text you wish to copy until it is all highlighted. Push the enter key or select [alt + space] → Edit → Copy to copy the text to the Windows clipboard. To facilitate copying text and viewing readouts, the size of the text window and buffer can be modified by selecting [alt + enter] → Properties → Layout and editing the “Buffer Size” fields.
Console Command List

The console application currently supports the following commands for displaying network status and managing components:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Displays the Locator service status and the paths of registered components.</td>
</tr>
<tr>
<td>Info</td>
<td>Displays the Information service status and the cached distributing information.</td>
</tr>
<tr>
<td>Events</td>
<td>Displays the Event service status and list of registered event clients.</td>
</tr>
<tr>
<td>Create</td>
<td>Creates a component on the network with the given path and PID on a specified machine. The component is created within the parent and registered on the locator specified by its path.</td>
</tr>
<tr>
<td>Unregister</td>
<td>Unregisters and destructs a currently registered component. If the component has children, the application will ask whether the children should be unregistered first or if the operation should be cancelled. Removal of parent components without prior removal of children is not allowed.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops the Locator, Information, and Event services.</td>
</tr>
<tr>
<td>Shutdown</td>
<td>Shuts down the entire IMA 2.5 network. Terminates the Locator, Information, and Event services on the specified machine. The console will exit after shutdown, leaving no IMA 2.5 services running on the machine. When the console or another IMA 2.5 program using the services starts up, the system will be restarted.</td>
</tr>
<tr>
<td>Exit</td>
<td>Closes the command console.</td>
</tr>
<tr>
<td>Help</td>
<td>Displays basic commands to the user.</td>
</tr>
</tbody>
</table>

Table 6. IMA 2.5 console command description.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Displays locally registered components.</td>
</tr>
<tr>
<td>List [machine]</td>
<td>Displays components registered on a given machine.</td>
</tr>
<tr>
<td>List all</td>
<td>Displays all registered components on the network.</td>
</tr>
<tr>
<td>Info</td>
<td>Displays locally registered component information.</td>
</tr>
<tr>
<td>Info [machine]</td>
<td>Displays component information on a given machine.</td>
</tr>
<tr>
<td>Info all</td>
<td>Displays all component information on the network.</td>
</tr>
<tr>
<td>Events</td>
<td>Displays locally registered event clients.</td>
</tr>
<tr>
<td>Events [machine]</td>
<td>Displays event clients for a given machine.</td>
</tr>
<tr>
<td>Events all</td>
<td>Displays all event clients on the network.</td>
</tr>
<tr>
<td>Create [Path],[PID]</td>
<td>Creates a component of the given type and with the specified path.</td>
</tr>
<tr>
<td>Create [Path],[PID],[Machine]</td>
<td>Creates a component of the given type and with the specified path on a specific machine.</td>
</tr>
<tr>
<td>Unregister [Path]</td>
<td>Unregisters and destructs the component with the specified path.</td>
</tr>
<tr>
<td>Shutdown</td>
<td>Shuts down IMA 2.5 on the local machine.</td>
</tr>
<tr>
<td>Shutdown [Machine]</td>
<td>Shuts down IMA 2.5 on the specified machine.</td>
</tr>
</tbody>
</table>

Table 7. IMA 2.5 console command variations.
CHAPTER V

FRAMEWORK OBJECT REFERENCE

The following chapter provides critical information about the primary interfaces within the IMA 2.5 distributing layer. The complete list of applications and libraries associated with the IMA 2.5 framework is first provided, and then the individual libraries and their contained interfaces are described. Within each library, the contained structures and enumerations are listed, as are the properties and methods found on each interface.

Object Library And Application List

The following figure provides the complete list of binaries associated with IMA 2.5 framework: their category, filename, and description. Understanding which files provide which information and services simplifies the process of explaining their contained interfaces.

Table 8. Binary files associated with IMA 2.5 framework.

<table>
<thead>
<tr>
<th>Category</th>
<th>Filename</th>
<th>File Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework Applications</td>
<td>IMA2_DAD.EXE</td>
<td>IMA2 Distributed Agent Designer</td>
</tr>
<tr>
<td></td>
<td>IMA2_MgrBook.EXE</td>
<td>IMA2 Manager Book</td>
</tr>
<tr>
<td></td>
<td>IMA2_Console.EXE</td>
<td>IMA2 Command Console Interface</td>
</tr>
<tr>
<td>Framework Libraries</td>
<td>IMA2.TLB</td>
<td>(IMA2 Base) - Base Interfaces</td>
</tr>
<tr>
<td></td>
<td>IMA2_Bindings.DLL</td>
<td>(IMA2 Base) - Bindings Collection</td>
</tr>
<tr>
<td>Framework Support</td>
<td>IMA2_Helper.DLL</td>
<td>(IMA2 Base) - Helper Library</td>
</tr>
<tr>
<td></td>
<td>IMA2_DADX.OCX</td>
<td>(IMA2 Controls) - DAD Helper Controls</td>
</tr>
<tr>
<td>Framework Services</td>
<td>IMA2_Locator.EXE</td>
<td>(IMA2 Base) - System Event Service</td>
</tr>
<tr>
<td></td>
<td>IMA2_EventService.EXE</td>
<td>(IMA2 Base) - System Information Service</td>
</tr>
<tr>
<td></td>
<td>IMA2_InfoService.EXE</td>
<td>(IMA2 Base) - System Locator</td>
</tr>
<tr>
<td>Included Components</td>
<td>IMA2_Agent.EXE</td>
<td>(IMA2 Components) - Agent</td>
</tr>
<tr>
<td></td>
<td>IMA2_StateMachine.DLL</td>
<td>(IMA2 Components) - State Machine</td>
</tr>
<tr>
<td>Included Managers</td>
<td>IMA2_StateMachine.OCX</td>
<td>(IMA2 Managers) - State Machine Manager</td>
</tr>
</tbody>
</table>

IMA2.TLB Core Interfaces

The IMA2.TLB file contains the core interfaces that define the distributing layer. Any piece of software wishing to interact with the IMA 2.5 system will need to reference this typelibrary to be able to interact with the
framework services to locate components. This section discusses each interface found in IMA2.TLB in detail, providing a functional summary as well as a method and property list for the interface.

**Structures:**

IMA2_DistributingInfo contains identification information used by the Locator, Information, and Event services, as well as the application layer. Every component must expose their distributing information structure, and most of the information to generate it is provided in the IComponent.Construct(…) call from the framework.

- *IsContainer* – Boolean flag indicating the component implements IContainer. Saves round-trips on QueryInterface.
- *Machine* – String name of the machine the component is running on (i.e. “MACHINE1”).
- *Name* – String name of the component (i.e. “Camera”).
- *Path* – String path of the component (i.e. “\Machine1\VisionAgent\Camera”).
- *PID* – Program identifier for the class (i.e. “IMA2_CameraLib.Camera”).
- *Process* – The ID of the process that the component is currently running in. This information can be useful for identifying shared process spaces, or selecting which process to shutdown through the task manager given a crash.
- *Thread* – The ID of the thread the component is currently running in.

IMA2_BindingInfo contains information about one binding published by the component. Every binding has a name associated with it (i.e. “InputVector”) and various IIDs or PIDs that restrict which types of components may be bound to it. If the IIDs and PIDs field are left blank, no restrictions are made about which component may be bound. The Path field indicates the path or paths of the components bound. A component may make a binding that accepts multiple targets by setting the BT_Collection flag on the BindType field.

- *BindType* – Specifies information about this binding, including whether multiple targets are allowed (collection status) and how the binding is cached.
• **IIDs** – A comma delimited string of IID s allowed for the binding. If this field is specified, the IID of any potential target is checked before assigning the binding.

• **Name** – The name of the binding, as seen by the user in the application layer.

• **Path** – The path of the currently bound component. May consist of a comma-delimited set of paths if the binding is a collection.

• **PIDs** – A list of PIDs that limit the types of component bound to. If empty, this criterion is not applied.

**Enumerations:**

*IMA2_BindingType* describes the properties of a specific binding. Multiple fields may be selected at a single time.

• **BT_Cached** – The binding is cached and not dereferenced each time. Should only be used in cases when speed is a top priority, as a loss in error handling is imposed.

• **BT_Collection** – Flag indicating this binding may have multiple targets. Useful for components such as engines, which may be required to interact with an unknown number of components.

• **BT_Default** – The default value for the field.

• **BT_Optional** – This binding is not required for the component to function properly or for the agent to be activated.

• **BT_Temporary** – This binding is only temporary.

*IMA2_LocatorAction* is used when specifying which action to take in calls to the ILocatorAdvanced.BatchOperation(…) function as well as in Locator event distribution.

• **LE_ActivateSeq** – A sequencer component was activated.

• **LE_Connect** – A new Locator was connected.

• **LE_DeactivateSeq** – A sequencer component was deactivated.

• **LE_Disconnect** – A new Locator was disconnected.
• LE_Locate – A component was Located.
• LE_Register – A component was registered.
• LE_Subscribe – An event client has subscribed.
• LE_Unregister – A component was unregistered.
• LE_Unsubscribe – An event client has unsubscribed.

IMA2_ContainerAction is used when specifying which action to take in calls to the ILocatorAdvanced.BatchOperation(…) function as well as in Locator event distribution.

• CE_Add – Indicates that a component should be added or registered.
• CE_Locate – Indicates that a component pointer is being Located.
• CE_Remove – Indicates that a component is being removed or unregistered.

**IComponent**

The IComponent interface must be implemented by any component wishing to work with the IMA2 distributing layer. In Visual Basic, the implementation can be automatically generated by using the IMA2 Component template, or by copying the contents of the C:\IMA2\Base Classes\ComponentImpl.cls into a new ActiveX DLL object. In Visual C++, the C:\IMA2\Base Classes\ComponentImpl.h file can be #included and then a new AT COM object class can be derived from the CComponentImpl base class. For more on how to do this, consult the Writing IMA2 Components chapter.

**Properties:**

- **IComponent.Bindings** returns a pointer to the IBindings interface on the component’s internal bindings collection. This pointer is often used by DAD to set and configure bindings.
- **IComponent.Distributing** returns an IMA2_DistributingInfo structure containing the essential distributing information about the component.
Methods:

*IComponent.Construct (Path As String, PID As String, pAL As ILocator)*

Called by DAD or IMA2 Console to construct the component. For convenience, the component’s appropriate Path, PID, and Locator are passed to it upon construction and should be used to complete the internal DistributingInfo structure. By default, the Construct method should be handled by an inherited base class or copied code – component developers should instead override the OnConstruct function to perform constructor and resource allocation code.

*IComponent.Destruct ()*

Called by the framework when the component is to be unregistered and removed from the network. Developers should override the OnDestruct event to handle resource deallocation.

*IComponent.DataSize () As Long*

Returns a long value indicating the number of bytes required to serialize (save) the component’s state. It is assumed the calculation may be computationally expensive, so it was not designed as a property. Developers should override the OnDataSize event to handle computation of this estimate.

*IComponent.Load (Data As Variant)*

Called by the framework to unseralize the component from storage in a DAD component file. The variant value contains whatever data was returned to the framework on a previous IComponent.Save() function call. During deserialization, the framework first calls the IComponent.Construct(…) function, then calls the IComponent.Load(…) function to complete the state restoration. Developers should override the OnLoad function to handle their component’s loading process.

*IComponent.Save () As Variant*

Called by the framework to serialize the component to binary storage in a file. The component must return a variant containing an automation type or an array of automation types that can be automatically marshaled to a file. Developers should override the OnSave function to store their components data, such as property values, internal state, and binding paths. A convenient way to implement this is as a binary array stored in the variant. Another approach is to use a Visual Basic PropertyBag object to store component data, then package its Contents property into a variant and return it for storage.
IContainer

The IMA 2.5 distributing layer identifies components that may contain other components by looking for the IContainer interface on them. This interface contains only one property and one method that are called by the framework applications (DAD, Console) to handle object serialization and child component creation. The only time developers should have to implement this interface is if they are designing a new Agent component or if they wish to make compound components such as IMA1 Relationships.

Properties:

InnateComponents returns a string array containing the names of the component that are innate to the container. Innate components are created and registered by the component during the Construction process. The framework will handle the deconstruction and unregistration of innate components, but not their deserialization, construction, or registration.

Methods:

CreateComponent (PID As String, Machine As String) As IComponent

Called by the framework to create a new child component, this function must simple instantiate a COM object of the given PID on the given machine. No other IMA-specific operations on the new COM object are required on the part of the container.

IBindings

The IBindings interface is implemented by a bindings collection object. IMA 2.5 ships with a default bindings object provider, IMA2_Bindings.DLL. The IMA2_Bindings.Bindings object maintains an internal list of bindings for components. During construction, the provided IComponent implementation instantiates this Bindings object for the component. During the OnConstruct event, each component should publish its bindings on the Bindings object. The IBindings interface provides the functionality that the host component needs to work with its contained Bindings object. The interface also provides a pointer to a separate interface (IConfigureBindings), which is used for configuring the underlying bindings object.
Properties:

Collections returns the array of resolved IComponent pointers to the targets given the name of a bindings collection.

Pointers returns the IComponent pointer of the resolved target given the name of a binding.

Settings returns a pointer to the IConfigureBindings interface, which provides configuration properties and methods for the underlying bindings object.

IConfigureBindings

The IConfigureBindings interface is implemented by bindings collection objects to allow for setup and runtime configuration of binding information. This interface focuses on publishing bindings, not resolving them. The IConfigureBindings interface is typically used by the framework and application layer, as well as by the component developer during object construction to publish bindings.

Properties:

Count returns the number of bindings and binding collections currently supported by the component.

Item returns the appropriate IMA2_BindingInfo structure given the name of a binding.

Items returns the entire list of IMA2_BindingInfo structures for the component.

Methods:

Add publishes a binding by adding the structure to the internal list given an IMA2_BindingInfo structure.

Construct constructs the bindings object. This method is used by the framework to configure the bindings collection at startup given the path of the owner and its Locator pointer.

Destruct causes the bindings object to be destructed in preparation for component shutdown.

Remove revokes the binding by removing it from the internal list of IMA2_BindingInfos.

ILocator

The ILocator interface provides developers with the primary interface to the distributed component network. The key function of the Locator is to maintain a list of and provide pointers to all registered components.
on the LAN. The primary interface for retrieving a pointer to a component is ILocator, with more specialized functions handled by ILocatorAdvanced.

Properties:

*Machine* returns the machine that the Locator is currently running on.

Methods:

*Locate* returns the IComponent pointer to it given the path of a component. The path must be provided in the form of `\<Locator name>\<parent component names>\<component name>`. An example path would be “\Armstrong\Vision\Camera\FrameGrabber”

ILocatorAdvanced

Provides an interface to the Locator aimed at framework developers. Functionality includes batch operations to minimize round-trips, Locator configuration functions, and a method for retrieving all local component pointers at once.

Properties:

*Components* returns a string array containing the complete path list of locally registered components. This list is not sorted.

*Locators* returns a string array containing the paths of all remote Locators the local Locator is connected to. This list is not sorted.

Methods:

*BatchOperation* performs an action from the IMA2_LocatorAction enumeration using the arguments passed in Args variant array. Currently, this supports adding, removing, and locating a list of components in one operation. The result is a variant array, usually containing IComponent pointers.

*Connect* attempts to connect the Locator to a remote one specified on the specified machine.

*Disconnect* disconnects the Locator from a given remote Locator.
Register registers a component in the Locator, making it locatable on the network.

Unregister unregisters a component in the Locator, releasing its reference and removing it from the list of locatable components.

ILocatorEventProvider

This interface is used by objects wishing to subscribe for Locator events, such as registering or unregistering of components and remote locator configuration. The Locator and Event service are typically the only objects that implement this interface, though the framework and application layer accesses it frequently.

Properties:

Clients returns a string array of the keys used by clients to subscribe to this service—useful for debugging updating errors.

Methods:

Subscribe subscribes the provided interface for further Locator events given an ILocatorEvents interface pointer, a unique string key, and the types of Locator events desired. This function is used by AgentView controls that display the network and by the application layer.

Unsubscribe removes the interface provided earlier for events from the internal list given the key.

IDistributingInfoProvider

This interface is implemented by objects providing component network information, such as the IMA 2.5 InfoService. The interface provides essentially the same information provided by the ILocatorAdvanced services, but with a search feature and with more direct support for batch operations. This interface is geared for the InfoService and replicates much of the Locator data retrieval functionality to reduce the distributed processing load and fragility of the Locator service. In nearly all cases, developers should access Locator information through this interface to preserve system stability.
Properties:

Components returns a string array of all local component paths currently registered as cached by the provider from the main list in the Locator.

Count returns the number of components locally registered

Item returns a cached IMA2_DistributingInfo structure for a specified component path.

Items returns an array of all cached IMA2_DistributingInfo structure for locally registered components.

Locators returns a string array of machine names for all currently connected remote Locators.

Methods:

Search performs a search of cached IMA2_DistributingInfo structures and returns an array of them meeting the criteria. Currently this feature has not yet been implemented.

ILocatorEvents

This interface is implemented by clients wishing to receive notification of Locator events, such as component registration/unregistration, remote Locator connection/disconnection, etc. This interface is primarily implemented by the Info service and Event service to acquire the preliminary event information from the Locator service, then the events are relayed to other clients such as the application layer applications or network-aware components.

Methods:

OnRegistrationEvent is sent when a component or set of components is registered or unregistered. Included is the action taken and the IMA2_DistributingInfo structures for the affected components.

OnOtherEvent is sent when a non-registration event is broadcasted by the Locator, such as a remote Locator connect/disconnect, or a Locator shutdown.

IManager

The IManager interface provides the most fundamental layer of support for graphical display of component information. It is very important to reiterate that IMA 2.5 implementations should not be written as to de-
pend on having a GUI or monitor with interaction capabilities. The final product is a component-only implementa-
tion targeted at an autonomous or near-autonomous platform. The final product should not rely on man-
gers in any way, as there may be no input or output devices to pipe data to. IMA 2.5 managers are purely a
convenient debugging and configuration tool for developers.

Properties:

*Component* retrieves or sets the IComponent pointer that the manager will use to display information for.

Methods:

*Update* indicates the manager should update itself based on information in the component. This function is
designed to be used only by the application hosting the manager, not the component itself. IMA 2.5 uses a pure-
polling approach to manager updates to prevent system bottleneck and fragility issues associated with compo-
nent-driven data display.
CHAPTER VI

CONCLUSIONS

Clearly, a monolithic software architecture using a single technology will not bring us closer to [artificial intelligence]. In our work, we are exploring an incremental approach for developing intelligent autonomic systems—systems that have self-awareness and can reason about their internal components and state. Autonomic systems must adapt to environmental changes and strive to improve their performance over time. They must be robust and be able to routinely overcome internal component failures. Autonomic systems must interact and communicate with other systems in a heterogeneous computing infrastructure. Our approach to building autonomic systems is based on combining autonomous intelligent agents in a well-structured way. This approach mirrors the structure of the human brain wherein there are clearly defined, function-specific processing centers connected by forward and backward communication channels and adaptive feedback loops.


The version of IMA 2.5 presented in this work was designed and tested on the ISAC humanoid platform. From preliminary tests, it has lived up to its design goal of refining key features found in IMA 2.0 and IMA 1.0. The following sections of this chapter discuss lessons and insights taken from two years of development work on IMA 2.5, and a cumulative six years of development work on IMA. Many of these discussions are intended for future architecture-level developers, or those component developers wishing to broaden their understanding of larger issues within IMA development. Of the topics discussed, those sections addressing the lack of software engineering practices being applied to IMA projects are considered most important for senior developers and management.

Software Platform & Programming Paradigm

IMA’s software platform continues to improve with time in stability, performance, and usability. Currently, IMA is not as fully developed as iRobot’s Mobility [iRobot, 1998] or IBM’s Agent Building and Learning Environment (ABLE) [Bigus et al., 2002]. IMA does have its advantages, however. Both Mobility and ABLE are aimed at experienced developers with a strong C++ or Java background. Whether switching to these technologies would benefit development in the long term is unclear. In the short term, a transition to them would be time-consuming; particularly for the ISAC group. The mobile group, with its lower level of project reuse and integration, could be an ideal candidate for trying these new tools. Preliminary work with Mobility on
the mobile robots indicates that ABLE may be a better choice, as Mobility suffers from poor documentation, support, and a Linux/C++ interface. ABLE provides IBM AlphaWorks (a developer network) documentation, support, and works with Java. Alternatively, rebuilding of IMA on top of a Microsoft’s .NET platform [Richter, 2002] would also be a course of action; one that could solve remaining issues such as COM versioning, the need for a common language-independent interface, and eliminating the many workarounds required by using DCOM.

Alternatively, simply abandoning IMA for small-scale robot projects is also an option. In my opinion, IMA does not provide any significant advantage to spot solutions (e.g. experimenting with an actuator control algorithm or researching a navigation algorithm), nor does it aid software development projects where requirements and specifications are clearly defined. Still, abandoning IMA for small projects entails dangers. Projects that are just large enough to need software reuse will find themselves regularly reinventing the wheel without training in software engineering principles or an architecture to enforce them. Even in small projects, knowledge transfer during developer transitions would be worse if IMA was not used to aid in reusability. For larger projects, such as the ISAC humanoid, abandoning a software architecture is not recommended. For systems such as these that require significant vertical integration and extensive exploration of the solution space, IMA does provide tangible benefits. Scalability, distributed processing, binary reuse, runtime reconfiguration, and dynamic programming are not easy to implement in a monolithic program, especially one that needs to change regularly. It is appropriate to say that IMA currently provides more advantages than disadvantages for ISAC-level projects.

IMA’s programming paradigm is in a more troubled state than its software platform, however. Whether this is due to the fading integrative robotics community and the agent/reactive/behavior-based paradigm as a whole, or if it is due to the IMA-specific implementation is unclear. Certainly, the IMA programming paradigm is still lacking a formal underpinning and significant usability refinements. There have been additions and deletions of key concepts from the original work (such as Policies and Relationships) by past developers, and no active developers claim to understand the IMA paradigm’s original scope. Whether the overall integrative approach to programming intelligent systems will continue into the next century is, in my opinion, doubtful. Considering the past decade of results at our own laboratory and the similar results of MIT’s Cog project,
new programming paradigms and functionality integration approaches need to be investigated. Currently, the most promising are soft computing, genetic, and developmental approaches.

**Why IMA Programming Is Difficult**

*Many of the classic problems of developing software products derive from this essential [conceptual] complexity and its nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, and schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From complexity of function comes the difficulty of invoking function, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors.*


There are many reasons, but the short answer to this question is provided in this section. To start off with, distributed concurrent programming alone is difficult. Event-based programming, fault tolerant programming, network programming, and object-oriented programming are also difficult. Remarkably, these are only the elementary principles involved in programming with the IMA software platform. Later on, IMA programmers must apply behavior-based robotics, agent-based systems, reactive-control, and subsumption architectures to build on the elementary software platform principles. Once the integrative robotics approaches are understood, developers must master IMA programming paradigm to bring these approaches together, and learn how to design their software to promote scalability and reuse. At this point, we can say that doing this type of programming at high quality levels would require nothing less than very seasoned programmers. However, the primary developers on IMA projects are not seasoned programmers. They are often university graduate students from diverse fields that lack a formal computer science background or even substantial programming experience. Add to this situation a lack of sound software engineering principles, lack of documentation, the moving target of research funding, and high developer turnover, and a picture of how difficult integrative robotics research has really become. IMA does not actually complicate these matters. In fact, IMA actually alleviates the situation by making the elementary development principles as transparent as possible. Ultimately, the effectiveness of IMA is not limited by the difficult of distributed concurrent programming, but by 1) problems inherent in the research of integrative robotics, 2) the university development and funding environment, and 3) the lack of a software engineering process within this environment.
Choosing COM

The Intelligent Machine Architecture is based on recent developments in component object software technology [...] Although it may have been faster to implement an initial system using traditional elements, a component object software technology was selected for the following reasons:

Strong Encapsulation - Access to component functionality is always through explicit interfaces with strong interface typing by globally unique identifiers.

Dynamic Linking - Components can be loaded and unloaded under explicit control, by unique identifier, during program execution. This allows much greater flexibility in software configuration than ever before.

Interface Abstraction and Definition - Interfaces can be defined using an abstract interface definition language (IDL) that aids in the generation of component code and helps separate the interface from implementation details.

Language Independence - Components can be implemented in any language that supports function pointers, such as C++, Visual Basic, or Java.

Version Control - By strong typing of components (by globally unique identifiers) it is possible to support multiple versions of components and smoothly add new components and versions to a working system.


Opponents of IMA have argued that COM was a poor choice upon which to base a software architecture. CORBA and other OOP-based approaches were viable options at the time IMA was developed, as they still are. In retrospect, we can say that Pack made a good choice but could not have anticipated some of the issues that would arise from using COM in our research environment. Of the reasons Pack cited, dynamic linking and language independence proved to be the two most useful features of COM to IMA. Programmers can still reconfigure their systems on the fly without wasting vast amounts of time on recompiling or reconfiguring other programs. As an added bonus, IMA systems can usually be kept online while being updated, fixed, or reconfigured. Even today, this is a hallmark advantage of IMA over other object-oriented approaches. Strong encapsulation also proved to be useful, but was difficult to implement well. Selection of the properties and methods of an object that should be exposed to other components and agents turned out to be a very complicated issue. Each time, it entails numerous conceptual, knowledge-representation, and performance tradeoffs. Interface abstraction and definition proved to be unspectacularly important (see Factors Affecting Scalability section), as most OOP methodologies implement these techniques. The COM runtime interface information system, however, proved to be very useful for the dynamic programming and runtime debugging features of IMA.
This leaves us with a discussion of version control. Of all COM’s features, the disaster of version control could not have been foreseen and is discussed in the next section. Overall, given the circumstances, COM was a good infrastructure choice for its time. Today, basing IMA on Microsoft’s .NET architecture or IBM’s ABLE infrastructure would probably be better alternatives from this perspective.

**COM Versioning**

COM’s versioning system slows development and introduces subtle bugs when not used in conjunction with proper software engineering techniques. The most common example of this in the IMA community is incompatible IIDs that lead to mysterious binding and object instantiation problems. This situation is usually found in three scenarios. In the first scenario, a developer is either still prototyping their object and thereby periodically causing drastic changes to their interface that requires incurring a change of IIDs. It becomes obvious that the need (imposed by IMA) to prototype COM binaries and thereby repeatedly incur the register/unregister process will tend to introduce bugs. The second scenario is where a developer’s code never reaches a final release stage so the code is written, left “as is” to be used in multiple parts of the system, then is later rewritten despite the need for the old version to still be supported. This scenario, which is quite common in a research environment, is thus very incompatible with COM’s black box approach if source code versioning techniques are not applied. The problems with this scenario are caused partly by VB’s cryptic version management system, partly by the complexity of explaining of GUID/CLSID/IID rules to domain experts, and partly by the lack of a proper prototype, design, release cycle for our COM objects.

The second scenario leading to IID problems is a developer’s code being reused, rewritten, or changed by other developers. This leads to multiple copies of the same project being passed around and compiled by other developers, resulting in everyone having their own custom version of the same project. While multiple versions of a COM server is acceptable if it is properly planned and designed within a software release cycle, it causes severe problems in a less disciplined environment. This is often the case in research, where multiple developers find it easier to rewrite someone else’s legacy code into a new component than to start from scratch or wait for the original author to update the code itself. COM and .NET’s versioning system will continue to cause problems until a software engineering life-cycle is adhered to prevent multiple concurrent releases, source code is properly managed by versioning software, and there is enough knowledge sharing to show developers what
concurrent development issues they may be facing. Until then, developers will continue to have problems with older code that, because it has been unused for a long time, expects one version of a COM object, when that source code has long since been changed. Visual Basic, in particular, is awful for this because it takes strict control over IIDS when, in our case, they would be better off being specified once during the perennial debug cycle.

DCOM Error Handling

A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.

- Leslie Lamport, MIT Lecture notes.

Our research environment, due to its high rate of change in system layout, has limited resources available for system planning, knowledge sharing, and versioning control. This results in a large number of relatively fragile COM objects being produced. When large numbers of these objects are connected together through DCOM, even if each object has very few bugs, the chance of an object crash in any part of the system increases proportionally to system size. The result is that the larger our agent network grows, the more connection difficulties we experience with our objects across machines; making our inter-object connections fragile. DCOM does not offer either sufficient support or transparent support for connection error handling, which is an unrealistic and reckless approach to designing software for our system. The fact that DCOM’s default approach to handling an object crash is with a “server timeout” dialog that freezes thread execution for nearly a minute makes control over the system unintuitive and inaccessible for the average developer. At the very minimum, fine-grained control over timeouts and intuitive support for asynchronous calls should be required for use with IMA. The only visible solution to this issue is to build this type of error support into the IMA system or move to a distributing protocol that provides support. From past experience, any attempt enforce safe coding principles will undoubtedly be poorly implemented, inefficient, or just plain ignored.
Security Considerations

**Question:** Why are there few or no tools to help with security, concurrency and synchronization issues and the like?

**Grady Booch:** We as an industry just don’t know the right ways to codify what we’re doing in the security space. … The problem of building secure systems is so hard that we have [people] to ascertain those policies, which also get pushed down to individual developers. Building secure systems is just a plain wicked topic insofar as we don’t know the patterns with which to follow them—that’s why it’s difficult to have tools.


Distributed concurrent dynamic programming is difficult enough to accomplish without having to worry about security issues. In IMA systems, these issues are brought upon us through the use of an underlying IP-based data transfer network and the DCOM / NT security model. While many issues have been raised about Windows security in the past, the overall security provided by Windows NT is imperfect, but certainly not poor. However, it is important that IMA developers be aware that every version of IMA essentially sidesteps the security issues at hand. For network configurations found in research scenarios (a subnetwork of a university network) this results in system security roughly equal to the overall security of the laboratory’s domain.

Hacking attempts at IMA systems can take two primary paths: 1) attempting to gain access to a valid user account to gain terminal access, and 2) attempting to impersonate a valid account (or its IP signature) and remotely creating DCOM objects to manipulate the component network. These primary hacking attempts can be blocked by university firewalls, and additional internal attempts are limited by the robot domain controller. In other deployment scenarios, however, the security considerations would likely need to be taken more seriously. This would be the case in military or domestic use scenarios.

As it stands, IMA 1.0 uses a simple logged-on model for security. In this model, only a computer belonging to the host domain, with a valid domain-user logged on, and with the proper IMA software installed can modify the component network. IMA1 systems are primarily prone to attack by direct manipulation at the machine terminals that must be logged on and accessible at all times. Additionally, individual users within the domain are not distinguished for levels of clearance. These are system-level issues that must be added to the list of standard DCOM issues of using no additional security checking for object instantiation checking.

In IMA 2.0 and 2.5, the security loophole of exposed terminals is removed through the use of a hidden IMA2 account to perform background object instantiations and hosting from remote terminals. This allows a
machine to host IMA2 components without a logged on terminal. Naturally, this account and other user accounts are still vulnerable according to the DCOM security implications of using domain-level security instead of client/user-level measures, but the modified model provides a somewhat higher level of security. Ideally, the recent revisions made to DCOM by Microsoft (COM+ and .NET) could be exploited in future versions to provide higher security for IMA 2.5 implementations. This support could also be coupled with architectural support for component network modification permissions.

**Language Support In Development Environments**

Language and development environment support is critical when a diverse developer community needs to be supported. Additionally, supporting multiple languages allows large software projects to leverage the strengths of each language to specific problems, such as Visual Basic for interface design, Java for cross-platform interoperability, and C++ for hardware interfaces. Within the academic environment, this becomes even more important since timelines for software are much smaller, expertise is much more limited, and documentation is nonexistent or out of date. While in some cases support for multiple languages can hurt productivity (if many languages or developers are involved), proper choice of languages can provide significant gains. IMA 2.5 primarily uses Visual Basic since its development environment and rapid prototyping abilities provide significant reduction in errors and debugging time over Visual C++. Additionally, Visual Basic provides gains in code maintenance due to its high-level statements and superior readability. Support for Visual C++, however, is important for low-level programming and for developers that would be more productive in such an environment.

**GME, UML, And Modeling**

_Modeling exists to help you manage, reason about, understand, and construct complex systems. The problem is that the complexity curve keeps growing. As even James Gosling [the creator of Java] has said, "for certain Java applications there are things that I just can't reason about or understand if I stare at the code itself." The same is true for C++, for any language._


The advantages of graphical modeling methods [Sztipanovits, 1997] and UML-style system decomposition [Booch et al., 1999] in software design and maintenance have been repeatedly shown in the computer in-
dustry. As system complexity increases, relying on code or language-based documentation for analysis becomes increasingly difficult and should be discouraged. Because of such situations arising in IMA development, an attempt was made early in 2000 to integrate the GME graphical modeling tool [Ledczeki, 2001] and its underlying UML-based system decomposition methods with IMA 2.0. The project was an attempt at leveraging the strengths of each tool on IMA software and agent development. Unfortunately, these initial attempts failed [Khaliq, 2002]. The reasons for this failure can be subdivided into the traditional causes of academic software project failures (not enough time, not enough people) and other, more situation-specific reasons. The most important of these situation-specific reasons are listed below and should be considered by future developers considering the integration of modeling tools into IMA:

1) IMA and GME/UML are targeted for fundamentally different software development environments. IMA was designed as a development platform to facilitate the rapid and prolonged software experimentation found in research robotics. GME and UML, in contrast, were designed for the more traditional commercial/industrial software engineering environment. In most IMA environments, radical analysis and design changes never stop. The component network may be completely different from hour to hour, and the development environment must therefore provide programmers flexibility and speed at all times in order to prove an effective tool for experimentation. The commercial/industrial development environment is completely different. In these environments, radical changes to software are few, far between, and typically limited to the specification and design phases. In these environments, software does not change while it is still being executed, nor do programmers routinely have to hot swap code in and out of the system. Attempting to fit tools designed for the relatively stable commercial/industrial software development environment into the typical IMA environment resulted in predictably poor results.

2) Process wise, IMA developers had to contend with another layer of user interface and constraints on their actions. Overall, enforcement of these constraints gained individual developers little, while requiring more significantly more task and cognitive overhead. Most IMA developers already knew which components were allowed in specific agents and simply wanted to be able to build their software as rapidly as possible. Because the majority of IMA developer time is spent making changes to a small area of the component network, model information about the system as a whole could not be advantageously exploited.
3) Computationally, these tools had to contend with COM object interfaces and DCOM agent designs that were stored on-disk as rapidly changing binary data, requiring a constant and expensive reparsing of system and object information on the fly. Because of the limitations of the COM typelibrary system, this data could not be provided history information, nor was change notification available. Oftentimes, the data was not even available during system debugging because of IDE limitations, ultimately leading to situations where the modeling tool could misrepresent the structure of the network and its components. Our attempts at modeling and validation based on this unstable knowledge base proved to be very problematic.

Potentially, GME and UML can provide elegant methods for validating and documenting systems designed by IMA programmers. Additionally, the gains from their integration into IMA development during system maintenance would likely increase proportionally with system size. However, in our experience, the overhead imposed by naively adding these tools to the run-time agent development and modification environment becomes unwieldy. If the best features of GME could be incorporated into DAD (or a separate piece of software) without hampering the IMA development process, the long-terms gains in software documentation, validation, and maintenance would be significant, within a more traditional software engineering environment.

Hidden Knowledge

The larger a system becomes, the harder it becomes to document and search through existing documentation. This brings us to the problem of hidden knowledge. IMA implementations are very complex and involve much more information to maintain than is often provided. To truly understand an agent, one needs to know which components it consists of, which agents it is designed to interact with, which bindings, properties, and methods will be invoked by other agents, the timing of these invocations, the proper initialization and configuration steps for the agents and components, how memory and resources are being managed by the components, which properties are for external use and which are for internal use, etc. The majority of this information is not captured in component documentation and is therefore called hidden knowledge. In fact, since the thesis stage of the academic software environment is considered the documentation stage, documentation is often not provided accurately when it is most needed. There is also system specific hidden knowledge, such as why a particular agent crashes under certain circumstances and techniques for managing certain issues when making or debugging agents. Ignorance of this hidden knowledge severely limits new development and maintenance of
Software, as well management’s abilities to make pertinent decisions. IMA 2.5 has an automatic documentation feature that lets agent developers share tidbits of information about components and agents, but this is only considered a start to solving the problem. As the size of the system grows, the hidden knowledge required to maintain the system dramatically increases; foreshadowing an array of problems with future knowledge transfer.

**Software Engineering Considerations**

*After two decades of largely unfulfilled promises about productivity and quality gains from applying new software methodologies and technologies, industry and government organizations are realizing that their fundamental problem is the inability to manage the software process.*


Although software engineering is not a silver bullet for achieving scalability and efficiency in software development, it is a proven method of increasing software quality and reusability [Schach, 2002]. Even more importantly, there is a wealth of empirical evidence indicating that sound software engineering principles significantly decrease the time and monetary costs of projects [Schach, 2002]. In some cases, these gains can range to an order of magnitude. For these reasons, we include a rudimentary analysis of the software development and software engineering environment used for implementing the core IMA 2.5 architecture, the IMA implementation of a humanoid robot (ISAC), and the IMA implementation of two mobile robots (Helpmate and Scooter). It is hoped that the information found in this analysis will provide a self-evident argument for instituting software engineering principles in future IMA projects. Readers are directed to “Object-oriented and Classical Software Engineering” by Steven R. Schach as a useful reference.

Before proposing that software engineering principles be applied to IMA development projects, we first establish that these projects show a degree of nontriviality that warrants the time and effort required to implement software engineering methodology. Ideally, this would be established through a cost-benefit analysis of shifting the development environment to a sound methodology, but this is left as future work. The first section of Table X shows the relatively large number of active software developers assigned to robot implementation work and the conspicuously small number of developers assigned to architectural work. Any professional software development organization would consider this situation remarkably dangerous, especially when their software architecture directly affects the productivity of all employees. Additionally, we see the large developer
to management ratio in our environment (~10:1) due to lack of an elected organizational model. The number of active developers alone reflects a non-trivial project, but the point is made even more strongly when the number of previous developers (developers who have left the organization, but whose code is still used) is considered. These previous developers represent large amounts of man-hours invested in software that must be leveraged for scalability and reuse in current IMA implementations. When considering project size in this situation, the total number of active and previous developers should be considered. There is a large amount of code written by non-computer scientists that have left the organization, and this code must be maintained by individuals who similarly lack knowledge of computer science. Clearly, software developed by 40+ domain-specific engineers or used by 100+ people should not be considered trivial. Even the smallest of these projects involves more than 100 source files, with the largest ones involving 1000+ and 4700+ source files.

Table 9. Software engineering environment for three IMA software projects.

<table>
<thead>
<tr>
<th>Workforce Composition</th>
<th>IMA 2.5 Architecture Implementation</th>
<th>Humanoid Robot Implementation</th>
<th>Mobile Robot Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Management</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Active Developers</td>
<td>1</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Developers with CS background</td>
<td>1</td>
<td>0</td>
<td>1?</td>
</tr>
<tr>
<td>Total Developers (1998-2003)</td>
<td>2</td>
<td>~30</td>
<td>~45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW / HW Environment</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages Used</td>
<td>Microsoft Visual C++</td>
<td>Microsoft Visual C++</td>
<td>Microsoft Visual Basic</td>
</tr>
<tr>
<td>Machines Used</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Robots Supported</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Development Environment</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Associated DLLs</td>
<td>68</td>
<td>254</td>
<td>649</td>
</tr>
<tr>
<td>Primary DLLs</td>
<td>21</td>
<td>65</td>
<td>73</td>
</tr>
<tr>
<td>Project Files</td>
<td>30</td>
<td>158</td>
<td>732</td>
</tr>
<tr>
<td>Source files</td>
<td>121</td>
<td>1103</td>
<td>4725</td>
</tr>
<tr>
<td>Documentation Files</td>
<td>21</td>
<td>125</td>
<td>84</td>
</tr>
<tr>
<td>Users Guide</td>
<td>Complete</td>
<td>Partial</td>
<td>Partial</td>
</tr>
</tbody>
</table>
Table 10 lists core details of the software engineering environment for the IMA architecture implementation (all versions), humanoid robot implementation, and mobile robot implementation at the Vanderbilt IRL. Clearly the most disturbing detail found in this table is the lack of any elected software life-cycle model or organizational structure. Additionally, despite the large code base described in the previous table, no Computer Aided Software Engineering tools (CASE tools) or development methodologies are in use. The complete lack of requirements, specifications, object-oriented analysis, and object-oriented design phases for these systems are also clearly evident. From their description in the table, the various IMA software projects must all be rated Capability Maturity Model Level 1 (CMM Level 1) [Software Engineering Institute, 2000]. This rating implies only the most primitive software management principles, or what is described by Schach as an “ad hoc process … [with] no sound software engineering management practices in place.” Indeed, the lack of any elected software life-cycle model has resulted in a default of IMA projects to the notorious build-and-fix model, where software is developed without planning, then frantically repaired for the duration of its lifetime. This model is infamous within the software industry for incurring large maintenance penalties and cost overruns. Schach characterizes build-and-fix as “fine for short programs that will not require any maintenance [but] totally unsatisfactory for nontrivial programs.” As we saw in the previous table, neither the IMA architecture, IMA humanoid implementation, or IMA mobile robot implementation should be considered trivial software. The fact that build-and-fix is being used for these projects is very alarming.
Table 10. Software engineering details for three IMA software projects.

<table>
<thead>
<tr>
<th>Software Engineering Practices</th>
<th>Architecture Implementation</th>
<th>Humanoid Robot Implementation</th>
<th>Mobile Robot Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability Maturity Model Level</td>
<td>Maturity Level 1</td>
<td>Maturity Level 1</td>
<td>Maturity Level 1</td>
</tr>
<tr>
<td>ISO-9000 Certification</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>CASE Tools</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Fault Database</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Documentation Database</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirements Phase</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Document</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Rapid Prototype</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specifications Phase</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Life-cycle Model</td>
<td>Build &amp; Fix</td>
<td>Build &amp; Fix</td>
<td>Build &amp; Fix</td>
</tr>
<tr>
<td>Organizational Model</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Project Management Plan</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Milestones</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Budget</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Deliverables</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Object-oriented Analysis</td>
<td>Some</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Specifications Testing</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design Phase</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-oriented Design</td>
<td>Some</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation &amp; Integration Phase</th>
<th>Undocumented</th>
<th>In Progress</th>
<th>In Progress</th>
</tr>
</thead>
</table>

| Maintenance Phase                 | In Progress  | In Progress | In Progress |

As previously described, none of the three key IMA projects have exceeded CMM Level 1. CMM levels (along with ISO 9000 certification) are the most widely accepted evaluation of software management practices. This emphasis on software management is especially important because of the empirically validated rule of software maintenance: the cost of fixing a bug in the maintenance phase of the software cycle is up to 350 times more expensive as fixing it in the specifications or design phases [Schach, 2002]. Only proper software management can enforce the proper software life-cycle models, organizational structure, documentation, and CASE tools needed to reduce maintenance costs. Simply put, critical or large-scale software projects require

63
careful management to achieve acceptable performance and maintenance goals. The likelihood of any vast software project being achieved successfully and efficiently without at least CMM Level 2 (basic project definition) or 3 (process definition) is unlikely. The idea of such software being maintainable over long periods of time without at least level 2 is also highly unlikely.

There are other related issues as well. Until sound software engineering principles are implemented by management and developers, attempts at performing project requirements and specifications analysis will be futile. Additionally, further object-oriented analysis and design will also not be possible without documents from the previous phases. The documentation and thought put into proper implementation of the software life-cycle provides huge gains to later development and maintenance. To reap these gains, organizations using IMA will need to seriously consider the software engineering literature and devise ways to implement its principles effectively within the academic environment. Once the IMA architecture implementation is updated using these proper software engineering principles, and carefully defined across phases of the software life-cycle, the majority of its unresolved issues will be exposed and made reparable. It is only at this point that the scalability desired from IMA 2.5 systems will be successfully achieved. If long-term integrative robotics research is to be seriously pursued, and if this research follows the reasonable assumption that the requisite software will not be trivial, then efforts must be made to manage the software process.

The IMA Architectural Software Process

IMA, as most academic research software, is not methodically engineered according to conventional software engineering principles. This is primarily because academic software is regularly used in feasibility analysis of future, professionally engineered government or commercial software. In many of these cases, academic software has a lifetime of less than 2 years and less than 30 users. In the rare cases where academic software is intended for prolonged use, it is often only for internal use within academic laboratories and rarely revised. The overall nature of academic software is, of course, a direct reflection of the limited time, man-power, expertise, and process management resources generally available to graduate students and professors. This is not necessarily bad if the scope of software being developed is reasonable for this development environment. Often, however, the scope and size of academic projects in integrative robotics is not reasonable.
Professional software, in contrast, is intended for prolonged use by large client organizations or customer bases. This situation requires application of a software life-cycle model (such as the stepwise refinement model) that clearly defines the order in which the product will be developed, as well as the requirements and quality controls required for each step. To produce the software using this process, teams of developers, software quality assurance specialists, documentation specialists, and managers are necessary. These costs are a direct reflection of the requirements of these systems. Professional software often has many orders of magnitude larger complexity, client base, and lifetime than academic software. Additionally, professional software has significant costs and penalties associated with system faults and poor design, because an organization’s productivity and profits can be directly affected.

The contrast between the requirements that inspire academic software development and professional software development point to a significant problem with IMA. The requirements made of IMA more closely match the needs of professional software products instead of academic software products. Therefore, an entire software development package such as IMA can be expected to encounter implementation problems if the academic software engineering approach (often characterized by the build-and-fix model) is applied. All versions of IMA show the symptoms of this miscategorization:

- Lack of an elected software development process model other than build-and-fix.
- Lack of clear requirements, specifications, and design phases.
- Lack of man power necessary to deliver and maintain a quality product.
- Lack of software process definitions and enforcement.
- Lack of long-term maintenance and CASE tools.

Even more troublesome is the simple observation that architectures such as IMA are critical to the long-term success of clients in the area of integrative robotics research. The field of robotics has been historically hampered by its lack of vertical integration [Engelberger, 2002], and software architectures such as IMA are the only tools we currently have to manage this problem. As long as architectures such as IMA remain a critical part of the system integration and software development infrastructure within an organization, users will suffer from the non-professional development of these professional-level products.
The Agent Development Software Process

The previous section addressed issues in the lack of software process used in the development of IMA. This section discusses issues in the software process of developers using IMA to write robot software. Many of these issues are found in the previous section. Primarily, IMA components and agents have potentially long lifetimes and large user bases, thus the application of academic research software development approaches is in conflict with the nature of the end product. Software scalability is a product of both the tools and processes available to the developers. It is the lack of software engineering processes that is currently hindering the effectiveness of the underlying IMA infrastructure. How useful is a system that can handle distributing thousands of objects over a hundred machines if the code always collapses under its own weight long before encountering infrastructure limits? How useful is the IMA programming paradigm if the life-cycle of developed software isn’t managed to ensure quality, accountability, documentation, and proper integration? This section is not meant to demean IMA developers or management, but to call attention to a conflict of interest in the requirements of the product and the cost of not implementing the proper processes to meet those requirements. In fact, it may be more effective to abolish IMA and invest the resources into process management with OOP than to attempt to use IMA without that process. To use an analogy, a good hammer and a set of nails may be sufficient to build a dog house, but they will never yield a skyscraper.

Measuring IMA Software Complexity

If IMA software is not proven to scale to a higher degree than other conventional approaches, it will ultimately fail to gain acceptance in the wider robotics community. This issue generalizes to any robot software architecture that hopes to achieve the high levels of software integration needed to control an intelligent machine. For these reasons, it is critical that an effort be made to design and perform quantitative measures of IMA’s scalability. We can begin to approach this problem through the fact that the scalability afforded by a software architecture is directly related to the complexity of its generated software. If resulting software is not very complex, then it follows that extending that software will not be too difficult (the software will be scalable). On the other hand, if the resulting software is quite complex, it can be safely said that extending the software will be difficult (the software will not be scalable). Measuring IMA’s generated software complexity is thereby important to measuring its scalability, and, in a sense, its relevance.
This brings us to the remarkably stagnant issue of how to measure software complexity. Countless measures of software complexity have been proposed over the past 30 years, most stemming from the original approaches of McCabe's Cyclomatic Complexity [McCabe, 1976], Halstead's Software Science [Halstead, 1977], and direct measurement of code size in KLOCs (thousand-lines-of-code). In fact, there are currently so many software metrics available that some researchers are using statistical methods to look for correlations between them [Lake, 2002]. To further complicate matters, there is also controversy whether any of these traditional software metrics can or should be applied to object-oriented systems [Tegarden, 1997]. Despite 30 years of research in the field of software metrics, there are two standing issues that must be considered. First, that there is still strong contention that almost all complexity measures are related to KLOCs [Schach, 2002], and second, that software’s complexity is quintessentially conceptual [Brooks, 1987; Mays, 1994]. While KLOCs is still considered the most robust measurement of software complexity, it provides little information about what aspects of code are the key contributors. Ultimately, after all, the designers of a software architecture would like to identify these principle contributors of complexity and minimize their effects.

In IMA development, there are currently a number of potential culprits: excessive or insufficient functional decomposition, poorly implemented object-oriented programming, poor module cohesion, high module coupling, excessive architectural constraints, or excessive conceptual constraints. Functional decomposition, for example, provides advantages when the connectivity between black boxes is kept low or ordered (cleanly hierarchicalized or encapsulated), but eventually leads to serious issues in cases of higher lateral and vertical connectivity. This effect can be addressed by attempting to maximize good cohesion (degree of interaction within a module) and minimize coupling (degree of interaction between modules), however it cannot be eliminated. Consider the antiquated telephone switchboards of the 1960’s. A naively designed switchboard with little rationale behind its organization (poor cohesion, poor coupling) is far less modifiable than a carefully designed switchboard which optimizes the distance between relevant and commonly used connections (good cohesion, good coupling). However, no amount of careful organization would allow this style of switchboard to scale to the traffic load of today’s telecommunication networks. In other words, while effective object-oriented programming and well planned functional decomposition can manage the complexity of software to a certain point, they cannot completely eliminate its rate of increase. The traditional way to manage this complexity is modularization. However, as we can see from the switchboard example, modularization offers rapidly diminishing
gains in systems that consist of many heavily interconnected modules. This is the type of connectivity found in IMA.

The items mentioned are only a few of the potential culprits contributing to the complexity of IMA software. Threading and fault tolerance issues also add entire dimensions to the complexity of component development and the resulting systems. It is hoped that future work in IMA will look carefully at measures for the potential contributors and attempt to provide foundations for their analysis and optimization. These efforts would be significantly aided by the advancement of the IMA software engineering environment to CMM level 3 (defined) equivalence or higher, where CASE tools could be applied to begin obtain code statistics.

Searchability, Constructability, And Scalability

An architecture’s primary purpose is to guide the development of software. To achieve this goal, IMA is confronted with three broad requirements: searchability, constructability, and scalability. The first requirement, searchability, refers to how efficiently (speed, utility of possible solutions, beneficial reduction of solution space) the software architecture allows researchers to investigate new solutions. An architecture that does not support reuse, reconfiguration, runtime debugging, or a GUI, for example, will provide lower searchability of the solution space. The second requirement, constructability, denotes how costly the architecture makes implementing a solution found within the solution space. An architecture that poses serious timing, maintainability, fault-tolerance, or performance problems will have a low constructability rating, despite possibly having a high searchability rating. Scalability refers to the additional cost imposed by the architecture on adding new layers of functionality to a solution. An architecture that creates many dependencies between modules, or does not address cohesion and coupling issues, will adversely affect scalability. Based on these requirements, we are confronted with the following questions when refining IMA:

- How can a software architecture provide means for beneficially downsampling and efficiently searching the solution space?
- How can an architecture reduce the cost of implementing a solution within the solution space?
- How can an architecture reduce the cost of adding new functionality to a solution, or maximize the gain from that addition.
Any new versions of IMA will need to systematically address these issues. Hopefully, future work in software architecture will provide a formal, quantitative basis upon which to examine them.

**Factors Affecting Scalability**

Robot software architectures were developed to solve the problem of scalability in integrative robotics. When standard approaches (functional decomposition, OOP) are used to add many layers of functionality, more and more dependencies are created across tiers. The usual result is a system that can be easily changed in some places, but incredibly hard to change in others. While this is acceptable in commercial software where the system is carefully designed at the beginning and is not expected to change drastically until release, in a software research environment these dependencies cause severe problems. In IMA this is particularly apparent with representations, where making a simple interface change to any of the core representations (despite their design to minimize coupling) would require a near system-wide rewrite and recompile. These dependencies present challenges not just at the code level, but at the conceptual, algorithmic, and binary levels. Despite many attempts by researchers across the world, however, robotic architectures have predominantly failed in their intended purpose. No modern robot currently shows any promising degree of scalability outside of its original design domain, despite a kaleidoscope of architectures and AI algorithms available. This situation begs a simple question. If the architectures for these robots are different, then what is the underlying reason scalability is not being realized? Why is it that these architectures provide some short-term gains in scalability, but have their benefits depleted exponentially as vertical or horizontal integration continues?

There are a couple of shared features that could be responsible. First, integrative robotics systems are never well defined before implementation. This puts software engineering approaches in the difficult situation of having to deal with the moving target problem. Second, integrative robotics presupposes that there is little immediate or long-term cost associated with adding new layers of functionality to existing software. Anyone with experience writing large robotic systems will say this is certainly inaccurate. Each programming language, software architecture, design decision, and function call can be thought of as imposing an additional, proportional cost to the scaling of software along a particular axis of functionality. Each layer of functionality can also

---

6 In terms of integrative robotics, vertical integration refers to hierarchical approaches such as subsumption and horizontal integration corresponds coarse-grained parallelism approaches such as agent networks.
be thought of as imposing a further cost on new layers being added, as well as a cost on modifications of existing functionality. These costs are largely dependent on how interconnected the functionalities are (how much operation of one piece of code affects another piece) and how the architecture manages, reduces, or isolates these dependencies. Third, all major software architectures in integrative robotics are based on structured programming, which opposes structural changes late in development. Experimentation in integrative robotics requires incremental structural changes late in the game (e.g. changing the parameters or function of low-level objects), and even properly implemented OOP makes these changes costly due to code refactoring. Newer programming methods, such as Aspect Oriented Programming (AOP), aid this problem by encapsulating the points at which changes are necessary, but are not a total solution as they do not eliminate the need for the changes.

The only way around the scalability issue is to have a complete specification, analysis, and design of the software desired from the start, so that costly changes do not need to be made later as development progresses. However, it is unlikely that this will be the case in integrative robotics as no such design is available. Indeed, to arrive at such a design, architectures that allow this level of exploration and experimentation will be needed, leaving us with a chicken-and-egg style paradox. Along these lines, there is a worst case scenario for scalability in integrative robotics research: that the vast number of lateral and vertical module interconnections found in biological control systems may be 1) necessary for high-level intelligence and 2) directly incompatible with the functional decomposition and fixed parameters required for software engineering. If this scenario holds, then soft computing techniques may be needed to handle integration and scalability issues.
REFERENCES


J. Engelberger, "Whatever became of robotics research?", Robotics Online, 2002.


A. Lake, C. Cook, “Use Of Factor Analysis To Develop OOP Software Complexity Metrics”, Computer Science Department, Oregon State University, 2002.


