INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Intelligent Multi-Agent Based Virtual Gauges (IMAGES)

January 12, 1999

Bashar Abdullah
Computer Science
Ottawa-Carleton Institute for Computer Science
Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada

A thesis
presented to the University of Ottawa
in partial fulfilment of the thesis requirement for the
degree of ***Master of Computer Science***

Table of Contents

Acknowledgments iii

Abstract IV

List of Figures VI

List of Tables IX

1. Introduction 1
1.1 Goals 1
1.1.1 Software Agents 7
1.1.2 Virtual Gauges 8
1.2 Client and Server Terminology 9
1.3 Thesis Structure 9

2. Agent-Based Software 11
2.1 Introduction 11
2.1.1 Fundamentals 13
2.1.2 Benefits 15
2.2 Multi-Agent Architecture 15
2.2.1 Agents in IMAGES 21

3. Virtual Gauges 25
3.1 Introduction 25
3.2 Agent Based Virtual Gauges 30
3.3 Areas of Application 33
3.3.1 Simulation Behavior Instrumentation 33
3.3.2 General Program Instrumentation 34

4. Intelligent Multi-Agent Based Virtual Gauges (IMAGES) . . 36
4.1 Benefits 36
4.2 Simulation Behavior Instrumentation 37
4.2.1 Measuring 37
4.2.2 Observing 38
4.2.3 Monitoring 39
4.3 Building Blocks of IMAGES 41
4.3.1 Client-side Application Interface 41
4.3.2 Server 42
4.3.3 Gauges 50

5. Integration of IMAGES With An Application 65
5.1 Requirements To Use IMAGES 65
5.2 IMAGES Interface 66
5.3 Integration Process 74
5.4 IMAGES Gauge Specification (IGS) Tool 75
5.5 An Integration Example 75

6. Conclusion 79
6.1 Work Achieved 79
6.2 Future Research 84

References 86

Appendices 91
A: Gauges of IMAGES A1
B: IMAGES User-System Interface B1
C: Integration Example With A Windows Application C1
D: Development Requirements D1
E: Glossary of Terms E1

Acknowledgments

Throughout the years of effort I have devoted in preparing, researching, restarting, reworking and finalizing my thesis, a number of people encouraged me to maintain a will to continue during immensely difficult times. To them, and the many more, too numerous to list here, who have shared in my concern to accomplish this very important goal in my life, I offer my heart felt thanks and acknowledgment. However, there are always those individuals who stand out, and are deserving of a personal mention.

First, to my thesis supervisor, Dr. Tuncer I. Ören, who maintained his encouragement of my work and constantly provided me with technical, professional and personal guidance; without your assistance, this work would not have been realized. To my brother, Raed Abdullah, my sister, Nadine Abdullah, and both my parents, who not only understood my personal sacrifices in accomplishing this task, but also assisted directly in completing it, by reviewing and editing this paper; I hope to be as supportive of your goals as your were of mine. To both my immediate and senior managers at Nortel, who accommodated my priorities with those of the company's; I thank you for your patience and understanding. And, finally, to my friends who withstood my inexplicable long hours in front of the computer terminal and innumerable hours researching for this thesis, instead of living life with them; I hope to spend as much time with you, to make up for those times.

Abstract

The surge of software agents has spurred interest in harnessing this technology with intelligent behavior. The empowerment of software agents attempts to relieve us of our mundane, laborious and often repetitive tasks. The aim of software agents is to offer intelligent automated assistance or guidance to users (Norman, 1994), machines, other software, or other software agents (Guha and Lenat, 1994), whenever requested and wherever employed.

Intelligent Multi-Agent Based Virtual Gauges (*IMAGES*) is a software tool which provides a highly modular, customizable and easy to integrate solution for automating and enhancing the instrumentation of simulation output generation. The *IMAGES* system design is based on a software agent architecture to promote ease of use without compromising functionality. The integration of software agent technology relieves the user from having to manage every aspect of the back-end interface of a simulation environment. Furthermore, by decoupling the client application (i.e. the simulation or application program), from its back end interface, we achieve a virtual integration that enhances modularity, facilitates maintenance and enables distributed observation and monitoring. Both of these characteristics, the software agent based architecture and its virtual observation and monitoring capabilities, define the goals of *IMAGES*.

Being native to the Microsoft Windows NT operating system, *IMAGES* takes advantage of the powerful advanced OLE communication facilities within the context of a true multi-tasking, multi-threaded, and graphical based operating system. Designed and implemented with object-oriented concepts from inception to completion, within the Microsoft Visual C++ development platform, the development and maintenance effort is minimized, thus facilitating future enhancements. Built on the base of the Windows 32-bit API, an industry standard for software development targeted at Microsoft Windows operating systems, future compatibility is assured. *IMAGES* represents a viable option for integrating state-of-the-art software instrumentation into existing simulation systems.

List of Figures

Figure 1-1 The Extended Model-View-Controller (MVC) Design Pattern for GUI-based Applications 6

Figure 2-2 The Agent Building Shell (ABS) layered architecture 16

Figure 2-3 The MIX agents and network relationship 19

Figure 2-4 The MIX network model layers 20

Figure 4-5 An example of a dial gauge type customized to represent a speedometer in units of kilometers per hour (km/h). 39

Figure 4-6 Gauge thresholds and range attributes. 39

Figure 4-7 Dynamic ranges and peak-water marks.. 40

Figure 4-8 The IMAGES main components. 41

Figure 4-9 The IMAGES server architecture and its supervisor agent. 43

Figure 4-10 The IMAGES components and their communication, object and agent relationships to the supervisor agent. 45

Figure 4-11 The IMAGES distributed supervisor agents and their potential connectivity across diverse private and public communication facilities for remote observation and monitoring. 48

Figure 5-12 Three stage process for the integration of IMAGES gauges into an M-Gest generated simulation program 76

Figure A-1 Sample application specific circular and arc dials. A2

Figure A-2 A sample of odometers. A2

Figure A-3 A sample of digital LED displays.. . . . . .A3

Figure A-4 A sample of toggle indicators (i.e. on/off). A3

Figure A-5 Both standard and user customized percentage indicators. A4

Figure A-6 A sample of horizontal and vertical bars. A4

Figure A-7 A sample of the plotter and its interactive controls. A5

Figure A-8 A sample of horizontal and vertical bar variations. A5

Figure B-1 The IGS tool user-interface showing two projects. B3

Figure B-2 The IGS tool preferences. B4

Figure B-3 The IGS tool listing the variables of a project. B5

Figure B-4 The IGS tool common variable configuration attributes. B6

Figure B-5 The IGS tool gauge interface attributes.. . . . . .B7

Figure B-6 The IGS tool range and thresholds attributes. B8

Figure B-7 The IGS tool warning indicators attributes. B9

Figure B-8 The IGS tool repository attribute group.. . . . . B10

Figure B-9 The IGS tool unit of measure conversion attributes. B11

Figure B-10 The IGS tool statistics calculations and equation editor. B12

Figure B-11 Mathematical notation for the mean of a measured variable. . . . B13

Figure B-12 IGS tool button sequence to specify the mean expression. B14

Figure B-13 The mean of a measured variable expression as it would appear in the IGS tool's statistics preview window. B14

Figure B-14 Mathematical notation of the expression for the average measurement over a measured interval. B14

Figure B-15 IGS tool button sequence to specify the expression for the average measurement over a measured interval. B15

Figure B-16 The average measurement over a time interval expression as it would appear in the IGS tool's statistics preview window. B15

Figure B-17 The IMAGES Server main user interface window. B16

Figure C-1 Client Application User Interface. C2

Figure C-2 The ImagesCI.h header file. C3

Figure C-3 Example file to include the ImagesCI.h header (highlighted). . . .C5

Figure C-4 Example uses of the ImagesGaugeUpdate() statements. C6

Figure C-5 Example use of the ImagesGaugeUpdate() statements C7

List of Tables

Table 2-1	The IMAGES agent performatives	22
Table 4-1	Example of the Component Management Table	46
Table 4-2	IMAGES Gauge Attributes	51
Table 5-1	IMAGES API - ImagesInterfaceOnOff	67
Table 5-2	IMAGES API - ImagesInitialize	68
Table 5-3	IMAGES API - ImagesGaugeUpdate	69
Table 5-4	IMAGES API - C/C++ header file	73
Table 5-5	IMAGES API - Borland Pascal uses and library statements	73
Table 5-6	IMAGES API - Visual Basic library declarations	73
Table 6-1	Software status	83
Table B-1	Unit of measure SI prefixes	B13

1. Introduction

1.1 Goals

This thesis presents the design and development of the ***Intelligent Multi-Agent Based Virtual Gauges*** (***IMAGES***) system. *IMAGES* incorporates software agents to facilitate the use of common visual instruments, referred to collectively as gauges. Gauges are usable in many applications, particularly simulation and their environment(s), that require the visual representation of a quantified measure or qualitative value.

Existing software gauges offer a minimum set of measuring and visualizing capabilities. Some gauges provide more sophisticated features, such as, storing of the measured values to many repository formats (e.g. eXcel, dBase, and ODBC), calculating basic statistics (e.g. mean and standard deviation), offering man-machine customizable interface options, and component-level integration within an application. These are the most common and consistent features available from commercially available software gauges.

Although these software solutions provide valuable benefits for measuring and visualizing data, they do so at a high cost to application development and maintenance. Contrary to claims by personal computer software industry vendors, component-based development is not the most

efficient means of developing an application. It is true that component-based development has improved the pace of software development by introducing component-level reuse. However, despite the incremental contributions of component technology to software reuse, several problems remain to be addressed. One problem is that any change made to a component's interface, whether harnessed by an application or not, requires the client application to be recompiled. Another problem is that human-machine interface customizing is limited to the options chosen at design-time by the application software designer. Today, component-based applications load all the components into memory, whether or not the components are used. This is a waste of computer resources. As software applications increase in size, many features that are the source of this increase remain unused. Present day components require that they be integrated, and available, at the time the application is implemented. Furthermore, today's components must exist at the time the application is executed, and must be located on the same physical computer as the application. Though ActiveX[1] components provide a means for integrating components into hypertext markup language (HTML) files for access over the Internet, these components maintain these same two limitations. Another major shortcoming of ActiveX technology is its lack of support for dynamic run-time integration with an application. This limits the ability to define dynamically the functionality of an application at run-time. The ability would enable the user to define or tailor the application to their needs, by enabling the application to load only those software elements that are used, only when they are required. The benefits of dynamically defining an application at run-time are realized in decreased computer resource requirements, such as memory, and with smaller applications, decreased disk space.

Advanced and feature-rich gauges can be very complex for end users to manage manually in their specific software applications. General purpose gauges generally require more configuring to tailor them for a specific application. Domain specific gauges, on the other hand, by their very nature, have a niche or limited applicability. For example, an electrocardiogram gauge has a specific application that has no usefulness to a pilot who needs to measure and monitor altitude or cabin pressure. Given that general purpose gauges can require a significant effort to tailor them for specific applications, repetitive configuring of similar gauges for large or many projects

1. ActiveX is a component technology introduced by Microsoft Corporation to enable the integration of component software into Internet web pages. Refer to the Microsoft website (http://www.microsoft.com/activex/) for information, and resources relating to this technology.

becomes tedious, and costly. More important, to observe and monitor several gauges simultaneously can be laborious and difficult.

The difficulty in managing several gauges is due, in part, to the lack of automated tasks that can, and should be handled by the gauge's software, but are not. Three specific features are lacking or are inadequate in existing gauges. The first feature is the method used by gauges to detect and react to measured values that are beyond the configured limits. Typically, a measured value that exceeds a preset limit is simply ignored. More advanced gauges either, assume the value to be equivalent to the limit, display up to the limit but record the actual value, or issue an out-of-range warning message. These approaches are unsatisfactory for applications that have no known or predictable limits. For these applications, such as in simulation, a more useful method of handling an exceedingly small or large value, is to dynamically adapt the range to accommodate the value.

The second feature, is the lack of a warning system that can inform a user of an impending breach of a threshold. Often, it is too late for a warning sign to be of use, if a threshold has already been reached or exceeded. This is akin to an oil gauge in a motor vehicle that only triggers a visual warning, perhaps by illuminating an indicator, after a critical threshold of fluid has been reached. It would be more useful to be warned of an impending critical event, rather than after the critical condition has occurred. Existing software gauges have only a single warning level, if any at all, and do not provide independent warnings for, nor distinguish between increases toward a threshold and decreases from a threshold. Also, no concept of threshold regions exists in which a region of low values, and a region of high values may both be critical. Both shortcomings require that a user constantly monitor each gauge for the possibility of a threshold breach. The implication of this is that the user is being distracted from the real problem they are trying solve.

The third lacking feature, is the ability to separate the gauge from the application, for enabling remote observing and monitoring. This hindrance may be imposed by the target operating system if it does not support, or it prevents inter-process communication, or the target operating environment if it does not support peer-to-peer communication. Another derivative of this shortcoming is the tight coupling of the application software with the gauge. The problem with statically coupling the application with the gauges, as has been stated earlier, is that any upgrade or modification to the gauge may require a costly reevaluation of the application software.

The latest JavaBeans (Sun, 1996) (JavaSoft, 1996) (Rodriguez, 1997) technology is an enhanced

Java architecture. It provides some of the solutions for the problems and shortcomings of its predecessor, Java, and those identified above. At the time this thesis was being written, the JavaBeans technology was in the process of being defined. The JavaBeans technology relies on an interpreter that is represented by a Java Virtual Machine (JVM). An advantage of an interpreted language is its potential ease of portability across diverse hardware platforms and operating system environments. However, this is also a disadvantage if the interpreter implementations are lacking, or dissimilar in their support of constructs. For example, the JVM provided by Microsoft Corporation is not compatible with Sun Microsystems' version, and thus a Java applet written for the Microsoft JVM may not execute on the Sun JVM. The ideal portability model that Java and JavaBeans have attempted to attain, in reality, has failed, primarily due to pure business and strategic technology interests. At this time, a major deterrent and disadvantage of Java technology is its poor performance, even on high-end personal computers[1]. This may be a short-term problem, considering that compiled JavaBeans components may become available. A new development introduced by JavaBeans is its support of distributed computing. This added support enables remote method invocation among both JVMs, and between beans and CORBA objects, and interoperation with remote databases. Interoperability between JavaBeans and existing component models is being addressed by bridges. Interoperability bridges would enable JavaBeans to exist within CORBA or COM containers, and interact with components based on these models. Currently, no such bridges exist. JavaBeans addresses some shortcomings of the original Java technology, but maintains some problems that are already addressed by *IMAGES*.

Beans are components developed with the characteristics of the JavaBeans technology. Although the *IMAGES* gauges could have been developed as beans, the absence of an ActiveX/OLE/COM bridge for beans prevented the use of beans in the *IMAGES* Server (i.e. an OLE/COM container). Some of the same difficulties encountered with other component-based technology (eg. Motif widgets, ActiveX and OCX) are not resolved by JavaBeans. The first is the complexity of the JavaBeans APIs, which range from trivial event-firing beans with no public methods or attributes, such as timer beans, to highly functional applications with numerous public methods and attributes, such as the GLG toolkit beans (Generic Logic, 1998). The second is that the JavaBeans development model is more complex than that for Java applets. This complexity adds to the cost

1. A high-end PC is considered to be a 400MHz Intel Pentium II, with 128Mb RAM.

of developing beans and hinders adoption of this technology. Finally, there remains the coupling of the application with the component. Without the bean(s), the application cannot execute.

IMAGES has three main goals. The first goal is to provide agent-based gauges that could relieve an end-user from both the managing and the monitoring of simulation output variables. The second goal is to design a highly modular object-oriented architecture that permits advancements in technology to be realized for future ease of maintenance. Finally, the third and most critical goal, is to ease its integration into existing applications, making it a viable advanced software enhancement. *IMAGES* is designed to achieve the integration of software agents into modular, highly customizable, and easy to integrate application independent graphical instruments.

IMAGES is a possibility in knowledge-based simulation experimentation or knowledge-based environments to process simulated behavior as elaborated in the taxonomy given by (Ören, 1994). As such, it is an advanced tool for enhancing the back-end interface of a simulation environment.

The *IMAGES* architecture is based on an extended Model-View-Controller (MVC) triad pattern (Gamma et al. 1995). A graphical representation of this pattern is illustrated in Figure 1-1. The separation of the model, which represents the status, data and behavior of an object in this design pattern, from both the view and controller of the same object, provides a modular and flexible architecture for GUI-based applications. The view provides the visual representation of the model, which may be in many forms and multiple concurrent representations. The controller enables the user to interact and modify the model indirectly, and the view directly. The controller may modify the view directly when the view represents a controller, such as a slider or state button, that has a direct correlation with the model. The *IMAGES* architecture is an extension of the MVC pattern for the following two reasons. The first is the abstract model on which the gauges are based. The abstract model represents the data and state of a gauge at an abstract class level, while each gauge-type has an extended model that more closely represents the model for its own particular view, that is the gauge interface. The abstraction of the gauge model, or the representation of a gauge's model in two forms, enables the reuse of the abstract model for any

type of gauge, and further improves and customizes the model for the specific gauge-type view.

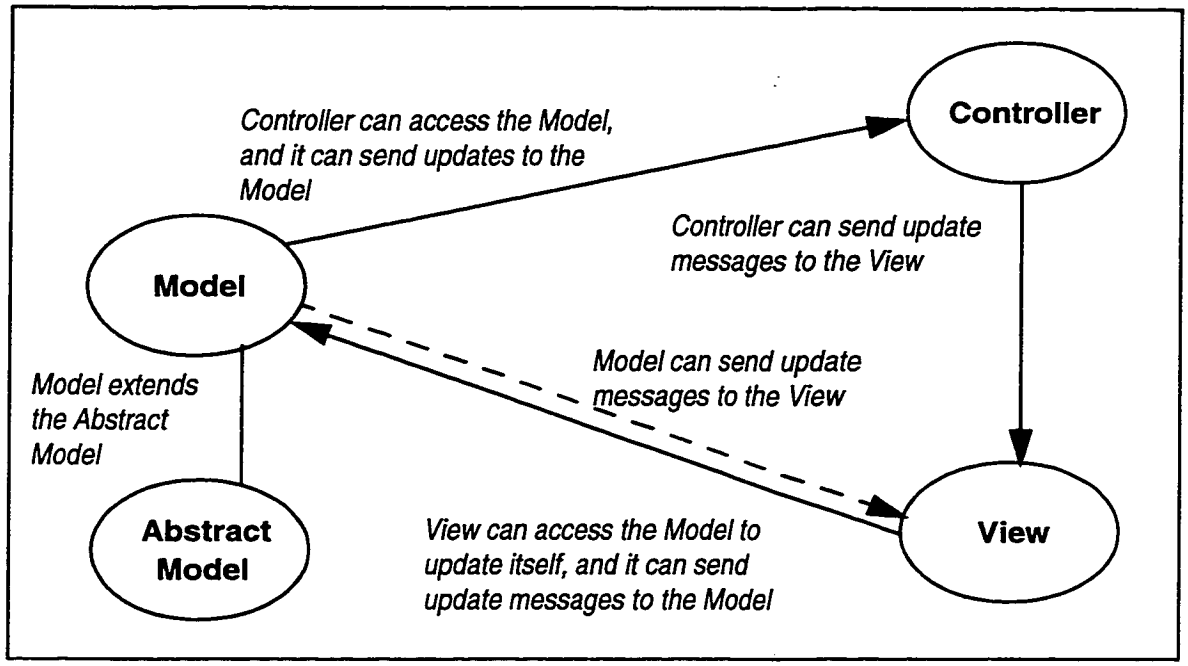


Figure 1-1 The Extended Model-View-Controller (MVC) Design Pattern for GUI-based Applications

The second is the distributed form of the MVC, where the model, view and controller are not executing in the same memory space. The *IMAGES* CS-API DLL represents one of three controllers for the gauge model. It is collocated with the client application, while the model it is manipulating is located in the gauge. The second controller is located in the *IMAGES* Server, which can control the behavior of the gauge. The third controller is located in the gauge. The view of the gauge is always a part of the gauge. The gauge model is mobile, in the sense that the initial status and data of the gauge are represented in the gauge configuration and repository files, which can be transmitted and shared among the *IMAGES* Servers. This enables the client application to be in one location and multiple concurrent views to be located in distributed locations. The distributed extension to the MVC design pattern provides greater flexibility and modularity for distributed GUI-based applications, such as *IMAGES*.

1.1.1 Software Agents

The term *software agent* represents a merger of several technical and social disciplines. Technologically, specialties that share integral contributions are the fields of software engineering emphasizing agent architectures, user interfaces and development environments, artificial intelligence focusing on distributed knowledge sharing and search techniques, dynamic scheduling, negotiation and conflict resolution, data communications and real-time systems design. Despite the heterogeneous technological contributions to software agent technology, many critical questions cannot be addressed by technology alone. Without anthropomorphizing the social aspects of software agents, also called social agencies (Sichman, 1995), comparable attention is required on issues such as agent cooperation, belief-desire-intent (BDI) states (Castelfranchi, 1995) and (Rao and Georgeff, 1995), multi-agent negotiation which may involve planning, collaboration, bidding and commitments, and a highly volatile issue of rights (Krogh, 1996). Clearly, software agent development presupposes a technical solution, nevertheless, these latter points demand a myriad of non-technical, specifically psychological, and sociological perspectives for resolution.

A human agent, in whatever form or purpose, can represent another individual, group or organization in pursuit of their interests. Guided by their interests, goals or objectives are defined, and the means for achieving them are assessed and planned for feasibility, cost and constraints. Agents may be hindered or prevented from attaining a goal by any number of technical, social, or environmental reasons such as resource unavailability or limitations (Numaoka, 1995), laws, rights of other agents, security, and definitely a changing environment or even a change in the goal itself. Implicitly, agents must adapt to their dynamic contexts in which their existence may or may not have an impact on the environmental changes. Adaptation may also take place, not only to the environment but to the individual or organization the agent represents. The agent may also acquire knowledge, either directly from more experienced agents (i.e. mentors), or via the observation of other agents as they progress toward a similar goal(s). However, mentorship alone may not be sufficient to progress towards the goal. These circumstances may demand knowledge or experience of another agent, resulting in a collaborative effort in producing results. Any time a goal is deemed unachievable by the agent, it may be abandoned. The responsibilities of, and demands on the agent are essential in successful pursuit of the interests of whom they represent. A corollary to this statement, is that it is in the best interest of an individual, group or organization to

delegate the task of achieving certain goals to an agent or agency.

As an example, I need only mention a professional athletic event to conjure up images of the type of agent[1] being expressed here; that is, an individual who represents an athlete. The prime goals of an agent in this context are playing time, money, and a guarantee for obtaining that money. In other words this effort is all in the best interest of the client. The agent also understands a priori the rules that govern negotiations, the forms of communication that are available within the given sports league, and the guidelines and constraints, such as deadlines and salary caps, that must be adhered to in pursuit of their goal(s). More experienced agents may be more efficient at attaining their goal because of their knowledge in tasking the most effective resources and taking more fruitful routes over others. As a result, less experienced agents may study these more experienced peers in an attempt to learn the strategies or logic in achieving similar goals, or they may rely on their own knowledge for producing results if precedence does not yet exist. Sometimes the rules and regulations that govern agent behavior, directly or indirectly, play a major role in the success of an agent in adapting to the dynamics of the league, such as the introduction of salary limits or mandatory drug testing. Also, the client often simply modifies their interest, resulting in goal changes or a direct change on the environment. Using a real world example, the reinstatement of Michael Jordan to the NBA played havoc with many existing players, especially those on the team he rejoined, resulting in another player (i.e. the client) being affected. That other player's agent would have to adapt to the changes, set new goals, and repeat the attempts in achieving the goal(s). These types of events are not exclusive to any one sports league. Representing athletes from multiple sports is also common for an agent, and as such they take advantage of similarities, and consult specialists on peculiarities. The two prime examples of this type of situation are those of Bo Jackson and Deion Sanders. They each played both baseball and football, and each had a single agent who consulted with more than one special agent for the respective leagues. If we consider the commercial aspects of professional sports, then most athletes may have more than one agent in quest of their interests.

1.1.2 Virtual Gauges

Virtual gauges are gauges in software with the added characteristic of being detached from the

1. In this particular example we are referring to the agent that would represent an athlete during contract negotiations. Indeed, any type of agent can be substituted with similar applicability, such as a travel agent, an elected official, a detective or a lawyer.

source of the measured value. Gauges are a common instrumentation tool used for visualizing a measured value. Familiar examples are thermometers, weight scales, electrocardiograms, voltmeters, speedometers, tachometers, and many others. These measuring tools are necessarily physical in nature, and are present in many variations of appearance, accuracy, range and application. With physical gauges, the unit of measure, the range and the variable to be measured are often fixed. Even in the realm of software, more often than not, measured variables are fixed at compile time, providing minimal flexibility in instrumentation of the application itself or the resultant behavior generation. The concept of a virtual gauge, initially presented in (Abdullah and Ören, 1995), allows for the determination of the measured variable that is to be instrumented at run-time. Not only can the variables be selected at run-time, but the attributes of the measured variable can be altered to adapt the visualization capability to the application being instrumented or simply the user's preferences. This dislocation of the visual instrumentation tool from the application gives a gauge its virtual characteristic (Abdullah and Ören, 1997).

1.2 Client and Server Terminology

The use of the terms *client* and *server* in the context of this thesis, are consistent with the terminology of DCOM (Redmond III, 1997) and OLE Automation (Microsoft Press, 1997). The client in the context of *IMAGES*, is used to identify the DCOM object container. The server in this same context, is used to identify the DCOM object, that provides the services to the container. Examples of a client (i.e. container) in *IMAGES* are the *IMAGES* client-side API dynamic link library (CS-API DLL), and the *IMAGES* Server. The *IMAGES* CS-API DLL acts as the data-source for the gauges. Examples of a server (i.e. object), are the *IMAGES* Server and the *IMAGES* gauges. The *IMAGES* Server is both a client and a server (i.e. both a container and an object).

1.3 Thesis Structure

As with any new technology introduction, its basis must be presented. Chapter two is dedicated to the academics of software agents and the architectures they necessitate. I describe the characteristics of software agents that they share with and differentiate them from transaction or interactive-based systems. The relatively simple uni-agent-based architectures that enable existing software systems to harness software agent technology will be assessed based on both the interface requirements and the benefits accrued from their integration. With agent multiplicity

being an inherent requirement for a practical solution, especially in a simulation environment, I present the architectural alternatives along with the issues introduced by multi-agent systems.

Chapter three introduces the idea of virtual gauges, and the role that software agents contribute in achieving this advanced software instrumentation concept. As a result of their general usability, virtual gauges may be harnessed in many applications. I will focus on the applicability to simulation model behavior generation, and general software, which may include spreadsheets, software development environments, and operating system utilities, to name a scarce few.

The sequent two chapters, chapter four and five, focus specifically on the *IMAGES* system. *IMAGES* gauges can perform many functions, particularly in simulation behavior generation instrumentation. The benefits and responsibilities of advanced instrumentation and the architectural components and their functionality within *IMAGES* that accomplish these tasks are discussed in chapter four.

Chapter five details how to integrate the virtual gauges into a simulation environment or for general application integration. As with any software, the requirements for integration of applications with *IMAGES* are presented, along with the *IMAGES* interface. Due to the numerous customizable features of *IMAGES* and specifically the gauges, a tool has been developed in consideration of the significant effort required to configure the gauges, especially when they exist in large numbers. A discussion of how the tool may be used to facilitate the configuration and integration of gauges into software applications is presented.

The knowledge and experience acquired in researching and developing advanced software instrumentation will be summarized in chapter six. Concluding remarks include the work achieved to date, and the persistent problems that represent promising potential for further research are outlined.

Appendix A illustrates the various gauges provided by *IMAGES*. As with any visual software, the user interface is key to successful utilization. Appendix B details *IMAGES*' user interface. An integration example is provided in Appendix C, demonstrating an integration example with a native Windows application. In Appendix D, the development requirements for using *IMAGES* are presented. Finally, the terminology used with respect to *IMAGES* and the technologies harnessed by it are provided in Appendix E, the glossary.

2. Agent-Based Software

2.1 Introduction

Autonomous and semi-autonomous software empowered with cognitive abilities can be developed to perform or assist with monotonous and complex operations; this is accomplished by harnessing in software the properties of human agents and their environments. Operations such as user interface management, information filtering, and schedule coordination, are just a few examples. Essentially, tasks that require advanced processing to produce a result that would otherwise require human effort, are candidates for enhancement with software agent technology. Three software agent solutions that exist are as follows: a single agent on its own, such as the email agent in (Maes, 1994); a single agent interacting with like entities, such as the interface agent described in (Maes, 1994); multi-agent-based architectures comprising heterogeneous agents forming social agencies, such as the visitorbot and userbot task specific agents presented in (Kautz et al, 1994), the agent tracking agent presented in (Tambe and Rosenbloom, 1996), and the multi-agent framework proposed by (Iglesias et al, 1996). Similar to their human counterpart, agencies may consist of specialized agents coordinating themselves, or coordinated by a supervisor agent, or even cooperating among each other to achieve a desired goal. Software agents can be thought of as artificial reproductions of human agents, however, with broader

application, yet specialized objectives. Given the cognitive characteristics of human agents, modeling software agents after their human counterparts is beneficial to developing advanced autonomous software.

Agent-based software benefits the users, and even more so, the software designers. An inherent characteristic of agent-based software is their modularity, and in some cases, their autonomy. These inherent traits promote reuse and modular architectures that ultimately result in improved quality and decreased maintenance costs, as has been espoused by many software engineering practitioners and authors. Pressman describes the factor of modularity as a measure of perceived complexity (Pressman, 1987). He further suggests that the means for reducing the effort necessary to solve a complex problem, is the application of the "divide-and-conquer" approach. By breaking down a problem into atomic constituents, the sum of the effort to address each constituent is less than if the original problem were addressed as one monolithic goal. Extending this philosophy to suggest that the atomic problems can be represented by software agent solutions would not be a far stretch of one's imagination. However, before jumping to a conclusion, it would be a shortcoming not to consider the issue of coupling. As I will present below, atomic solution coupling is addressed by the agent communication languages (ACLs) that enable software agents to interact with their environment, and in particular, other software agents. This communication mechanism provides for dynamic architectural coupling at run-time, an impossibility with most legacy software.

The critical issues of software agents are not resolved in this thesis. *IMAGES* makes use of software agent technology to offer an advanced software instrumentation tool. However, it is acknowledged that both competence and trust (Maes, 1994), two critical aspects of software agents, are the *Achilles* heel of any agent-based software solution. An agent is competent if it can automate tasks that a human user would ordinarily be required to do. An agent can be trusted if it can do the tasks as the human would expect. Trust, as suggested by (Maes, 1994), is developed by learning from the user. If the user cannot trust the actions or suggestions of a software agent, the benefits of using the agent are diminished or completely lost. The focus of this chapter is to present some background on software agents, and conclude with the key considerations of the software agents in *IMAGES*.

2.1.1 Fundamentals

The term anthropomorphize means to associate human characteristics with an entity; the entity can be an object, an animal, or a plant. By anthropomorphizing software agents at this point, I can insist that the characteristics of their human counterpart be drawn upon to justify their properties. Two types of software exist: object and agent. Object software is passive; it does no more than process an input, and generate an output, therefore, it requires constant direction from a human user or other software. In contrast, agent software is active, in that it performs functions without user input or direction. However, agent software may begin in a passive state and acquire knowledge to rely progressively less on external direction. This is akin to human development, where babies acquire knowledge from their parents at birth, until they have enough knowledge to become independent. As humans age, they begin to share knowledge with each other. This definition of an agent is in contrast to other authors' broader definitions that include objects as agents, as suggested by (Luck and d'Inverno, 1995). I define an agent, whether human or android, as one that acts or has the power to act for another, to achieve a non a priori goal. Thus, I restrict my qualification for agency to only those agents that are active. The following list summarizes the agent relationship matrix:

1. a human agent represents a human
2. a human agent represents software
3. a software agent represents a human
4. a software agent represents other software

The first relationship is used as the basis for much of the characterization and modeling for software agents. Though I have presented an anecdote in Section 1.1.1, specific issues related to this type of agent relationship are not discussed in the remainder of this thesis.

The definition implies, not exclusively, that a human agent can act for an android (i.e. a software agent). Clearly this second relationship serves no beneficial purpose.

The two remaining relationships, three (3), and four (4), project the parallels that software agent technology is progressing to achieve. These two agent relationships hold the potential for propitious solutions. The third relationship, is a software agent that helps a human achieve his or her objectives by simplifying, or automating repetitive or elaborate tasks, or acting as their proxy. Examples of this type of relationship include the following: filtering of e-mail based on either-or a

combination of keywords, authors, or general domains, such as music and travel, or more specific domains, such as gothic architecture, and reconstructive retinal surgery; obtaining related information based on the author or subject in email (Krulwich, 1997); suggesting complementary music or goods that reflect a user's preference based on other users with similar habits or preferences (Burke et al, 1997), such as the Firefly agent[1]; also, helping a user in using a software application[2]. Prime application for proxy agents are, for example: in the exchange of goods or resources where negotiation or coordination of multiple resources (Youssefmir and Huberman, 1995) is essential, such as in electronic commerce (Andreoli et al, 1997), scheduling meetings (Sen, 1997), or arranging travel plans. I suggest that this be the primary objective of software agent technology and that the fourth relationship is a derivative of it.

The fourth (4) relationship, is a software agent that represents or interacts with other software agents. The scope of the interaction may involve exchange of beliefs, desires, and intentions (BDI). Beliefs are the information and state that the agent possesses about its environment, including itself and the user. Desires are its goals or objectives, which may not necessarily be feasible or attainable. Finally, intentions are its actions and commitments to carry out the actions, as expressed in the contract-net architecture. The use of the BDI concept to characterize software agents is elicited from their human counterpart. Examples of applications that can harness this form of relationship are truly limitless. Possible applications can include the following: contract negotiation, conflict resolution, auction bidding, electronic trade, and electronic banking.

One characteristic of agent-based software that distinguishes them from traditional software is the knowledge that agents acquire and possess. Software does not have agent characteristics if it does not have intelligent abilities. This may include characteristics such as the ability to: learn from its environment or its user(s), directly or indirectly, either for immediate or future decision making; to define its own goal(s); adapt to its environment by modifying its sub-goal(s) or plan(s) in achieving the goal; react to environmental stimuli; and plan, as some authors insist (Moffat and

---

1. The Firefly agent can be experienced at URL: http://www.firefly.com/
2. I am careful not to include the concept of *Wizards* that are predominant in today's PC software as examples of agent based software. This collection of software does not benefit a user once the *Wizard* has completed its assistance, and more importantly, it always requires that the user dictate the next action to take. *Wizards* are simply (better) thought-out scripts that define steps a user can follow to accomplish a task. The classification of these types of Wizards are incorrect. However, there are some *Wizards* that take initiative on behalf of the user, and offer suggestions, or take self directed action. One such example is the autocorrect feature of some newer wordprocessors.

Frijda, 1995). Software agents can have a goal devised either by themselves or by another entity. Their behavior may be reactive or goal-driven, or both (Tambe and Rosenbloom, 1996).

2.1.2 Benefits

In consideration of the properties of a human agent, we can extrapolate the benefits directly to software, and indirectly to the user of the software. The benefits can be measured by factors such as: simplification of complex tasks, minimization or elimination of redundant tasks that ordinarily require human interaction (Maes, 1994), and an increase in user productivity. The benefit to software is the reduced complexity resulting from increased modularity.

2.2 Multi-Agent Architecture

Software engineering promoted the use of function-level modularity and coupling during the nineteen eighties and prior. In the late eighties and through the nineteen nineties, object-level modularity and coupling, formed as dynamic binding, was achieved by using object-oriented programming (OOP) languages. Now, the next logical step is to achieve application-level modularity and coupling. Using ACLs as the basis for coupling modular objects or agents, software can become dynamically constructed at run-time, rather than at design or compile-time. More emphasis on both specification and design is required to define the objects and agents that do specific tasks, along with their communication facilities. This approach promotes reuse at the application level[1] rather than at the more detailed class or functional level. In doing so, generic architectures (Iglesias et al, 1994) can be devised to support architectural solutions dynamically (Fisher, 1995). The solutions could be defined by the presence or absence of particular (types of) agents.

Multi-agent architectures (MAs) are aimed at addressing the dynamic architecture issue. In doing so, it allows agents dynamically to join or retract themselves from a solution, either by itself or by user intervention. This simplifies the replacement or enhancement of any part of a solution without disturbing the remainder of the architectural components. It further lends itself to reduced design complexity.

Two examples of multi-agent software architectures have been developed and are presented here.

1. Application-level reuse assumes that we consider each software agent to be an application. In fact, by claiming that agents can be autonomous, it infers that they are necessarily disjoint from other software. They connect themselves to external entities by the communication facilities they support.

The first architecture is the Agent Building Shell (ABS) (Barbuceanu and Fox, 1996). This framework provides a means for building agencies by defining layers of: knowledge management; domain and self-representation via ontologies; cooperation and conflict management; and communication languages, coordination and legacy software integration. The importance of this framework is its ability to integrate with legacy or non-agent-based software. The layered ABS model is illustrated in Figure 2-2. The second architecture is MIX (Iglesias et al, 1996), which stands for Modular Integration of Connectionist and Symbolic Processing in Knowledge Based Systems. It is a framework for a hybrid of connectionist and symbolic software solutions. The hybridization details are not the focus of this thesis. However, the architectural framework is of particular interest, primarily due to its distributed characteristic, and objectives for modularity, and ease of integration.

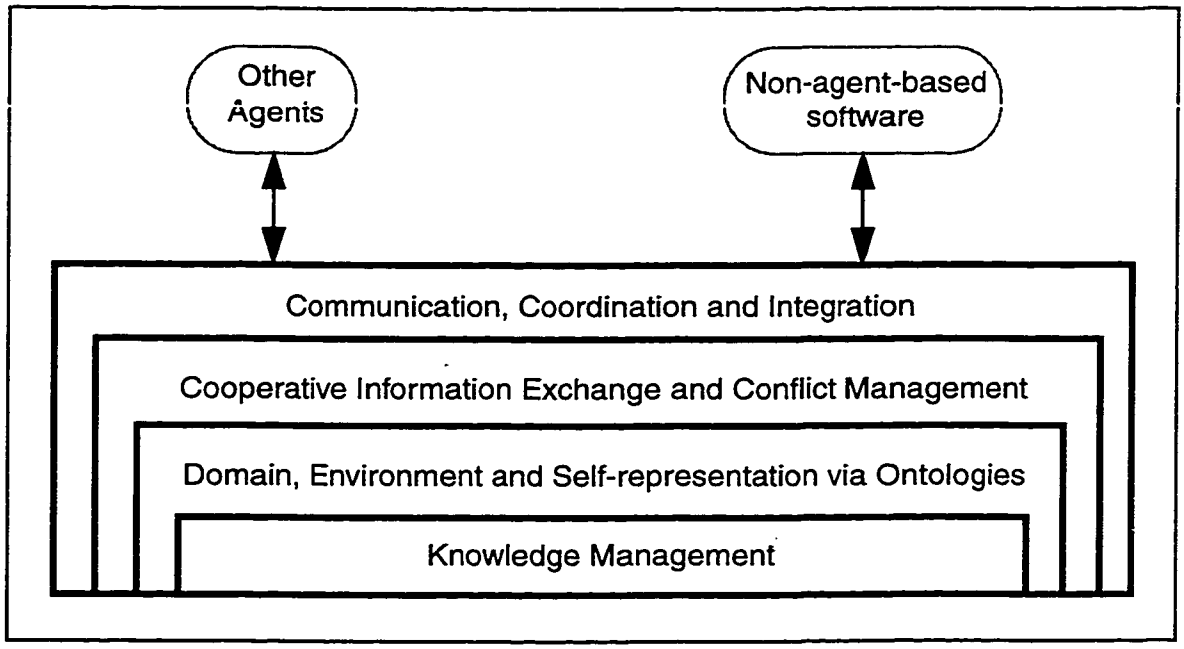


Figure 2-2 The Agent Building Shell (ABS) layered architecture

The Agent Building Shell (ABS) is a multi-agent framework with a primary goal of abstracting agents from their underlying environment. For example, the Internet comprises many technologies interconnected using many routed protocols. Each of the interconnected devices and

networks are themselves comprising potentially different technologies, both hardware and software. To enable inter-process communication for applications over the Internet, there needs to be a mechanism for making the communication independent of the platform on which the applications are executing. Furthermore, the communicating applications and agents need a consistent interpretation of data, and in this case knowledge. Though abstracting the underlying technology may be useful for interoperability at the agent level, an agent may also need to represent the state of its own environment and reflect this during communication with other agents. The ABS addresses four specific issues related to agent level abstraction. The first issue is agent interaction, which includes how agents communicate with each other, how they interpret each other's information, and how they interact in a cooperative environment. The second issue is knowledge representation and the effects on the representation when it is shared and thus open to change by different agents. The third issue is that of reasoning under different conditions, such as contradictory or skewed beliefs, particularly when it applies to an environment of shared knowledge and agents with different desires and intentions. Finally, the fourth issue addressed by ABS is that of integration with existing non-agent-based software.

Communication between the agent and other entities occurs through the communication, coordination and integration layer, which is represented by the COOL ACL (Barbuceanu and Fox, 1995). This layer also represents the interface for integrating agents with non-agent-based software. COOL is based on a superset of KQML (Finin et al, 1993), the knowledge query and manipulation language specification. COOL offers coordination constructs to handle coordination protocols. This is intended to enable two different agents to coordinate their activities toward achieving a common goal. The details of the coordination constructs are not relevant to the presentation of this thesis. It suffices to state that the purpose of providing a coordination mechanism is to enable goals to be reached with the knowledge and task specific capabilities of more than one software agent. For example, in an electronic commercial transaction system, a bank agent (e.g. a loan manager) and a customer (e.g. a manufacturing plant) would need to coordinate their behavior to enable the purchase of a product from a supplier. On the surface this may seem straightforward, however, in reality many problems may arise that would require attention by either, or both of, the bank agent and the customer. Two possible problems may arise when only part of the order can be filled, only some funds are available to complete the transaction, or the customer is deemed high risk for defaulting on loan payments. Some

negotiation and coordination would be required to complete or reject the request. The integration issue is resolved by either requiring the application to support a minimum subset of the ACL to exchange information with the agent, or develop the agent to use the application's API.

The cooperative information exchange and conflict management layer has two distinct responsibilities. The first is to exchange information between agents, and the second is to resolve contradictions implied by exchanged or shared knowledge. Information exchange allows an agent to communicate with other agents, based on the content of the information. For example, if a customer wanted to make a withdrawal request for money, the customer agent would have to make the request to a banking agent, as opposed to a store agent. The banking agent would have to ensure that it can satisfy the request. This layer provides the capabilities for both the request and proving that the request can be satisfied. Conflict management is addressed using the credibility/deniability model. The intent of conflict resolution is to deal with situations when contradictions in logic are inferred. For example, if two agents infer two separate beliefs to a third agent, and this third agent believes that the conjunction of both beliefs is false, a contradiction is derived. Now, the third agent must resolve which belief to infer, based on the relative credibility of the first two agents' beliefs, or the deniability of a belief. For example, the third agent may believe that credibility of the first agent's beliefs is greater than that of the second agent's. Furthermore, the cost of deniability of the beliefs helps determine which belief not to retract. A low level of credibility and high level of deniability would result in the belief being retracted in preference to the other. A high level of credibility and low level of deniability would result in the belief being inferred. The remaining two combinations may be insufficient for autonomous action, and would therefore require some user intervention. The derivation of the credibility and deniability factors are too detailed for this thesis, and are therefore not presented here. This second layer manages both the exchange of information and conflict management.

The ontology layer is used for defining the representation of the agent's domain, its environment, which includes models of other agents, and itself (i.e. its own state, beliefs, desires and intentions) within the environment. The authors of this model refer to the environment as "an organization." This layer gives an agent the ability to manage its role within an organization, by defining its behavior in four characteristics: what actions it takes based on organizational conditions, which coordination and communication protocols it uses, how it processes information shared with it,

and by sharing information about itself with the other agents in the organization. The authors call this "organizational modeling."

The innermost layer is the description logic for knowledge management. Again, the details of this layer are not pertinent to the goals of *IMAGES*. The knowledge representation and manipulation are based on a specific description logic language called MODEL, which supports a common set of concept learning operators.

The second architecture is a general purpose framework called MIX, intended primarily for hybrid multi-agent systems. The details of the connectionist and symbolic hybrid systems are not the focus of this thesis, and are therefore not discussed here. However, the architecture it defines is of interest. The primary goals of MIX are to achieve a high level of modularity, to support different levels of integration, and to ease the integration with software applications. Its framework is composed of two constituents, a network model, and an agent model. The relationship of these two constituents is illustrated in Figure 2-3. Any number of different agents can communicate with the network, and with each other via the network.

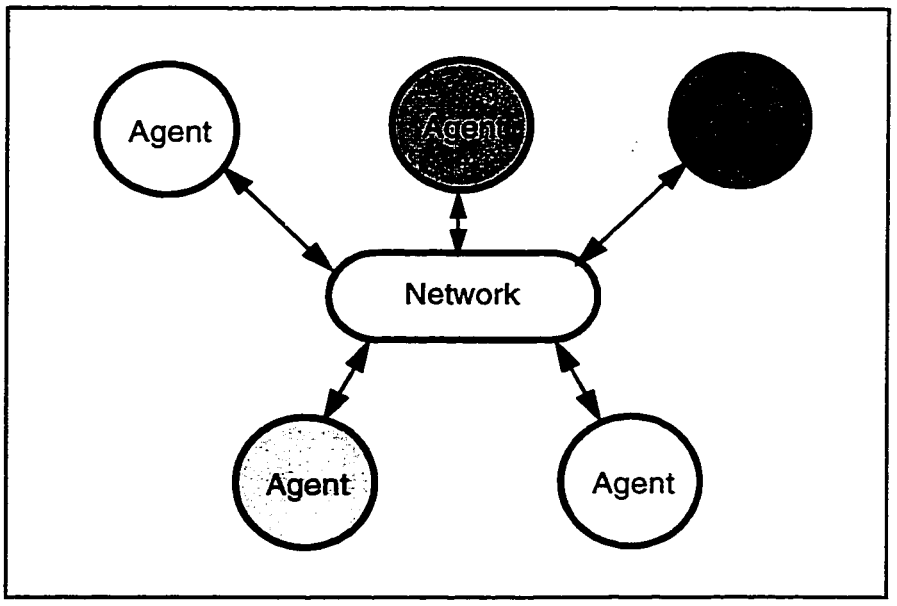


Figure 2-3 The MIX agents and network relationship

The network model consists of two types of agents, network agents and application agents. Network agents perform network specific functions, while application agents support agents that

communicate with the network and require specific services, such as a mechanism for communicating with other agents. The network model comprises three distinct layers, an interface layer, a message layer, and a transport layer. This layered model is depicted in Figure 2-4. The interface layer provides an abstraction of the network for the agents that communicate with it. It consists entirely of primitives that create, that process, that deliver and that receive messages. The message layer is composed of three distinct facilities, the network, coordination and knowledge facilities. Before these facilities are described in more detail, the final layer is the transport layer. This layer provides an abstraction of the communication medium for the network. This enables the transport layer to support any communication medium, such as dial-up ISDN, or direct LAN, and signal any protocol such as TCP, UDP, FTP or IMAP without any impact on the interface or behavior of the message layer.

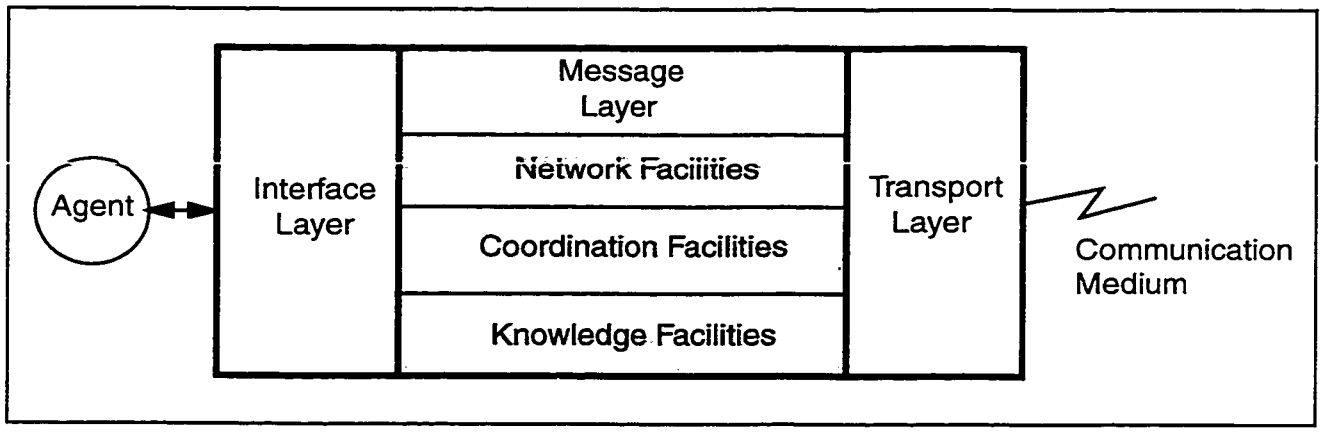


Figure 2-4 The MIX network model layers

The three facilities provided by the message layer are central to the network model. The network facilities provide the capabilities to manage and monitor the network environment, such as the availability of other agents. Some specific functionality offered by the network facilities includes the following: registration of the availability and capabilities of an agent in the network; registration of the set of capabilities an agent requires from other agents; and inter-agent features such as notification of the availability of other agents, and request and offering trades, by which one agent may request a service and another checks if it can fulfill the request. A special network agent is defined to offer these functions. The coordination facilities represent the communication

language(s) and coordination protocol(s). It defines the protocol between application agents within the network model. A contract net may be implemented as part of the coordination facilities layer to enable agent bidding on requests for services. The knowledge facilities support concept representation via ontologies. Its purpose is to abstract the knowledge representation from the rest of the architecture, and enable knowledge sharing using at a minimum three primitives: assert, retract and query.

Elaboration on the agent model is not necessary since agents can be distinct in purpose and behavior. However, different agents can be defined based on the agent model. By doing so, the agents could communicate with the network model via its interface layer. So, the only requirement for an agent based on the agent model is to support the network interface layer protocol for requesting and receiving services and exchanging information.

I have briefly presented both the ABS and the MIX frameworks by discussing the key elements of a multi-agent architecture: abstraction of domain specific knowledge; inter-agent communication with the functions and facilities required to support it; multi-agent coordination and cooperation with the functions and facilities required to support them; and, a centralized means of managing the agents, and representing and sharing their availability, capabilities, and requirements for services. These characteristics are the foundation of any multi-agent-based architecture.

2.2.1 Agents in *IMAGES*

The *IMAGES* system is based on a reactive multi-agent architecture, with a facilitator (i.e. a supervisor agent) responsible for agent coordination and management tasks. Refer to Figure 4-8, Figure 4-9, and Figure 4-10 for an illustration and description of the architecture. To achieve a high degree of modularity, the concept of agent wrappers is predominant for the major components. As such, the enhancement or replacement of any major component buffers other parts of the architecture from being affected.

This architecture presupposes an underlying protocol that enables the dynamic coupling of components within the architecture. Implementing a fully functional agent communication protocol, such as the Knowledge Query and Manipulation Language (KQML) (Finin et al, 1993), and the agent Coordination Language (COOL) (Barbuceanu and Fox, 1995), would require an inordinate amount of time for one individual to undertake. For this reason, I have chosen to define a sufficiently small set of performatives that achieve my intended agent-based goals for *IMAGES*.

These performatives are summarized in Table 2-1. The performatives devised for the *IMAGES*

Performative	Direction	Description
register(actionID, agentID, attributes)	agent→supervisor	This agent request induces the supervisor agent to assert the presence of the agent, which it identifies by its agentID. In addition to the unique identifier, the registering agent associates its own specific attributes, such as the actions it supports. This action must be followed with a confirm() response.
deregister(actionID, agentID)	agent↔supervisor	This request can originate from either an agent or the supervisor agent. In the case of the agent deregistering, it induces the supervisor agent to retract the presence of the agent identified by the agentID. In the other case, it permits the supervisor agent to inform the target agent of its intentions to retract its existence from the architecture, and that it must terminate its actions. This action must be followed with a confirm() response.
confirm(actionID, rxdActionID, agentID)	agent↔supervisor	This response is issued for actions that require some degree of commitment on the part of the recipient, to carry out the requested action. It is a statement of intent to carry out the action, on the part of the responding agent or supervisor.
reject(actionID, rxdActionID, agentID, reason)	agent↔supervisor	This response is issued for actions that require some degree of commitment on the part of the recipient, to carry out the requested action. It is a statement of intent *not* to carry out the action, on the part of the responding agent or supervisor.
tell(actionID, agentID, remoteAgentID, belief)	agent↔supervisor agent↔agent	This performative enables agents to share their beliefs with other agents. Agents or the supervisor that receive this message may assert the belief as fact. This message is used as the response to an ask() performative.

Performative	Direction	Description
ask(actionID, agentID, remoteAgentID, question, belief)	agent↔supervisor agent↔agent	This performative enables agents to request beliefs from other agents. Agents or the supervisor that issue this message may assert the tell() response as fact. This request may be used to validate a belief, or to obtain additional facts or rules.

Table 2-1 The *IMAGES* agent performatives

agents have an implicit trust associated with them. This is particularly evident with both the tell() and ask() performatives. The agent that receives the tell() message has to rely on the veracity of the agent that issues or responds to the message, that the belief is valid. For an agent to retract a fact or rule that it indicated in a previous tell(), it must issue another tell() message with a contradictory fact or rule. All of the *IMAGES* agents can understand and react to this entire set of performatives.

At a more detailed level, which I will not present in this paper, some messaging performatives require the specification of beliefs or questions for an action to be carried out. This requires that an information exchange protocol be defined, such as the Knowledge Interchange Format (KIF) (Genesereth and Fikes, 1992). Again, for feasibility reasons, a minimal protocol has been defined to address the exchange of information, which includes both data and knowledge as beliefs.

This set of performatives encompasses the generic set of actions an agent in the *IMAGES* system requires to execute its mandate. For example, the *IMAGES* network communication (NC) agent may be available, and could thus register() itself with the supervisor agent of the *IMAGES* server. The supervisor agent could then confirm() that it has asserted the NC agent into the architecture, and that it may proceed to carry out its tasks. If the supervisor agent receives a tell() message from the unit of measure conversion agent, this is a request to share with all supervisor agents, the existence of new unit of measure conversion knowledge. The supervisor agent would then issue a tell() message to the registered NC agent. The NC agent could then either reject() or confirm() the tell(). A reject may be due to any number of reasons, such as network communication problems or the inappropriateness of its current state. A confirm response from the NC agent is an indication that the NC has committed to broadcast the belief(s). Upon receiving a tell() from a remote NC agent, the steps are reversed, until the local unit of measure conversion agent receives the tell(). In

the final step, a confirm is not required since it is up to the agent to assert and confirm, or reject implicitly without responding. Similar examples could be derived for the other agents in *IMAGES*.

Agent knowledge in *IMAGES* is built up in a production-rule knowledgebase. This is achieved by harnessing the capabilities of the Eclipse data-driven production system that is based on the Rete Algorithm (Haley 1991). By embedding a production-rule inference engine into the *IMAGES* agents, the ability to formulate goals and update beliefs becomes possible. Using the Eclipse solution, allows *IMAGES* to offer these capabilities with two important benefits. The first, is the efficiency of Eclipse to handle modifications to a knowledgebase, which is a requirement for attaining critical monitoring constraints of client-applications. The second, is its ability to handle incremental volumes of knowledge effectively without compromising performance. This is a definite requirement for scalability of *IMAGES* to large systems.

3. Virtual Gauges

3.1 Introduction

A virtual gauge encompasses all of the functionality of a physical gauge in software, and more. For instance, it has the added benefit of not being bound to the application that it instruments. Two main identifying features of a virtual gauge, as the term implies, are that it exhibits gauge behavior, and that it is virtual. A typical gauge consists of at least three rudimentary parts: an input sensor, a transducer, an output display, and possibly a minimum of one input sensor that may require calibration against a standard value. The transducer facility, either mechanical, electrical or electronic, converts the input energy from one system into another, in the same form or another. The output is converted or mapped to a value that can be displayed, though it may require a scaling factor to adjust the value within the supported range. Visual representation of the measure includes all interface aspects, such as the unit of measure, and the indices within the supported range.

A virtual gauge includes these features with the addition of performing more sophisticated conversions, not only between the input signal and the output measure, but also between units of measure. For example, if the input value is measured in units of degrees Celsius, the displayed

value could be in degrees Fahrenheit or Kelvin. The input value could also be indexed against an application supplied time, or a system real-time clock. An entire interval of measure could be logged to a file for review later. A gauge is considered virtual if it can be made available when it is needed, and nonexistent when it is not, without modification to a program. By manifesting virtual characteristics, a gauge can be connected to any output source, such as a simulation model behavior generator, and maintain both basic and advanced gauge functionality.

Many benefits exist to using virtual gauges with an application. Since the application does not maintain the gauges, yet has only a line of communication with them, enhancements and modifications to the gauges can be kept hidden from the application. Virtual gauges are reusable in the sense that they are fully functional templates that can be plugged into an application with minimal development effort. The characteristic they draw on from the field of virtual reality is that their proximity to the application being instrumented need not be in the same location (i.e. the same computer or even the same network). The result of this detachment from the application is virtual gauges that can be adapted for multiple applications with little or no programming effort. Treated as black boxes, they also simplify the design and use of applications since the instrumentation is handled by the virtual gauges. From an engineering point of view, they can represent a solution to the growing size of applications by reducing the need for varied implementations that are essentially the same in function. Virtual gauges can also be used to present a consistent view and interface to the end user. This latter benefit has itself a hidden benefit of reducing training costs, and consequently a direct benefit of expedited return on investment.

I have presented here significant benefits in support of virtual gauges; however, as with most advantages there are counteracting disadvantages. Virtual gauges must provide significant functionality and flexibility to warrant existing (Hollan et al, 1987) and future applications for their beneficial and wide spread use. Existing applications already have tightly bound instrumentation tools or features. Usually, it is often difficult to justify replacing what is already functioning with something that claims to offer a "better" solution. The notorious industry adage, "don't fix what is not broken," supports this general claim. In this light, virtual gauges are not intended to be a replacement for what is proven to work effectively and efficiently. Instead, they represent the next stage in the evolution of software instrumentation. Evolving technology

inherently implies additions to, or improvements on existing technology or a new paradigm for accomplishing a task or solving a problem. Virtual gauges are no different in this respect. They offer a radical departure from the tightly integrated means of instrumenting applications, and simulation model behavior generation. The prime flexibility is achieved by postponing the definition and the behavioral characteristics of the gauge until run-time. This is made possible mainly by the separation of the gauges from the application. Even with the apparent contra-distinctions, their additional flexibility and functionality can advance the capabilities of instrumentation, although not for all applications.

Upgrading the virtual gauges should not affect the applications they instrument. This assumes that the interface between the gauges and applications does not change. A key design issue is to assure a simple, comprehensive, and sufficiently generic interface to accommodate existing and future needs. By ensuring that the interface is flexible and functional, the gauges can be enhanced without requiring modification to the already instrumented applications, while providing all applications with a path to harness new features. This abstraction reduces, if not eliminates, any differences between versions of the gauge software during migration of the application instrumentation facility (i.e. *IMAGES*).

As stated previously in Section 1.1, the second goal addressed by this thesis is the development of a robust, a modular, and a highly customizable means of instrumenting simulation model behavior generation.

A robust instrumentation system can handle extreme or unexpected conditions of variability in the measured value or representative quality. For example, a dial with a range of zero (0) to 200 would be undesirable for a variable that could potentially render a greater range of variance. At a more advanced level, a gauge should be capable of maintaining the most appropriate range for clearly representing the current value. At any level of sophistication, the ability to override the automated setting of the gauge must be available. This is to ensure that the user can maintain a factor of confidence and measure of control when interacting with the instrumentation tool. The ability of an instrumentation tool to handle unpredictable conditions enhances its chances of success in a broader range of applications.

Modularity is represented at three levels: functional, architectural, and in implementation. At each of these levels, modularity can be achieved in many ways and to varying degrees of component[1]

independence. At this stage, I am not particularly interested in implementation modularity and leave this level of detail to references, such as (Gray, 1994). Simulation models consist of several variables, and thus require many instances of the same or different type of gauge. The multiple instances of the gauges then demand that the individual gauge types be independent from each other and potentially from some supervisor application or server that mobilizes these gauges. Functionally, the gauges need to operate seamlessly with the application that is harnessing the virtual gauge capabilities. However, the functionality has to appear to be provided by the instrumented application and yet not be permanently tied to that application.

At the architectural level, the design options may be limited or even dictated by the functional modularity requirements. Functional modularity imposes a technical challenge by implication that gauge independence from the application with which it is functionally coupled, must have some dynamic mechanism for integration. Yet, architecturally, the system must present the appearance of operating as a single unit. Depending on the operating environment, the effort required to achieve both architectural and functional modularity may be unjustifiable. However, by making use of today's object-oriented (OO) and internet-driven languages such as C++ and Java, and architectural models and paradigms such as COM and CORBA, these two modularity requirements become realistically feasible.

The beneficial features that OO languages provide, which make an architecture with the modular criteria feasible, are: ease of adopting a formal development process for reuse, resulting in reduced development effort; object level modularity; and the ability to define object-level protocols between the objects in a system, so that two communicating objects need not be located together. This latter feature can potentially increase the complexity of a design by adding the requirement for protocol or task-specific protocols. However, this approach helps in defining modular architecture based on the concept of self-sufficient objects[2].

By basing an architecture on the CORBA reference model, and in particular the Object Request Broker (ORB), many benefits can be inherited. First, the protocol(s) that is (are) required to

---

1. A component in this general sense can be thought of as object instances within an application or to the extent of independent applications, interacting to arrive at a common goal. This is essentially the degree of variance being referred to.
2. In this thesis, I consider both an atomic class and a complex software agent as objects. Each, however, can be viewed as an object at different levels of system detail. Architecturally, a design is built of components/objects, each of which can be a software agent.

permit and enforce communication between objects, is (are) provided by the ORB, and the CORBA services and facilities. Second, the CORBA specification promotes a distributed architecture, for both heterogeneous and homogeneous environments. The environments include both the development and the operating environment. The CORBA reference model, by definition, is a modular architecture, whose intention is to develop a framework for objects to communicate with each other by means of requests for information, and replies with information. A full description of the CORBA specifications is dispersed among many OMG references. The main document is the ORB specification document, from which the additional references can be obtained (OMG, 1996).

IMAGES is based on the CORBA reference model, and has been developed using a language that supports the implementation of OO design. The OLE automation facilities of Windows – the operating environment for *IMAGES* – is CORBA compliant. The mechanism that is used by *IMAGES* to integrate the gauges with a client application is the OLE automation interface. The actual definition of the interface uses Microsoft's Object Definition Language (ODL) which is fully compliant with version 2.0 of the ORB specificaton, and is accepted by the OMG. However, ODL does not support all automation constructs. As such, during the implementation of the OLE automation interface, careful attention was made to minimize divergence from the supported constructs.

Attributes that can be customized and are adaptive include the following: dynamic ranges that intelligently adapt to fit measurements exceeding an initially set minimum and maximum range; audible warning indicators; and flexible and adaptable thresholds with support for both audible and visible warning indicators. Additionally, the virtual gauge can log and replay data, and calculate statistics on the data set. A single variable can have the option of being visualized in different forms, for example, as a circular dial or a simple LED display. The set of gauge types should not be limited by the architecture chosen; the types of gauges should be extendible. The gauge itself should be malleable to the user's needs, including operational and environmental demands. This suggests that a certain level of standardization with the operating environment procedures be maintained. However, this alone poses yet another difficulty in arriving at reconciliation between standardization, and the flexibility offered by customization.

Several instrumentation toolsets exist in varying degrees of sophistication and functionality. Some make claims of offering "virtual instrumentation," when they really mean software-based lifelike instrumentation (Hyperception, 1997). While others (ProtoView, 1997), including some with far more advanced visualization capabilities than *IMAGES* (MathWorks, 1997), still require the traditional coding effort to integrate instrumentation into an application or simulation program. The hidden costs remain: the effort required to train the users in the use of the many functions that achieve advanced visualization capabilities, and the difficult development and life-cycle constrained integration process. However, there are others (Taylor et al, 1994), that achieve a separation between the instrumentation toolset, which they term "concerns," and the client-application, which they term "concurrency." The first reference even goes as far as to define artists (software agents), that manage the visual interface for each function-specific gauge type, such as airspeed and compass artists. Inevitably, these solutions require that the user implement their own application interface to the instruments, which must also follow their more complex interface guidelines. By using virtual gauges, *IMAGES* achieves the separation of the instrumentation tools from the program it instruments.

3.2 Agent Based Virtual Gauges

Software agents are inherently modular, architecturally and functionally. Architecturally, a software agent can exist independent of any other like or unlike software agent, and more importantly from the application with which it is interacting. Functionally, a software agent can be self-sufficient. It can set its own goals and achieve them, with or without assistance or guidance from other agents, or even the user. This autonomy does not preclude an agent from being managed by other software or a user. These autonomous or semi-autonomous agents can adapt to their often dynamic environment. The modularity of software agents melds exceptionally well into the virtual gauge architecture.

The application of software agents to any software solution raises the question of need. Existing software solutions are a testament to the argument that solutions are achievable without software agent technology. However, software agent technology is not intended to rework existing software solutions to solve the same problem(s). Rather, the use of software agents to enhance a software solution, or in some cases to indeed solve a problem, is intended to further automate existing functionality that is mundane, repetitive and laborious.

Software agent efficacy enhances the virtual gauge with cognitive abilities. In doing so, many responsibilities related to the management and monitoring of gauges can be automated. The main benefit of automating such tasks as monitoring threshold breaches, and generating statistics calculations based on another engineer's experiences, is to enable the simulationist to concentrate on the simulation model, and not the tools required to deliver the information for analysis.

Three basic intelligent functions relevant to software instrumentation are as follows: acquiring new knowledge, adapting to the acquired or readily accessible knowledge, and applying the acquired or readily accessible knowledge. The behavior of virtual gauges demands some intelligent attributes for them to function in a dynamic environment. They must be capable of handling exceptions gracefully, and be able to interact with the user or other elements of the environment to resolve or avoid anomalies and unpredictable events. At the very least, they must be empowered to learn from the user how different situations should be handled so future occurrences can be handled autonomously. For example, if a circular dial is usually selected as the type of gauge to instrument a distance-over-time unit of measure (speed), then the virtual gauge system should be able automatically to default or select that gauge type for subsequent variables measured with distance-over-time.

Many aspects of virtual gauges could benefit from artificial intelligence-based processing (Sánchez et al, 1995). They include the following:

- automatic gauge type selection based on the variable's unit of measure;
- monitoring of the variable for thresholds and providing warnings;
- selection of visual warning indications, such as color, method and threshold(s);
- selection of audible warning indications, such as the wave, volume, threshold(s), and the means of issuing the warning(s);
- dynamic range adjustment;
- interface management, which requires knowledge of the other components on the screen and adjusting accordingly;
- unit of measure conversion between the input value and the displayed or logged value;
- statistics calculations; and,
- adaptation to the environment, such as memory, disk, and network resource availability, reliability, capabilities, and adaptation to the user's abilities, so advanced users need not be bur-

dened with unnecessary interaction and novice users can be willingly guided.

These varied aspects of software instrumentation could be enhanced by intelligence-based capabilities to the extent, over time, of being performed automatically for the user.

The knowledge acquired or accessed by one gauge or component may be shared among the gauges and their supervisory components. This is an important concept when cooperative goal processing is required, or a new component is trying to build its competence based on experience of other components. Cooperative or induced knowledge sharing is beneficial for such tasks as unit of measure conversion, when one gauge has knowledge of how to perform a conversion that other gauges have not yet experienced. Similarly, benefits are accrued in sharing, statistics calculations that have just been learned, and impediments resulting from resource limitations or unavailability that has been experienced. The benefits are as follows: decreased processing overhead by minimizing, if not eliminating, redundancy; and reducing time to completion in achieving a goal. There is an entire field of specialization related to distributed artificial intelligence (DAI) (Chaib-Draa et al, 1992). This thesis draws on some existing work in this field, but will not assess any particular topic. These means and benefits of shared and sharing knowledge play an influential role in a highly modular architecture, particularly for virtual gauges.

An inherent set of issues arises when distributed processing is addressed. Two factors are specifically relevant to virtual gauges. One factor is resource contention, such as for both volatile and nonvolatile storage space, and communication bandwidth, especially when the pattern of access or availability is non-deterministic. Another factor is resource availability, which may be either a lack of resources, such as non-volatile storage space, memory, time, and communication bandwidth, or simply a matter of insufficient privileges to access the resources. Some of these potential hindrances to achieving an agent's specific goal(s) may be used to build the agent's experience. In an ideal case, this experience or knowledge of problematic conditions would be shared, so that cooperation of many agents in either achieving or aborting the broader goal can be expedited.

Specific agent capabilities that could help a user in measuring, observing, and monitoring of simulation model behavior generation, are most potent when the instrumentation tools are truly virtual. That is, the gauges are architecturally separated from the client-application. This

introduces the issues with distributed processing that could be addressed by software agent solutions. In addition, the assistance of agents in setting up gauges, based on the experience of user associated gauges with unit-of-measures, can eventually relieve the user from having to decide. Furthermore, when unit-of-measure conversions, and statistics or data analysis calculations are added to the instrumentation toolset, agents can manage the sharing of this information with other users or other instrumentation toolsets. The prime benefit of enhancing virtual gauges with software agents is their ability to learn the habits and preferences of a user, and eventually make suggestions or completely automate the use and management of the gauges. This is particularly useful when a novice is presented with an environment that is complex. By making suggestions based on knowledge acquired from more experienced users, the adaptation of the new user becomes easier.

3.3 Areas of Application

The solutions for measuring, observing, and monitoring variables being proposed by this thesis may be applied to instrumentation of software programs in general. However, the solution for simulation behavior generation is a specialization of the *IMAGES* system to support both dependent and independent variables including multivariate statistics calculations. The possibilities for instrumenting non-simulation specific programs are too many to list here. Be it sufficient to say, though, that applications can range from software debuggers and basic progress indicators, to live electrical and medical monitoring systems, where the systems to be monitored are logistically remote from their monitoring stations.

3.3.1 Simulation Behavior Instrumentation

A prime application of virtual gauges is in simulation environments where the measuring, observing and monitoring of auxiliary, state and output variables is essential to understanding the behavior of a model. Not only are virtual gauges ideal for this task, but they can also be employed in generating statistics for performance comparisons, either between similar or dissimilar models, under varied initial and experimental conditions.

Beyond supporting extendible statistics calculations, *IMAGES* gauges support additional features not available in existing instrumentation systems (Taylor et al, 1994). One such feature is its support of both independent and dependent variables for simulation specific applications. The

dependent variables are measured and displayed based on a time parameter or on one experimental condition. The time can be either a simulation model simulated-time, or the host system-time. Another distinguishing feature that *IMAGES* supports is its ability to convert from one unit of measure to another. This is a very important notion, especially when a simulation model or application program is executed in one part of the world, and the observation of the variables occurs in another part of the world. The real benefit is in the ability to monitor certain variables from several sites simultaneously, and each site preferring a different unit of measure. This implies that the same unaltered simulation model or application program can be used in different locales. Similar to the extensibility of the statistics calculations, a user can also add their own units of measure and conversion formulae, if applicable.

Output data analysis can be accomplished by using a set of demonstrated effective statistical methods (Law and Kelton, 1991). Some statistical techniques promoted for their usefulness in solving real-world problems, are: confidence intervals, transient and steady-state distributions, means, and relative and percentage errors for stochastic processes. *IMAGES* provides these statistics calculations. In addition, *IMAGES* supports an extendible list of statistics calculations that can be exploited to generate data analysis results that assist a simulationist in deciding whether or not to apply Variance-Reduction Techniques (VRT). The intent of applying an appropriate VRT to a simulation, is to achieve convergence toward a simulation model with high performance, and a small confidence interval. Although suggesting automatically a particular VRT to use may be possible based on the analysis results, it is beyond the scope of this thesis. However, automating the statistics methods for analysis by a simulationist would simplify the process of choosing a VRT.

3.3.2 General Program Instrumentation

Although this thesis presents *IMAGES* with a bias toward simulation environment applications, it is hardly the only application. All of the benefits for simulation measuring, observing and monitoring can be harnessed for non-simulation specific applications. The most apparent are Supervisory Control and Data Acquisition (SCADA) solutions that constantly require real-time monitoring of system variables such as for electrical utility grids, network performance monitors, and medical unit vital sign observation stations. The common characteristic of these applications

is the potential for large distances between the systems to be monitored, and the location of the monitoring station.

However, not only are virtual gauges beneficial in those applications, but they can also be of assistance in local environments such as interfaces for debuggers, instruments for white and grey box testing of applications, and system performance characterization.

In these cases, the data that is to be visualized is live data. Under simulation conditions, the data may be postponed, depending on the circumstances, without adverse effects. The current implementation of *IMAGES* makes no claims as to the effectiveness of its live data support under congestion conditions. Further effort would be required to achieve reliability under most potential real world stress conditions.

Unlike a simulation environment, where the data types that are supported could possibly be limited to a select few basic types such as `float` or `double`, for general program instrumentation, applicability requires that all basic data types be supported. Sometimes, even this is insufficient. (Taylor et al, 1994) formulate a distinctive concept of abstract data types (ADTs) in the Chiron-1 project. *IMAGES* does not support the Chiron-1 concept of ADTs. It does, however, support all of the basic data types available in all the ANSI/ISO compliant C++, Pascal/Delphi and BASIC language compilers. Reference can be made to Appendix D for a detailed list of requirements for *IMAGES* integration development requirements.

4. Intelligent Multi-Agent Based Virtual Gauges (*IMAGES*)

The first goal of *Intelligent Multi-Agent Based Virtual Gauges* (*IMAGES*) is to help a simulationist in measuring, observing, and monitoring simulation output variables. Although I have emphasized *IMAGES* as a back-end interface enhancement for simulation environments, it is equally applicable to any Windows[1] computer program that requires the measuring, observing and monitoring of variables. This chapter describes the *IMAGES* system from the perspective of its benefits for simulation environments, its purpose in simulation instrumentation, and its main building blocks. The integration procedure is discussed in Chapter 5.

4.1 Benefits

Standing out among the many benefits of using *IMAGES* gauges in a simulation environment is the transparent and independent operation of the simulation program. This most beneficial characteristic is achieved by providing *IMAGES* as a self-contained executable application within the Windows operating environment, and the interface to *IMAGES* as a dynamic link library used by the simulation program. With an environment that allows for the operation of the simulation program independent of the back-end interface provided by *IMAGES*, focus can be maintained on

1. All references to *Windows* are implicit to both the Microsoft Windows 95 and Windows NT v4.0/x86 operating systems, except where indicated otherwise.

the simulation rather than the management of the model behavior results. This environment facilitates inter-application communication via the object-linking and embedding (OLE) protocol. The OLE facilities, under control of the Windows kernel, standardizes this inter-application communication protocol. The OLE client-side application programming interface (CS-API) is used by the user application, while the OLE server is another component of the *IMAGES* toolset. A user application maintains the client side of the interface via a maximum of three function calls. *IMAGES* on-the-other-hand provides the OLE client API as a dynamic link library (DLL). With the client interface integrated into the user application, and the server contained within *IMAGES*, this inter-application communication becomes the means by which transparent and independent operation of *IMAGES* is achieved.

The *IMAGES* architecture is designed as a modular basis from which future technology advancements can be harnessed with least effort. All the major sub-components within *IMAGES* are readily and transparently upgradable to the client application, and to the other subcomponents within *IMAGES* itself. All modifications to *IMAGES* would have no impact on the user application. Management and configuration of the gauges, which are dynamically linked to the user application, are handled by the *IMAGES* server. Thus, if the client application does not use the new functionality, there is no need to recompile the application. However, only if the enhancements are wanted and harnessed by the client application, would the client application require recompilation. A client application that does not need recompilation would still function, without exception, with a new enhanced *IMAGES* version as it did with a previous version. As each sub-component within the architecture is presented here, a demonstrated existence of this modularity will become apparent.

4.2 Simulation Behavior Instrumentation

The intent of the *IMAGES* gauges in a simulation environment is threefold: to measure the value of an output variable, to visualize the measured values of an output variable for observation, and to monitor the output variable for thresholds.

4.2.1 Measuring

Measuring a variable requires simply that its value be obtained from the client application. When using *IMAGES*, this is accomplished by enabling the client application to trigger updates to the

gauges. Gauge updates are achieved by placing `ImagesGaugeUpdate` statements at strategic locations within the application, or immediately proceeding any statement that modifies the variable's value, which results in an update to the appropriate *IMAGES* gauge.

Connecting a gauge to a specific variable is done by using either a variable list (VAR) file or an *IMAGES* Gauge Configuration (IGC) file. The VAR file can be a list of variable names, or additionally associated variable characteristics such as datatype, initial value, range, and unit of measure. The IGC file contains variables, their associated gauge configuration (see Section 5.4), and the corresponding gauge types. When the client application is started, as part of the initialization process, the gauges are created and logically connected to the variables based on the IGC file. If only the VAR file is available, the gauges are associated and configured by the gauge association agent. More details on the gauge association agent will be described in Section 4.3.2.3.

Beyond measuring variables, each gauge stores the values in a repository. The repository[1] comprises of configuration information for use during playback, and the measured values along with a time stamp as the primary key. The repository structure is project dependent, which simply means that the non-keys are the individual variables. Since each project is application specific, two projects may contain different numbers of variables. This characteristic makes each project repository dynamic. Each repository may also represent one or more simulation runs. By storing the variables in this manner, playbacks can be synchronized for all gauges in a given project. This playback feature allows for interactive analysis of the stored results. Note though, that since not all variables may be updated simultaneously, some non-key values may be undefined for some time stamps. During playbacks, undefined values for a given time stamp are interpreted to mean no update is required for the associated gauge.

4.2.2 Observing

The most exposed aspect of the *IMAGES* system is its visualization capabilities. Once an output variable's value is measured, it is displayed using one of many gauge types. The gauge types range from variations on the vertical or horizontal bar, such as a thermometer, to dials and timelines. Despite the variety of gauges, the underlying normalization mechanism is shared. Each gauge type is only responsible for mapping the normalized value to its interface based on the IGS

1. This repository is stored in 2NF (Date, 1990), with the primary key being the *time* variable.

configuration for the respective variable. This allows for extensions to the gauge types without sacrificing functionality, which is inherited by all *IMAGES* gauges.

Observation of the results can be customized using each gauge specific interface. Although all of the gauges share several common attributes, such as range of values, threshold levels, audible alarm indicators, statistics calculations, and unit of measure conversions, each gauge has specific characteristics that may be modified to suit the application or reflect the simulation output variable. For example, an output variable to represent speed with a unit of measure of kilometers per hour (km/h) may associate a dial gauge type customized to represent a speedometer, as illustrated in Figure 4-5.

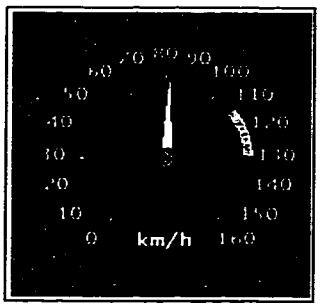


Figure 4-5 An example of a dial gauge type customized to represent a speedometer in units of kilometers per hour (km/h).

4.2.3 Monitoring

Many advanced features of the *IMAGES* gauge are its ability to monitor variables for thresholds, to indicate peak-water marks, and to calculate common statistical results. Depending on the appropriateness for a given type of gauge to give the user feedback, and based on the user's customization of the gauge, many attributes of a gauge are made available to facilitate the monitoring of a variable. The attributes consist of the following:

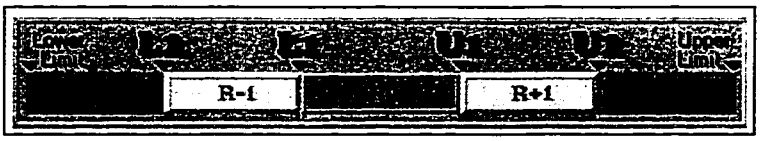


Figure 4-6 Gauge thresholds and range attributes.

- two high thresholds indicated in Figure 4-6 by the labels U1 and U2, and two low thresholds

indicated by the labels L1 and L2

- audible alarms at each threshold level for both increases to threshold regions R-1, R0, R+1 and R+2 and decreases to threshold regions R+1, R0, R-1 and R-2
- an adaptive range based on the peak-water mark or current measured value as illustrated in Figure 4-7
- unit of measure conversions
- statistics calculations with extendible formulae
- user override customization at run-time
- system-time or simulation/application-time based observation and monitoring
- visualization of both live, i.e. at client application run-time, and logged data
- gauge grouping, which allows a single variable to be observed using more than one gauge type, and
- remote observation and monitoring, which must be explicitly set by the user.

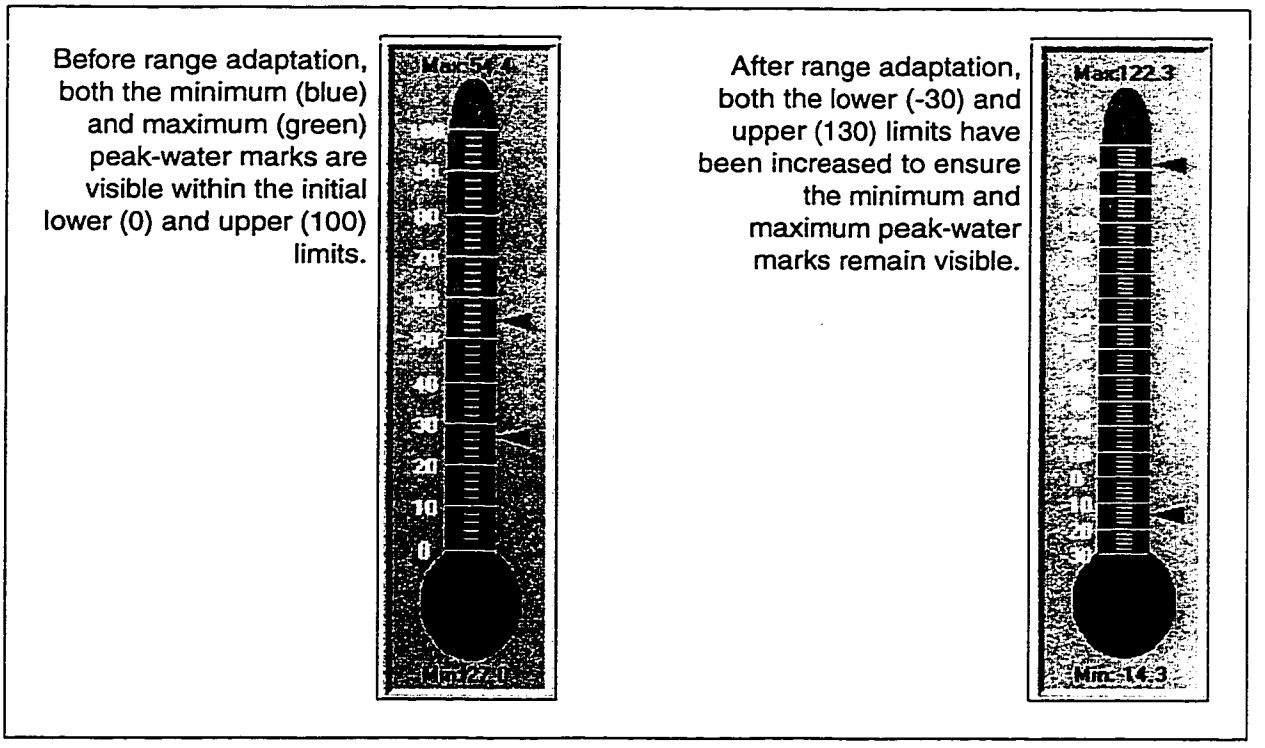


Figure 4-7 Dynamic ranges and peak-water marks.

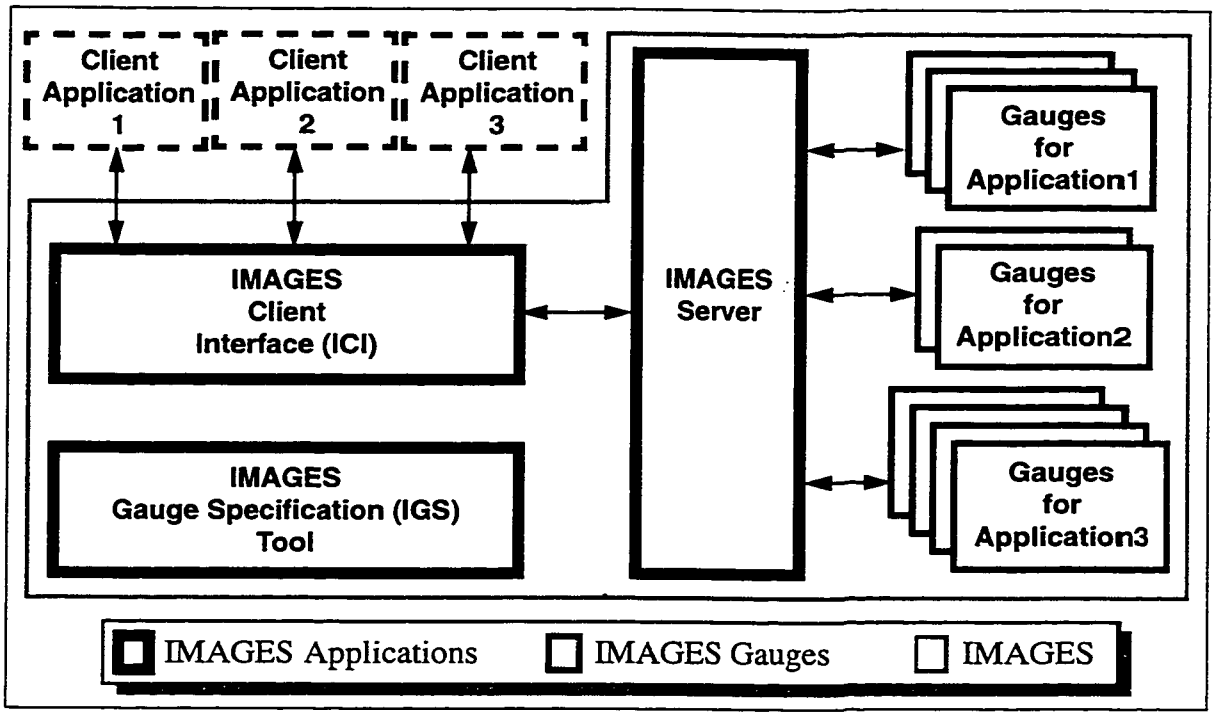


Figure 4-8 The *IMAGES* main components.

4.3 Building Blocks of *IMAGES*

IMAGES consists of three major components: the client-side application interface, the server application it communicates with, and the gauges (see Figure 4-8). The client interface is a Microsoft Windows[1] 32-bit dynamic link library (DLL). Its purpose is to manage conversations with the *IMAGES* server, and ultimately with the individual gauges, on behalf of the client application. The server is a Windows based executable application that communicates directly with both the client application and the gauges. It is also responsible for creating and initializing the gauges based on a gauge configuration file supplied by the client application. The gauges are Windows OLE custom extensions (OCX). Manipulation of a gauge can be initiated from three sources: the client application, the server, and directly via its user interface.

4.3.1 Client-side Application Interface

The client-side application programming interface (CS-API) provides access to the *IMAGES*

1. The generic term Windows will be used to implicitly mean Microsoft Windows 32-bit (i.e. Windows 95 and Windows NT), unless indicated otherwise.

gauges transparently via the server. By enabling client applications with an OLE automation mechanism to interface with the gauges, the client application does not require embedded gauges. This allows the gauges to execute in a separate memory space or even a separate host from the client application. This is precisely the same separation of concern and concurrency achieved by the Chiron-1 project (Taylor et al, 1994), with the restriction of application within the UNIX operating environment. The *IMAGES* application represents the modularity and reliability that would otherwise be lacking with alternative integration approaches.

The *IMAGES* CS-API is represented by the *IMAGES* Client Interface (ICI) in Figure 4-8. The CS-API is an OLE automation client dynamic link library (DLL). It achieves the low complexity and high flexibility of gauge integration into (existing) applications by enabling a client application to communicate with the *IMAGES* gauges via a set of two functions. One function dynamically creates and initializes a set of gauges based on a user prepared and specified gauge configuration file. A second function communicates an updated data value to the gauges. A one time invocation of the `ImagesGaugeInitialize()` function creates and initializes all gauges, and establishes communication channels between the client application and the *IMAGES* server and gauges. Gauge configuration information can consist of the variable name (i.e. airPressure) and its unit of measure (e.g. kPa), high/low ranges, high/low thresholds, gauge visibility, a repository to be used for serializing the data, audio level of threshold alarm(s), and sounds to play for each particular threshold alarm. The gauge configuration can be modified by simply using a different project file, also called an *IMAGES* gauge configuration (IGC) file. The function `ImagesGaugeUpdate()` sends a new data value to the gauge when the variable name is specified. Based on just the variable name, *IMAGES* can determine which gauge is measuring, observing, and monitoring it. A gauge is closed when the client application that requested the gauge terminates. This simple interface of two functions provides both less complex and a highly flexible means for interfacing with the *IMAGES* gauges.

Additional details on these two functions, and other value added functions, are presented in Section 5.2.

4.3.2 Server

The *IMAGES* server is both in-place OLE automation container and server enabled. It also comprises four main components: a supervisor agent, a network communication agent, a

component management table, and a gauge manager (see Figure 4-9). The interface management

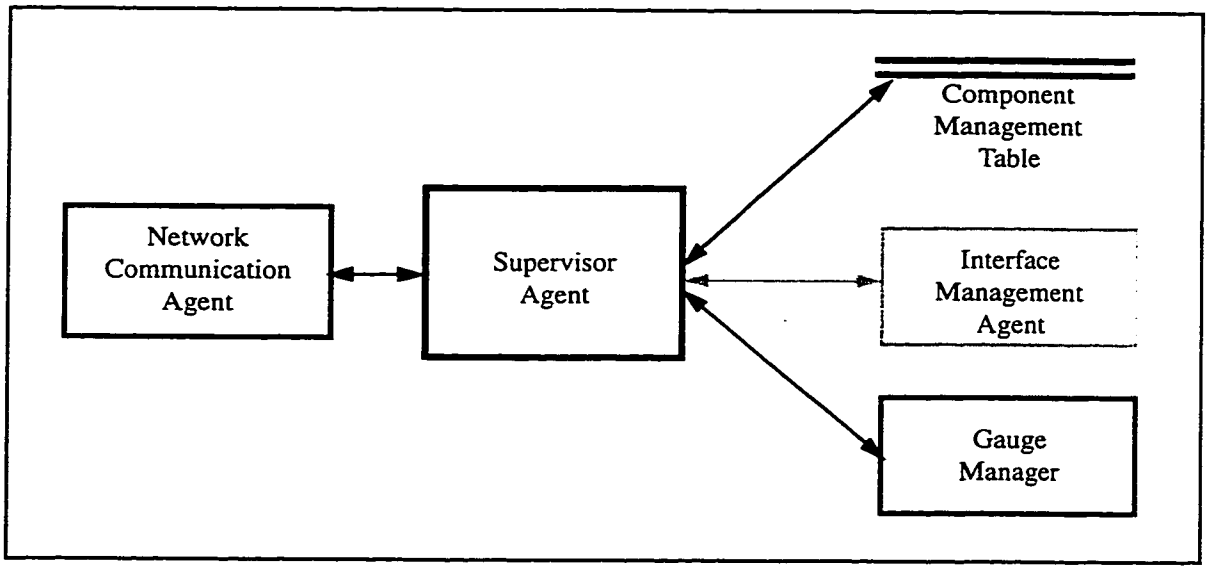


Figure 4-9 The *IMAGES* server architecture and its supervisor agent.

agent, presently in the design process, is intended to assist in managing the layout and orientation of the gauges for a project based on a particular user's preference. For example, a user may always prefer to orient voltmeters above ammeters and clocks, and hide the ammeters until a threshold is reached. In comparison to a gauge's interface management agent, the server's agent manages the relationship among a set of gauges, whereas a gauge's interface management agent manages its own interface. Only one *IMAGES* server instance exists for each computer.

Each server can communicate with other servers distributed across networks via the network communication agent. The network communication agent manages the communication channels between supervisor agents, and thus between the *IMAGES* servers. Supervisor agents communicate with each other transparently, and they have no knowledge of the communication medium being used. The network communication agent operates on the same concepts as routers, but without the use of address resolution protocols needed to build routing tables. Currently, the *IMAGES* inter-server communication table must be specified by a user or network administrator. This is accomplished by explicitly adding the IP addresses of the hosts, that are running the *IMAGES* server, as part of the server's preferences. The table can be augmented incrementally, and each address needs to be added only once, and from only one host. The network

communication agents share their new set of IP addresses among each other. This enables each network communication agent to learn new addresses from the other agents. This sharing of addresses is done whenever an *IMAGES* server comes on-line. As part of the initialization of the network communication agent, the local set of IP addresses is used to commence communication with the *IMAGES* servers operating on the hosts specified by these addresses. One agent goal, during this communication, is the exchange of these local IP addresses. By sharing these locally specified hosts that are running an *IMAGES* server, the exploration phase of determining other servers becomes semi-automated.

Fully automated exploration is technically feasible, however, in a worst case scenario, the vast expanse of the Internet would present a new set of challenges that are not a focus of this thesis. Nevertheless, there are three important benefits of semi-automated exploration for *IMAGES* servers. The first is the assurance that only those hosts specified will ever share their addresses, providing user managed privacy. The second is performance, that is, the time it takes to obtain all relevant *IMAGES* servers. The third is reduced manual configuration of *IMAGES* server connections.

4.3.2.1 Supervisor Agent

The supervisor agent is responsible for creating, instantiating, registering, and deregistering gauges. Creation of a gauge occurs based on the contents of a client application supplied *IMAGES* Gauge Configuration (IGC) file. Once created, a gauge must be initialized (by the supervisor agent) with the configuration specified in the IGC file. When a gauge completes its initialization process, it registers with the supervisor agent. Beyond these essential responsibilities, the supervisor agent's role is facilitator among the diverse software agents distributed among the three main *IMAGES* components (see Figure 4-10).

Several cooperative components are involved in achieving the intelligent, adaptive, and application-independent gauges. At the highest level, the components are grouped into those that are visible to the end-user and those that are functioning behind the scenes (or hidden from view). The three visible components are the IGS tool, the server, and the gauges. All remaining components are not visible[1] and they comprise the software agents and the gauge manager. Together, they cooperate to enhance programs with advanced gauges.

The most distinctive attribute of *IMAGES* is its cooperative multi-agent environment.

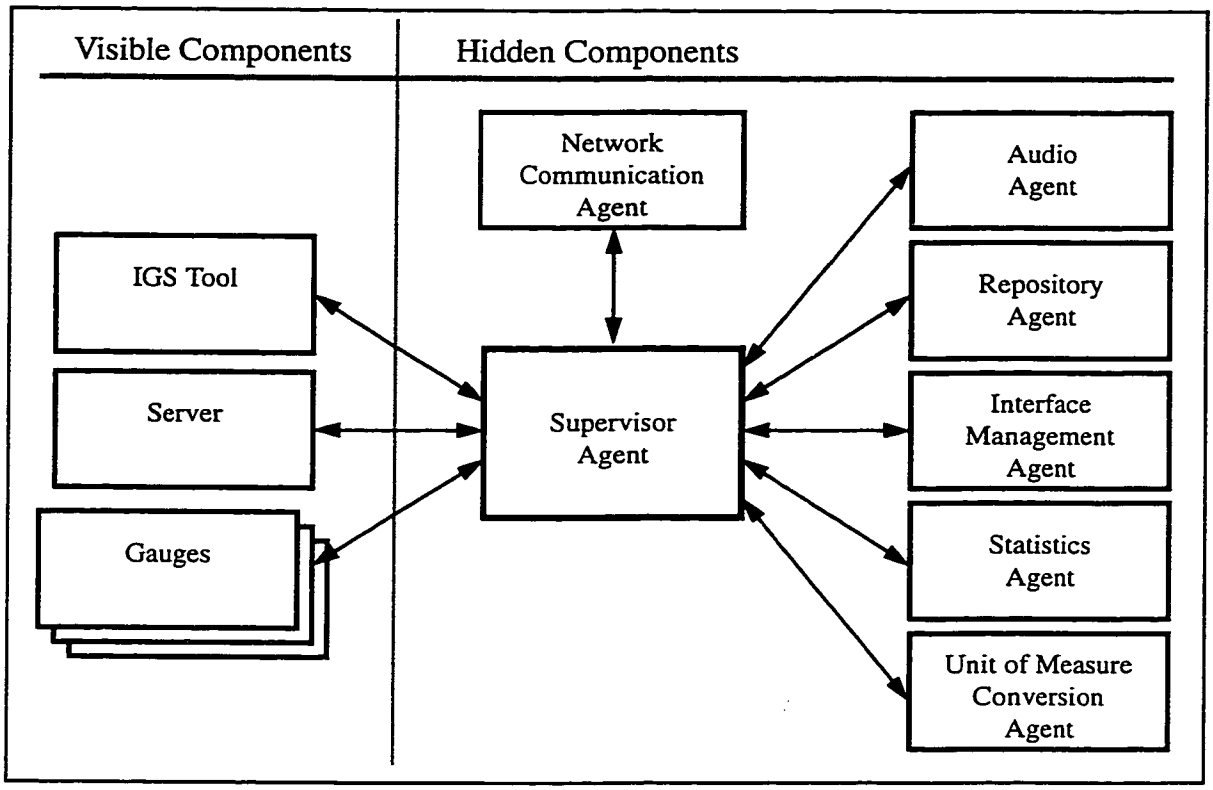


Figure 4-10 The *IMAGES* components and their communication, object and agent relationships to the supervisor agent.

1. Under normal operation all agents are hidden, however, for the purposes of diagnostic analysis of the agents at run-time, the *supervisor agent* manifests a facility (similar to a wire tap on a data communications link), to allow a software designer or simulationist, to visually monitor all on-going communication between all of the components.

Component Name	Protocol	State			Transition	Incarnation		Last Message
		previous	current	status		max	actual	
Server	IAP-A	A3	A4	good	15	1	1	LoadIgcFile
AudioAgent	IAP-B	B2	B3	bad	6	1	1	PlaySound
DatabaseLogAgent	IAP-C	-	-		-	1	0	-
InterfaceMgmtAgent	IAP-D	-	-		-	1	0	-
StatisticsAgent	IAP-E	E4	E8	good	22	1	1	CalcStat8
UnitConversionAgent	IAP-F	-	-		-	1	0	-
IGS Tool	IAP-G	-	-		-	32	0	-
Gauge	IAP-H	H3	H2	good	9	*	1	
Gauge	IAP-H	-	H0	bad	1	*	2	Register

Table 4-1 Example of the Component Management Table

The purpose of the component management table, demonstrated in Table 4-1, is to maintain design and run-time characteristics of all the components that comprise the *IMAGES* multi-agent environment. This table is maintained by the supervisor agent. The information contained in this table (i.e. each row entry) includes the following:

- the component name (e.g. Server, AudioAgent, Gauge, etc.). This is an *IMAGES* internal name, and has no meaning beyond the purpose of identifying a type of component;
- the protocol that each component supports for communicating with other *IMAGES* components. For example, the UnitConversionAgent uses the *IMAGES* Agent Protocol F (IAP-F) to communicate with other components (i.e. its interface language). Any component that needs to communicate with the UnitConversionAgent must do so using the IAP-F protocol;
- state information for the component, both previous and current states, and a qualitative status of the protocol state-machine (i.e. good or bad) for diagnostic purposes. Most components have a distinct state machine for their particular application. Each distinct state machine has states that are internally labeled, for example, as A1, and A2 for one component, and B1, B2, and B3 for another component. These labels are used to report state machine information to the supervisor agent. To maintain the availability status of a component, the supervisor agent

needs to ensure that a component can handle requests from other components, including itself. During the lifetime that a component is running, it may change states several times. Every component reports its current state information to the supervisor agent each time its current state changes. The current state information consists of three pieces of information, the current state, the transition traversed to reach the current state, and the status of the component. This timely information ensures that if a component reaches a bad state, the supervisor agent can cease or limit all further communication with that component. The primary purpose of the state information is for development diagnostics;

- the transition traversed from the previous to the current state. Each transition in a state machine is uniquely identified by an index. A component uses this index to report the transition traversed, in its state machine change, to the supervisor agent, as part of the state machine information;
- the incarnation of the component, both the maximum number of instances allowed to exist within the *IMAGES* environment, along with the instance number. Not all components within the *IMAGES* environment should have more than a single instance running. This information enables the supervisor agent to control the instances of a particular component, and maintain a count of components, such as the gauges, that may have many instances. The largest instance number could be interpreted as the quantity of that particular component; for example, if more than one gauge exists, the largest actual incarnation number among the gauge components is the total number of gauges that exist. This information is helpful for proper termination of the *IMAGES* environment; and
- the last message handled by the component. This information is primarily for development diagnostics. However, the supervisor agent uses this message and the resulting status of the component to decide if the component's protocol can handle the message. In the case that the component reaches a bad state after receiving a particular message, the supervisor agent can block these same messages during future communication with that component.

Further elaboration of this table is beyond the scope of this thesis.

The relative importance of the supervisor agent and its relationship with all the *IMAGES* components raises the two inextricable issues of performance and reliability. For reliability, the component management table is in the system registry not in memory. It is therefore crucial that

the system registry be backed up before using any *IMAGES* component. The need for reliability currently outweighs concerns for performance. However, for scalability, the responsive performance of the gauges to update requests becomes a serious issue.

4.3.2.2 Network Communication Agent

The network communication (NC) agent is responsible for managing communication between its supervisor agent and remote NC agents. The distributed nature of today's communication facilities provides the means to communicate across multiprotocol networks and over diverse global geographies. With a prime goal for *IMAGES* to enable distributed measuring, observing and monitoring, the ability to support the numerous network access and communication protocols is essential (see Figure 4-11). What is desirable when supporting the diverse network protocols, is

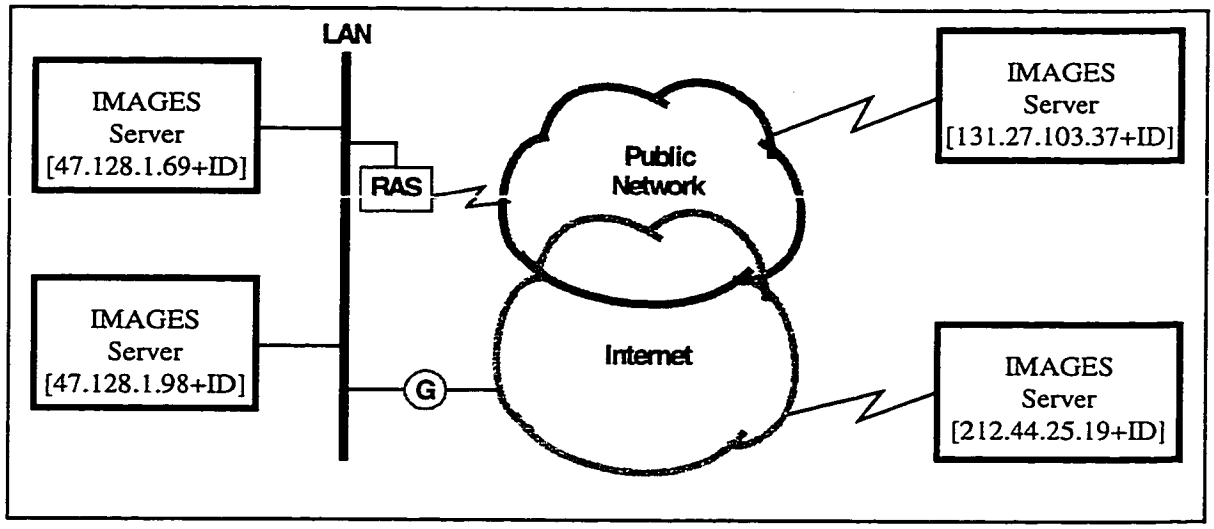


Figure 4-11 The *IMAGES* distributed supervisor agents and their potential connectivity across diverse private and public communication facilities for remote observation and

some knowledge of what facilities are available and some capability to choose the best option, considering performance and cost, at any given time.

To use the remote capabilities, for any given project, the user must explicitly specify the endpoint address of the remote computer running the *IMAGES* server. This endpoint address may be an IP (or Internet) address, a phone number plus an IP address, a computer name, or a computer name plus a domain name. Depending on the address option specified, the agent could proceed with a

connection, suggest a better alternative, or choose not to establish a connection due to historical problems. The NC agent may also suggest alternative communication methods if it has determined that the chosen option suffers from network congestion, lacks reliability, or is not cost effective.

Once a connection is established between two servers, the supervisor agents can communicate with each other as if they were on the same machine as the gauges they are managing. This maintains a single interface between the supervisor agent and the gauge manager, whether they are together or far apart. The NC agent is expected to maintain the connection and manage the transport of the protocol between the supervisor agent and the gauge manager.

4.3.2.3 Gauge Manager

The gauge manager is responsible for creating, instantiating, registering, and deregistering gauges. Creation of a gauge occurs based on the contents of a client application supplied *IMAGES* Gauge Configuration (IGC) file. Once created, a gauge must be initialized with the configuration specified in the IGC file. After a gauge completes its initialization process, it must register with the server. Registration involves adding an entry into a gauge management table, along with unique identifying attributes for the gauge, which is exchanged between the server and the gauges. Only upon successful registration is the client application permitted to commence gauge updates or further configuration. After communication is completed, or the client application terminates, the gauge manager sends the gauges a termination request. If a gauge is closed, via its user interface, it initiates a deregistration request to the server. When a gauge is terminated, its entry in the gauge management table is released and management resources are reclaimed.

For projects with no IGC file, a non-empty VAR file must exist to use *IMAGES*. Under this circumstance, when no gauge types are yet associated with the variables, it is *IMAGES*'s responsibility to automate this task. One service that can be used by the gauge manager is a gauge association agent responsible for deciding which type of gauge to associate with individual variables. The basis of these decisions is a knowledgebase of unit-of-measure and gauge types chosen during other projects. As more and more projects are used with *IMAGES*, by either a single user or multiple users that may be distributed across networks, the knowledgebase can be built up. This process of sharing knowledge of gauge associations is handled by the gauge association agent, using the supervisor agent as a facilitator.

Currently, there is no implementation of a concept learning system for the gauge manager's gauge association agent, although it is noted that the knowledgebase could be augmented by learning from the user. One possibility for accomplishing this is by experiencing the type of gauge to unit associations made by the user. Storing this learned information in the knowledgebase would provide the current gauge type selection and suggestion process with a more trusted and certainly more desirable intelligence feature (Maes, 1994).

4.3.3 Gauges

Several commercially available visualization tools address a variety of applications; however, those with sophisticated visualization capabilities are complex to use. *IMAGES* is not intended to develop a better visualization tool, but it is intended to improve the utilization and integration of such tools in software. As such, the gauge interface employed by *IMAGES* is based on the Global Majic Software (GMS) toolkit. This toolkit offers flexibility in visualizing data using many gauge types, including support for custom user created types. It also supports gauge groups, such that a single value can be plotted using more than one gauge, with the restriction that all gauges be of the same type. The most attractive characteristic of this toolkit is that all of this functionality is accomplished at a very low cost.

IMAGES provides the extended capability of visualizing the same variable using more than one gauge type. More important, the *IMAGES* virtual gauge adds the software agent enhancements, including: unit-of-measure conversions, statistics calculations, repository logging, replay of the data, and monitoring for thresholds. The *IMAGES* gauge can be viewed as a wrapper around the GMS toolkit, handling and managing all aspects of the GMS instruments. *IMAGES* buffers the user and client application from the instrumentation details.

Currently, *IMAGES* supports a selection of seven gauge types. They include: horizontal and vertical bars, circular and arc (i.e. semi-circular) dials, an LED display, an odometer, a boolean indicator, a thermometer, and a time-based graph. A sample of these gauges is illustrated in Appendix A.

4.3.4 Gauge Configuration

An *IMAGES* gauge has 59 attributes, 58 of which can be configured. The attributes of an *IMAGES* gauge are listed and described in Table 4-2. Configuration of a gauge is simplified by using the *IMAGES* Gauge Specification (IGS) Tool.

There are six groups of gauge attributes: 1) interface, 2) range and thresholds, 3) warning indicators, 4) logfile, 5) knowledgebase, and 6) statistics. The configuration details for each of these groups are described here. Some attributes are modifiers for others. In these cases, when the modifier attribute is selected, the resulting value of the modified attribute is called the effective value (i.e. the value calculated and used by the gauge).

The interface group represents most of the attributes that affect the visual presentation of a gauge.

The range and thresholds group specifies the limits and threshold regions that are to be monitored for the variable. An upper and a lower dynamic limit exists so that if a value goes beyond the limit, the gauge can still visualize the result by expanding the range as necessary. Five threshold regions (R-2, R-2, R0, R+1, and R+2) are created by four threshold levels (L2, L1, U1, and U2).

The warning indicators group enables or disables both visual and audible warnings for each threshold region. A threshold region may be reached either by an increase or a decrease in the value. A total of eight possible threshold breaches may result: as an increase to R+2, R+1, R0, or R-1, or as a decrease to R+1, R0, R-1 or R-2.

The logfile group enables or disables storage of the variable's values to a file.

The knowledgebase group currently only manages conversion between units of measure. Conversion between units of measure can be enabled or disabled.

The statistics group enables or disables the statistics features of the *IMAGES* gauge.

Group	Attribute	Description
n/a	variable name	The variable to be monitored by a gauge.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Interface	variable type	The data type for the variable. The options are: - Boolean - Double (15 digit precision) - Float (7 digit precision) - Long double (19 digit precision) - Signed Int16 - Signed Int32 (*default*) - Signed Int64 - Unsigned Int16 - Unsigned Int32
	gauge type	The type of gauge to be associated with the variable. It indicates which type of gauge is to be used for visualizing the values of the variable. The options are: - Bar Horizontal (*default*) - Bar Vertical - Dial Arc - Dial Circular - Odometer - LED display - Toggle Indicator - Thermometer - Plotter If no gauge type is manually associated with a variable, the default gauge type is automatically associated with the variable. In addition, if the option to *learn new gauge associations* is selected, under the project preferences, default associations are not used to build the gauge association knowledgebase.
	visible	This option indicates whether or not the gauge is to be visible. If selected, the gauge is always visible. If not selected, the gauge only becomes visible if a warning is raised due to a threshold condition. A gauge is always visible by default.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Interface	unit of measure	A unit of measure indicates the unit to be used for the quantity being visualized. The user may choose from among a predefined set of units of measure, or expand this set of units of measure, to include any units they deem appropriate for their application. For example, since *millennium* is not included in the predefined set of units of measure. it may be added and associated immediately with a gauge. Any unit of measure can become part of the unit of measure-to-gauge type association knowledgebase.
	color	The gauge color specifies the color of the indicator. For example, if blue is selected as the gauge color, then a horizontal bar gauge displays the value of the variable using a blue indicator. That is, if the range is 0 to 100, and the value is 37, then 37% of the horizontal bar would be blue.
Range and Thresholds	initial value	The initial value to be displayed by the gauge.
	lower limit	The lowest expected or acceptable value for the variable. This value may be changed dynamically, by the gauge, if the *lower limit dynamic* attribute is selected. If the dynamic option is not selected, this value enforces a lower limit on the visualized value for the variable. This value cannot be less than the minimum value supported by the selected *variable type*, and no greater than the effective upper limit. Refer to Figure 4-6.
	upper limit	The highest expected or acceptable value for the variable. This value may be changed dynamically, by the gauge, if the *upper limit dynamic* attribute is selected. If the dynamic option is not selected, this value enforces an upper limit on the visualized value for the variable. This value cannot be greater than the maximum value supported by the selected *variable type*, and not less than the effective lower limit. Refer to Figure 4-6.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	lower limit dynamic	This option, if selected, allows the gauge to dictate the lowest value to be visualized. The gauge determines the effective lower limit dynamically based on the instantaneous minimum value of the variable, plus a 5% factor of the new range. For example, a range of -1200 to 100, with a new value of -1210, would cause the gauge to automatically adjust the range to -1276 to 100. If this option is not selected, the lowest value to be visualized is dictated by the *lower limit* attribute.
	lower limit relative	This option enables the user to indicate whether the *lower limit* attribute is a value relative to the *initial value* attribute. If selected, the *lower limit* is relative to the *initial value*, otherwise the *lower limit* is an absolute value. For example, if this option is selected, with the *initial value* set to 75, a *lower limit* of -100 would result in an effective lower limit of -25.
	upper limit dynamic	This option, if selected, allows the gauge to dictate the highest value to be visualized. The gauge determines the effective upper limit dynamically based on the instantaneous maximum value of the variable, plus a 5% factor of the new range. For example, a range of -1200 to 100, with a new value of 195, would cause the gauge to automatically adjust the range to -1200 to 265. If this option is not selected, the highest value to be visualized is dictated by the *upper limit* attribute.
	upper limit relative	This option enables the user to indicate whether the *upper limit* attribute is a value relative to the *initial value* attribute. If selected, the *upper limit* is relative to the *initial value*, otherwise the *upper limit* is an absolute value. For example, if this option is selected, with the *initial value* set to 75, an *upper limit* of 200 would result in an effective lower limit of 275.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold L2	This is lower threshold level-2. It is the second lowest threshold that can be set. This optional threshold, if specified, can be used to set the second lowest range of values that the user would be notified of, via the gauge warning indicators. The value specified for this attribute delineates the point at which the R-2 threshold region is divided from the R-1 threshold region. The effective value of this attribute can never be less than the *lower limit*, and no greater than *threshold L1*. If the *threshold L2 percentage* option is selected, this value must be in the range 0 to 100. This threshold is never modified by the gauge, even if the *lower limit dynamic* attribute is set. If this value is not specified, it is automatically set to be equal to the effective lower limit. Refer to Figure 4-6.
	threshold L2 relative	This option enables the user to indicate whether the *threshold L2* attribute is a value relative to the *lower limit* attribute. If selected, *threshold L2* is relative to the *lower limit*, otherwise *threshold L2* is an absolute value. For example, if this option is selected, with the *lower limit* set to -25, and a *threshold L2* of 20 would result in an effective threshold L2 of -5.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold L2 percentage	This option enables the user to indicate whether or not the *threshold L2* attribute is a percentage. If this option is selected, and the *threshold L2 relative* option is not selected, then *threshold L2* is calculated as a percentage of the total range between the effective upper and lower limits. For example, if the effective upper limit is 100, the effective lower limit is -150, and *threshold L2* is 15%, then the effective threshold L2 is derived as: -150 + ((100 - (-150)) x 15%) = -112. If both this option, and the *threshold L2 relative* option are selected, then the effective threshold L2 is calculated as a percentage of the range between the effective lower limit and the *initial value*. For example, if the effective lower limit is set to -150, the *initial value* is 22, and *threshold L2* is 15%, then the effective threshold L2 is derived as: -150 + ((22 - (-150)) x 15%) = -124.
	threshold L1	This is lower threshold level-1. It is the first lowest threshold that can be set. This optional threshold, if specified, can be used to set the first lowest range of values that the user would be notified of, via the gauge warning indicators. The value specified for this attribute delineates the point at which the R-1 threshold region is divided from the R0 threshold region. The effective value of this attribute can never be less than the effective threshold L2 value, and no greater than the effective threshold U1 value. If the *threshold L1 percentage* option is selected, this value must be in the range 0 to 100. This threshold is never modified by the gauge. If this value is not specified, it is automatically set to be equal to the effective threshold L2. Refer to Figure 4-6.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold L1 relative	This option enables the user to indicate whether the *threshold L1* attribute is a value relative to the *threshold L2* attribute. If selected, *threshold L1* is relative to *threshold L2*, otherwise *threshold L1* is an absolute value. For example, if this option is selected, with *threshold L1* set to 60, and an effective absolute threshold L2 of -50, results in an effective absolute threshold L1 of 10.
	threshold L1 percentage	This option enables the user to indicate whether or not the *threshold L1* attribute is a percentage. If this option is selected, and the *threshold L1 relative* option is not selected, then *threshold L1* is calculated as a percentage of the total range between the effective upper and lower limits. For example, if the effective upper limit is 100, the effective lower limit is -150, and *threshold L1* is 25%, then the effective threshold L1 is derived as: -150 + ((100 - (-150)) x 25%) = -87. If both this option, and the *threshold L1 relative* option are selected, then effective threshold L1 is calculated as a percentage of the range between the effective threshold L2 and the *initial value*. For example, if the effective threshold L2 is -120, the *initial value* is 22, and *threshold L1* is 25%, then the effective absolute threshold L1 is derived as: -120 + ((22 - (-120)) x 25%) = -84.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold U1	This is upper threshold level-1. It is the first highest threshold that can be set. This optional threshold, if specified, can be used to set the first highest range of values that the user would be notified of, via the gauge warning indicators. The value specified for this attribute delineates the point at which the R0 threshold region is divided from the R+1 threshold region. The effective value of this attribute can never be greater than the effective threshold U2 value, and no less than the effective threshold L1 value. If the *threshold U1 percentage* option is selected, this value must be in the range 0 to 100. This threshold is never modified by the gauge. If this value is not specified, it is automatically set to be equal to the effective threshold U2. Refer to Figure 4-6.
	threshold U1 relative	This option enables the user to indicate whether the *threshold U1* attribute is a value relative to the *threshold U2* attribute. If selected, *threshold U1* is relative to *threshold U2*, otherwise *threshold U1* is an absolute value. For example, if this option is selected, with *threshold U1* set to 90, and an effective absolute threshold U2 of 250, results in an effective absolute threshold U1 of 160.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold U1 percentage	This option enables the user to indicate whether or not the *threshold U1* attribute is a percentage. If this option is selected, and the *threshold U1 relative* option is not selected, then *threshold U1* is calculated as a percentage of the total range between the effective upper and lower limits. For example, if the effective upper limit is 100, the effective lower limit is -150, and *threshold U1* is 55%, then the effective threshold U1 is derived as: -150 + ((100-(-150)) x 55%) = -12. If both this option, and the *threshold U1 relative* option are selected, then effective threshold U1 is calculated as a percentage of the range between the effective threshold U2 and the *initial value*. For example, if the effective threshold U2 is 220, the *initial value* is 22, and the *threshold U1* is 55%, then the effective absolute threshold U1 is derived as: 22 + ((220 - 22) x 55%) = 130.
	threshold U2	This is upper threshold level-2. It is the second highest threshold that can be set. This optional threshold, if specified, can be used to set the second highest range of values that the user would be notified of, via the gauge warning indicators. The value specified for this attribute delineates the point at which the R+1 threshold region is divided from the R+2 threshold region. The effective value of this attribute can never be greater than the *upper limit*, and no less than *threshold U1*. If the *threshold U2 percentage* option is selected, this value must be in the range 0 to 100. This threshold is never modified by the gauge, even if the *upper limit dynamic* attribute is set. If this value is not specified, it is automatically set to be equal to the effective upper limit. Refer to Figure 4-6.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Range and Thresholds	threshold U2 relative	This option enables the user to indicate whether the *threshold U2* attribute is a value relative to the *upper limit* attribute. If selected, *threshold U2* is relative to the *upper limit*, otherwise *threshold U2* is an absolute value. For example, if this option is selected, with the *upper limit* set to 225, and a *threshold U2* of 120 would result in an effective threshold U2 of 105.
	threshold U2 percentage	This option enables the user to indicate whether or not the *threshold U2* attribute is a percentage. If this option is selected, and the *threshold U2 relative* option is not selected, then *threshold U2* is calculated as a percentage of the total range between the effective upper and lower limits. For example, if the effective upper limit is 100, the effective lower limit is -150, and *threshold U2* is 95%, then the effective threshold U2 is derived as: -150 + ((100-(-150)) x 95%) = 87. If both this option, and the *threshold U2 relative* option are selected, then the effective threshold U2 is calculated as a percentage of the range between the effective upper limit and the *initial value*. For example, if the effective upper limit is set to 220, the *initial value* is 22, and *threshold U2* is 95%, then the effective threshold U2 is derived as: 22 + ((220 - 22) x 95%) = 210.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Warning Indicators	visual	This option enables or disables visual warning indicators for all threshold regions (i.e. R-2, R-1, R0, R+1, and R+2). A visual warning indicator is raised if the current value of the variable crosses a threshold into one of the five threshold regions. The visual warning depends on the type of gauge selected. Each gauge has its own visual warning indicators that provide visual cues for the warning levels. The warning for each threshold region can be turned either on or off, and a color can be specified for the visual cue.
	audible	This option enables or disables audible warning indicators for all threshold regions (i.e. R-2, R-1, R0, R+1, and R+2). An audible warning indicator is played if the current value of the variable crosses a threshold into one of the five threshold regions. The warning for each threshold region can be turned either on or off, and a sound can be specified for the audible cue.
	audio player	This is the audio player that is to be used for providing the audible warnings. A default player is provided, however, any audio player can be used.
	Increase to R+2 warn	This option enables or disables warnings for value increases into the R+2 threshold region.
	Increase to R+2 color	This attribute specifies the color of the visual warning cue for value increases into the R+2 threshold region.
	Increase to R+2 audiofile	This attribute specifies the filename of the audible warning cue for value increases into the R+2 threshold region.
	Increase to R+1 warn	This option enables or disables warnings for value increases into the R+1 threshold region.
	Increase to R+1 color	This attribute specifies the color of the visual warning cue for value increases into the R+1 threshold region.
	Increase to R+1 audiofile	This attribute specifies the filename of the audible warning cue for value increases into the R+1 threshold region.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Warning Indicators	Increase to R0 warn	This option enables or disables warnings for value increases into the R0 threshold region.
	Increase to R0 color	This attribute specifies the color of the visual warning cue for value increases into the R0 threshold region.
	Increase to R0 audiofile	This attribute specifies the filename of the audible warning cue for value increases into the R0 threshold region.
	Increase to R-1 warn	This option enables or disables warnings for value increases into the R-1 threshold region.
	Increase to R-1 color	This attribute specifies the color of the visual warning cue for value increases into the R-1 threshold region.
	Increase to R-1 audiofile	This attribute specifies the filename of the audible warning cue for value increases into the R-1 threshold region.
	Decrease to R+1 warn	This option enables or disables warnings for value decreases into the R+1 threshold region.
	Decrease to R+1 color	This attribute specifies the color of the visual warning cue for value decreases into the R+1 threshold region.
	Decrease to R+1 audiofile	This attribute specifies the filename of the audible warning cue for value decreases into the R+1 threshold region.
	Decrease to R0 warn	This option enables or disables warnings for value decreases into the R0 threshold region.
	Decrease to R0 color	This attribute specifies the color of the visual warning cue for value decreases into the R0 threshold region.
	Decrease to R0 audiofile	This attribute specifies the filename of the audible warning cue for value decreases into the R0 threshold region.
	Decrease to R-1 warn	This option enables or disables warnings for value decreases into the R-1 threshold region.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Warning Indicators	Decrease to R-1 color	This attribute specifies the color of the visual warning cue for value decreases into the R-1 threshold region.
	Decrease to R-1 audiofile	This attribute specifies the filename of the audible warning cue for value decreases into the R-1 threshold region.
	Decrease to R-2 warn	This option enables or disables warnings for value decreases into the R-2 threshold region.
	Decrease to R-2 color	This attribute specifies the color of the visual warning cue for value decreases into the R-2 threshold region.
	Decrease to R-2 audiofile	This attribute specifies the filename of the audible warning cue for value decreases into the R-2 threshold region.
Logfile	log to file	This option indicates whether or not to serialize the measured variable values. ***If disk space is limited, it is recommended that this option not be selected.***
	action on existing file	This option indicates the action to be taken if the *log file* attribute specifies a file that already exists. There are three options: - *Append*: adds the new values to the end of the existing file. - *Rename*: prompts the user to specify a new name for the existing file. - *Replace*: deletes the existing file, and creates a new file using the specified filename.
	log file	This attribute specifies the filename that is to be used for serializing the values of the variable.

Table 4-2 *IMAGES* Gauge Attributes

Group	Attribute	Description
Knowledge-base	convert unit of measure	This option enables or disables unit of measure conversion for the variable. If selected, the unit of measure used, to generate the values of the variable, are converted to a new specified unit of measure.
	from unit of measure	The unit of measure that the values of the variable will be converted from. The unit of measure for the variable must already be defined by the *unit of measure* attribute (from the Interface group). However, this attribute overrides the *unit of measure* attribute for the purpose of the conversion process. For example, if the *unit of measure* is kilometer, and this attribute is meter, then the conversion process will use meter as the conversion factor instead of kilometer.
	to unit of measure	The unit of measure that the values of the variable will be converted to. The list of available units of measure to convert to must already exist in the set of units of measure. See the *unit of measure* attribute under the Interface group to add to the units of measure list. The *from unit of measure*, and the *to unit of measure* attributes cannot be the same.
Statistics	calculate	This option enables or disables the calculation of statistics. If selected, this option provides access to the *IMAGES* statistics features. This option can only be selected if the *log to file* option is also selected. If not selected, the statistics features are not available. ***If system memory is limited, it is recommended that this option not be selected.***

Table 4-2 *IMAGES* Gauge Attributes

5. Integration of *IMAGES* With An Application

5.1 Requirements To Use *IMAGES*

The requirements and procedures discussed in this chapter are valid for software systems at any stage of their life cycle (Pressman, 1987), including legacy software. A clearly defined three stage integration process permits advanced planning for integrating the *IMAGES* system into a software project. Whether a project is in as-early-as the initial requirements specification phase of its life cycle or as-late-as the maintenance phase, the simplicity of the *IMAGES* integration requirements facilitates this life cycle independent integration possibility. The integration requirements of *IMAGES* consists of three elements. The first element is the independence of the implementation language. This is to allow any development environment capable of invoking dynamic link library calls, to integrate with the *IMAGES* interface via its minimal API[1]. The second element is the API itself, which consists entirely of three function calls, all of which are simple yet, highly functional. The third element is the clearly defined three stage process for integration, of which one stage may be omitted, and yet another stage that is handled automatically by the *IMAGES* system at run-time. These three elements satisfy the integration requirements for simplifying the

1. All references to the abbreviation API (application programming interface), refers to the *IMAGES* API, unless explicitly stated otherwise.

integration process with the *IMAGES* system. The simple integration process of *IMAGES* allows for it to be integrated during any stage of a software life-cycle.

The following subsections expand on the three elements that define the integration. In the first subsection, the *IMAGES* interface, consisting of the API and the means for enabling a simulation or application program to communicate with the external gauges is covered. In the sequent subsection the three stages of the integration process are detailed. A third subsection illustrates the integration process with the M-Gest simulation environment.

5.2 *IMAGES* Interface

The *IMAGES* interface is defined by its API which consists entirely of three functions. By design, it is minimal in complexity while providing maximum modularity, flexibility, scalability and functionality. The modularity is achieved by separating the interface from the client application that is harnessing the gauges. This allows for maintenance activities to be performed on the *IMAGES* side of the architecture, while not imposing development requirements on the client application. Its flexibility is represented by its ability to be turned *on* or *off* either at run-time or compile-time, and by its ability to be integrated during any stage of a software life cycle. Scalability is supported from single client applications to multiple applications on a single host, and from very large systems requiring hundreds of gauges to distributed client applications that are executing in a heterogeneous network. The only restriction to achieving this level of scalability is that the hosts be fully CORBA/IIOP[2] (OMG, 1996) compliant. As has already been presented in Chapter 4, there are several functions offered by the *IMAGES* gauges and their supporting cast of specification tool, server, and embedded software agents.

The API is language independent such that a client application sourced in Visual BASIC, Pascal, Delphi, C/C++, or any language or development environment that is capable of making calls into

2. Common Object Request Broker Architecture (CORBA) is a specification developed by a consortium of major software vendors whose primary objective is to develop specifications that enable object-oriented technology to be supported in a distributed computing environment. This group is called the Object Management Group (OMG.) All applications that use ObjectBroker on Windows 95, Windows NT, OSF/1, OpenVMS, ULTRIX, HP-UX and SunOS platforms are CORBA compliant. A complete list of compliant operating systems and applications are available at the OMG URL http://www.omg.org/.

a Microsoft Windows[3] dynamic link library can be integrated with *IMAGES*. In the following three tables, the API is detailed:

Function	`ImagesInterfaceOnOff( <flagOnOff> )`
Parameters	***flagOnOff***
C/C++	`bool flagOnOff`
Pascal/Delphi	`flagOnOff : Boolean`
Visual Basic	`byVal flagOnOff As Boolean`
Values	IMAGES_ON - to turn the IMAGES interface on IMAGES_OFF - to turn the IMAGES interface off
Frequency	Single - optional
Location	To be inserted at any point in the client application.
Description	This function/procedure may be called by the client application at run-time to either turn the *IMAGES* interface *on* (IMAGES_ON) or *off* (IMAGES_OFF). The equivalent can be performed at compile-time to permanently turn the interface *on* (`#define IMAGES`) or *off* (`#undef IMAGES`), by setting the appropriate preprocessor flag, or development environment conditional compiler directives. The benefit of turning it *off* at compile-time is to reduce the executable size, since no *IMAGES* API code is added to the client application. The disadvantage is the elimination of the flexibility to turn it *on* or *off* at run-time. Even if the compile-time option is chosen, to turn the interface *off*, no *IMAGES* API statements need to be removed from the client application, the *IMAGES* implementation will take care of both options. For the compile-time option, it is either *on* or *off* for **all** instances of an application. For the run-time option, it can either be *on* or *off* for **each** instance of an application.
Example	`#define IMAGES // Enable IMAGES Interface`

C/C++	`ImagesInterfaceOnOff( (myMenuItem.GetCheck() == 1)`
	`? IMAGES_ON`
	`: IMAGES_OFF );`

Table 5-1 *IMAGES* API - ImagesInterfaceOnOff

3. All references to Windows is implicitly the Microsoft Windows 95 or Windows NT operating system.

Example
Delphi or Borland Pascal
	{$DEFINE IMAGES} // Enable IMAGES Interface
	
	if (myMenuItem.GetCheck() = 1)
	then ImagesInterfaceOnOff(IMAGES_ON);
	else ImagesInterfaceOnOff(IMAGES_OFF);

Example
Visual Basic
	If (myMenuItem.GetCheck() = 1) Then
	  ImagesInterfaceOnOff(IMAGES_ON)
	Else
	  ImagesInterfaceOnOff(IMAGES_OFF)
	End If

Table 5-1 *IMAGES* API - ImagesInterfaceOnOff

Function	ImagesInitialize(<*igcFilename*>)
Parameters C/C++ Pascal/Delphi Visual Basic	***igcFilename*** const char* igcFilename igcFilename : Char[256] byVal igcFilename as String
Frequency	Single - mandatory
Location	To be inserted into the initialization sequence of the simulation program.
Description	This function performs all preparatory tasks necessary to access the *IMAGES* Server and the gauges. It initializes the local *IMAGES* client-side data structures and the communication facility (OLE) between the simulation program (client) and the *IMAGES* server. Once initialized, the gauges are requested to be created and initialized.

Example
C/C++
	BOOL myApp::InitInstance()
	{
	  AfxEnableControlContainer(); // For Visual C++ w/MFC
	
	  // Other application initialization - before or after is okay
	
	  ImagesInitialize(getIgcFilename());
	
	  // Dialog based applications should use DoModal() after
	  // ImagesInitialize() or within the dialog class
	
	  return FALSE;
	}

Table 5-2 *IMAGES* API - ImagesInitialize

Example
Delphi or
Borland Pascal
	`function initInstance() : Boolean;`
	`begin`
	`  AfxEnableControlContainer(); { For Pascal/Delphi for Windows }`
	
	`  { Other application initialization - before or after is okay }`
	
	`  ImagesInitialize( getIgcFilename() );`
	
	`  { Dialog based applications should use DoModal() after }`
	`  { ImagesInitialize() or within the dialog class }`
	
	`  initInstance := False;`
	`end.`

Example
Visual Basic
	`Function InitInstance() As Boolean`
	`  AfxEnableControlContainer()`
	
	`  ' Other application initialization - before or after is okay`
	
	`  igcFilename = getIgcFilename()`
	`  ImagesInitialize( igcFilename )`
	
	`  ' Dialog based applications should use DoModal after`
	`  ' ImagesInitialize or within the dialog class`
	
	`  InitInstance = False`
	`End Function`

Table 5-2 *IMAGES* API - ImagesInitialize

Function	`ImagesGaugeUpdate( <variableName> );`
Parameters	***variableName***
C/C++	a variable of any basic system type (e.g. `bool`, `int`, `uint32`, `float`, etc.)
Pascal/Delphi	a variable of any basic system type (e.g. `Boolean`, `Integer`, `Real`, etc.)
Visual Basic	a variable of any basic system type (e.g. `Integer`, `Double`, etc.)

Table 5-3 *IMAGES* API - ImagesGaugeUpdate

Values	A variable that evaluates to an atomic and basic system type value. For example: - `m_nVector[5]` : the 6th (C/C++) or 5th (Pascal/Delphi/Basic) element in the one-dimension array of integers - `m_dFlag` : a simple variable holding a double value - `(bool)(m_bFirstFlag | m_bSecondFlag)` : evaluates to a single boolean value - `((m_nMyIndex > 100) OR (m_nMyLimit < 150))` : evaluates to a single boolean value - `&myMemoryAddress` : (C/C++) evaluates to the address of the variable, which may be used to monitor memory allocation. All three examples above, although some are complex, are valid parameters since they all resolve to a single stored value. However, the following examples, although they may compile, may not work as expected: - `m_nVector` : an array (in C/C++ this is equivalent to `m_nVector[0]`, in Pascal/Delphi it evaluates to the address of the array, in Basic this is a compile-time error) - `m_f2dArray[9]` : 9th row of a 2-dimension array, not a single value in the row - `*(m_pMyMemory + nXyzOffset)` : a value at a calculated address in memory has no predetermined type. To make this work, the referenced value must be typecast to the expected basic system type. For example, `(float)*(m_pMyMemory + nXyzOffset)` would make it acceptable. There may be other examples resulting in potentially unexpected results when complex expressions are used as the parameter for this API function. As a result, the fundamental rule to adhere to consists of the following two criteria: 1) the expression must evaluate to an atomic value, and 2) the value must be of a basic system type.
Frequency	Multiple - optional
Location	To be inserted after each instance that an auxiliary, state or output variable is updated (i.e. following each statement that has one of these three types of variables as an *l*-value).
Description	This function can be considered to be similar to an *l*-value in an assignment statement, where the gauge is the *l*-value, and the parameter is the variable holding the value to update the gauge with. The *<variableName>* parameter must be unique within the scope of the gauge, since it is used to access a specific gauge.

Table 5-3 *IMAGES* API - ImagesGaugeUpdate

Example C/C++

```
void CMySimulator::start()
{
#if defined(DEBUG)
    int simIterations = 0;
#endif

    while (stopCondition() == false) {
#if defined(DEBUG)
        simIterations++;
        ImagesGaugeUpdate( simIterations );
#endif
        updateTimeInterval();
        updateStateVariables();
        updateImagesGauges();
    }
}

void CMySimulator::updateImagesGauges()
{
    int glbSyncTime = GetNetSyncSystemTime();
    ImagesGaugeUpdate( glbSyncTime );
    ImagesGaugeUpdate( mInt32_simTime );
    ImagesGaugeUpdate( mFloat_simStateVar1 );
    ImagesGaugeUpdate( mDouble_simStateVar2 );
}
```

Table 5-3 *IMAGES* API - ImagesGaugeUpdate

Example
Delphi or
Borland Pascal

```
{-------------------------------------------------------}
{ Note how each variabe that is to be connected with   }
{ a gauge appears in an ImagesGaugeUpdate() statement. }
{-------------------------------------------------------}
procedure CopyFileMultiple( var file; var nCopies : Integer )
    var fileCount, byteCount, totalByteCount : Integer;
begin
    ImagesGaugeUpdate( nCopies );

    totalByteCount := GetFileSize( file ) * nCopies;
    byteCount := 0;

    for fileCount := 1 to nCopies do
    begin
       ImagesGaugeUpdate( fileCount );
       byteCount := byteCount + CopyFile( file );
       ImagesGaugeUpdate( byteCount );
    end;
end.
```

Example
Visual Basic

```
Private Sub someCounter(startIndex As Integer, _
                        endIndex As Integer, _
                        anyStep As Integer)
Dim countI As Integer

For countI = startIndex To endIndex Step anyStep
    ImagesGaugeUpdate( countI )
Next countI

End Sub
```

Table 5-3 *IMAGES* API - ImagesGaugeUpdate

Most compilers require that a prototype definition of functions be provided, even if the function bodies are not available. As an example, for ANSI C/C++, a function `foo()` that is referenced within a file `myFile.cc`, must have at the very least a prototype for `foo()` to perform type validation. This is particularly important for the more strict type matching C++ language. To satisfy these language requirements, an appropriate header, library, and unit or inclusion file is provided for facilitating the integration development stage.

The following tables (Table 4, Table 5, and Table 6) list the files provided to facilitate client application development for the languages currently supported. These files must be included during compiling and linking of the client application. Despite the limited number of languages supported, there are no technical reasons precluding support for any language that has the

facilities to use Windows dynamic link libraries at run-time. Based on the prototypes provided for

Language	**C/C++ (ANSI)**
Statement	`#include <ImagesCI.h>`
Frequency	Single - mandatory
Location	To be inserted into a globally accessible header file
Description	Required by the compiler in order to define references to the *IMAGES* API.

Table 5-4 *IMAGES* API - C/C++ header file

Language	**Borland Pascal, Delphi**
Statement	`uses ImagesCI;` `library ImagesCI;`
Frequency	Single - mandatory
Location	To be inserted into a globally accessible include file
Description	Required by the linker to resolve references to the *IMAGES* API.

Table 5-5 *IMAGES* API - Borland Pascal uses and library statements

Language	**Visual Basic**
Statement	`Declare Sub ImagesInitialize Lib "ImagesCI" ()` `Declare Sub ImagesGaugeUpdate Lib "ImagesCI" (ByVal szVariableName As String, ByVal nValue As Long)`
Frequency	Single - mandatory
Location	To be inserted into a module with globally visible declarations
Description	Required by the linker to resolve references to the *IMAGES* API.

Table 5-6 *IMAGES* API - Visual Basic library declarations

the languages currently supported, other language specific prototypes could be derived to satisfy compile-time and link-time requirements.

5.3 Integration Process

The integration process consists of three distinct stages. The first stage, application development, involves producing an *IMAGES* enabled program, by harnessing the API during program development. The second stage, gauge specification, involves associating and configuring *IMAGES* gauges with the program variables. In the final stage, program execution, the program is launched, whence interaction between the program, the *IMAGES* server and the *IMAGES* gauges takes place producing the intended result of program measuring, observing and monitoring. These three stages are autonomous operations with a requisite that all antecedent stages be completed.

There are two goals in the application development stage. The first goal, is to take into consideration, at design-time, the API requirements. The intention here is to enable the development team to become familiar with the ease with which the API is used. The second goal is to apply the API integration rules and functions during implementation of the application[4]. Implementation involves providing access to the *IMAGES* client interface dynamic link library using the appropriate solution for the development language being used.

There are three API requirements that must be followed to successfully integrate with *IMAGES*. By including the language specific requirements for making Windows Dynamic Link Library (DLL) calls into the application, references to the API functions in the ImagesCI.DLL file can be resolved by the compiler and/or linker.

The first API requirement is to add the *IMAGES* initialization function or procedure (`ImagesInitialize`) into the application initialization control flow. The second API requirement is to add the gauge update function or procedure (`ImagesGaugeUpdate`) at strategic points within the application. The strategy may vary from project to project; however, a general rule-of-thumb is to place the call at points that require the constant monitoring of variables that would otherwise use a printf(), write or print type of function or procedure.

The gauge specification stage is optional, and over the period of several projects, may even be eliminated by automation due to the learning capabilities of the integrated software agents. If this stage is skipped, default configuration information for the gauges are used at run-time. Otherwise, the result of this stage, which is to produce an *IMAGES Gauge Configuration* (IGC) file from the

4. The "application" refers to the client software that is to be harnessed with the *IMAGES* gauges.

variable list (VAR) file, is used at run-time. This stage is necessary for the developer or end-user to customize the characteristics of the gauges that will be used to measure, observe and monitor the behavior output of the application. By using the *IMAGES Gauge Specification (IGS) Tool* to configure the gauges interactively, this stage is simplified.

During the program execution stage, the efforts of integrating with the *IMAGES* system are realized.

5.4 IMAGES Gauge Specification (IGS) Tool

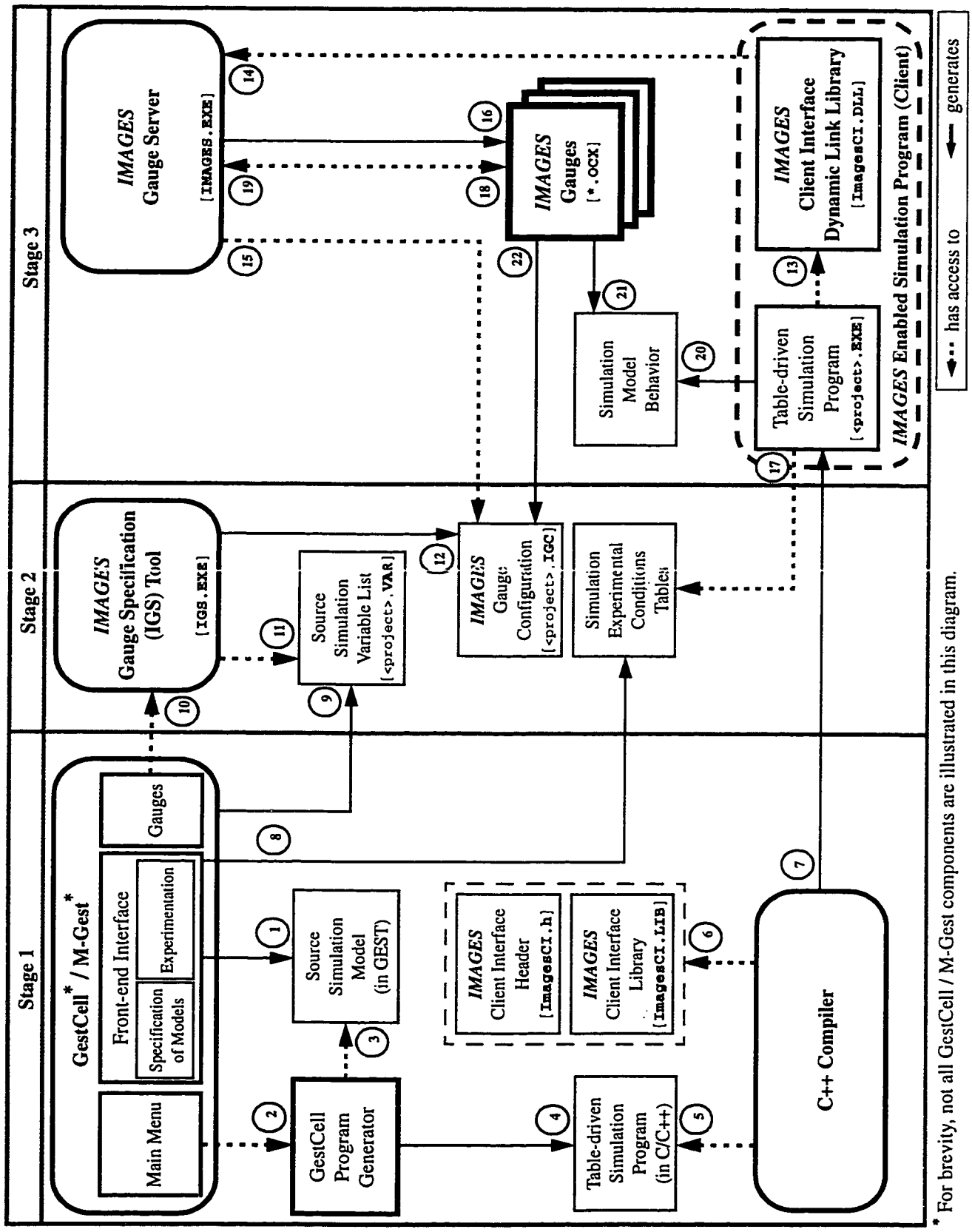
The *IMAGES* Gauge Specification (IGS) tool, which is distinct from the *IMAGES* run-time server, is one software component designed to facilitate the integration of client applications with *IMAGES* gauges. The purpose of this tool is to provide an interactive interface for specifying and configuring *IMAGES* gauges for an explicit list of variables. Once the gauge-to-variable associations and configurations are completed, the appropriate IGC file, for creating *IMAGES* gauges at run-time, is generated. This process can be accomplished manually by a skilled programmer. However, this would require sufficient knowledge of the API, and of the numerous parameter options to successfully integrate *IMAGES* gauges into client applications. The IGS tool shadows the details of configuring and integrating *IMAGES* gauges into client applications.

Figure 5-12 illustrates the role of the IGS tool in an example of an integrated architecture for a simulation environment with *IMAGES*.

5.5 An Integration Example

In Figure 5-12, the integrated architecture of a simulation environment (M-Gest), the IGS tool, the *IMAGES* server and the *IMAGES* gauges is presented. Only one additional step (6) is to be introduced into the normal M-Gest process (steps 1-5,7,8) to produce a table-driven simulation program. The IGS tool's role within the process of integrating *IMAGES* gauges into a GestCell generated simulation program comprises four steps (9-12). The remaining ten steps (13-22) are achieved at run-time by the combined *IMAGES* server, the *IMAGES* enabled simulation program and the *IMAGES* gauges. Each of the twenty-two steps will be described in detail below, with respect to *IMAGES*.

In the first stage, all application development is conducted within the first seven steps of the integration architecture. Step one (1) is to represent a source simulation model as a GEST



* For brevity, not all GestCell / M-Gest components are illustrated in this diagram.

Figure 5-12 Three stage process for the integration of *IMAGES* gauges into an M-Gest generated simulation program

specification. The next step (2) is to launch the GestCell program generator which references the source simulation model (in GEST) (3) to produce a source simulation program in C/C++ (4). In generating the simulation program (4), integration with *IMAGES* requires that the program generator ensure the three statements in Tables 5-1, 5-2 and 5-3 are inserted appropriately into the generated C/C++ code.

Once the simulation program has been augmented according to the rules of these three API statements, it is compiled (5) along with both the *IMAGES* client interface header file and the *IMAGES* client interface library (6) to produce the *IMAGES* enabled simulation program (7). At this stage, no additional application development is required.

The second stage is to prepare all dependencies for successful run-time operation. This includes producing the simulation experimental conditions tables from GestCell/M-Gest (8) and an IGC file. The table-driven simulation program references the experimental conditions at run-time. Also generated by GestCell/M-Gest is a source simulation variable list (VAR) file (9). It is important to note that the VAR file is project dependent and is only used by the IGS tool during this stage of the process. Its contents must reflect the variables to be instrumented with IMAGES gauges, and the update frequency for the gauges at run-time. After launching the IGS tool from either the GestCell/M-Gest environment (10) or directly, the variable list file can be added as a new project (11). Once the new project is added, the project variables can be associated with gauge types that can be configured with their initial run-time characteristics. Resulting from the gauge specification is the generation of a project specific IGC file (12) used by the *IMAGES* server and active gauges to manage the gauges' run-time characteristics. Both the simulation experimental conditions tables and the IGC file define essential run-time operations.

Finally, the effort in the first two stages are realized in the third stage of the integration process. When the table-driven simulation program is launched, it loads the IMAGES client interface dynamic-link library (DLL) (13) to access the *IMAGES* facilities. This step is automatically performed by the `ImagesInitialize` API function. The DLL is automatically initialized at the time it is loaded, and then attempts to establish communication with the *IMAGES* server (14). If successful, the *IMAGES* server will then locate the project specific IGC file (15) and commence the creation of the appropriate instances of the gauges (16, 18, 19). At the same time, the simulation program can load its simulation experimental conditions tables (17). Only then can the

simulation program initiate its simulation model behavior generation, which may continuously trigger the following six steps: 13, 14, 18, 19, 20, and 21. Since there may be redundancy, either one of step twenty or twenty-one may be skipped. Step twenty-two (22) considers the possibility that a gauge's characteristics may be modified at run-time. In this case, the new configuration is stored back into the IGC file (22) for subsequent simulation runs. The new configuration may be further altered by the IGS tool. The overall result of these last ten steps is the integration of *IMAGES* gauges into a table-driven simulation program generated from the M-Gest simulation environment.

6. Conclusion

6.1 Work Achieved

Intelligent Multi-Agent Based Virtual Gauges (*IMAGES*) has three main goals summarized here.

The first goal is to provide agent-based gauges that could relieve an end-user from both the managing and the monitoring of simulation output variables. The *IMAGES* architecture comprises three key components: the gauge specification (IGS) tool, the server, and the gauges. Each of these three key components employs software agents to enhance the usability, both directly and indirectly, of a particular aspect of a gauge.

The IGS tool, presented in Section 4.3.4 and Section 5.5, and demonstrated in Appendix B.1, employs a gauge association agent to minimize the effort in manual gauge configuration and management. By using knowledge about unit-of-measure to gauge-type associations that is either foundational or acquired from *IMAGES* users over time, the gauge association agent can take over the task of associating a gauge type with a variable. Although one could argue that a default gauge type could be used, eventually a manual selection may have to be made. However, unless the user finds it acceptable to observe all variables using a default, for example, air pressure and time using a horizontal bar, this approach to automatic gauge type selection is not useful. The gauge

association agent would automate this task by suggesting appropriate gauge types or autonomously making the selection.

The *IMAGES* server, presented in Section 4.3.2 and demonstrated in Appendix B.2, employs a supervisor agent to manage the dynamic behavior of the other supporting agents. Two of these agents are the communication agent, and the audio agent. The communication agent assumes responsibility for communication between distributed *IMAGES* servers, and more specifically between supervisor agents. Its intelligent behavior involves adapting to changing communication mediums based on several factors such as cost, availability, and reliability. The audio agent manages and monitors for audio files both on the computer it is operating on, and on the computers that are reachable from the *IMAGES* server with which it is registered. It can also auto-configure or suggest audible warnings based on experience acquired from users that have performed such configurations. The supervisor agent is also able to report on the availability status of the registered agents.

The gauges, discussed in Section 4.3.3 and illustrated in Appendix A, use a gauge agent that manifests itself in two ways. The first manifestation is the monitoring of four threshold levels and providing both aural and visual warning indications for each threshold as the measured value approaches from both the negative and positive directions. The second manifestation is the monitoring for measured values that are less than or greater than the respective pre-configured minimum and maximum limits, and adapting the limits to fit the values. The gauges can also employ the statistics, repository, and unit of measure conversion agents that register with the *IMAGES* server's supervisor agent.

These three key *IMAGES* components use software agents to both ease and enhance the usability of the gauges. However, the improved usability is not the only benefit of employing software agents. This leads us to the second goal.

The second goal is to design a highly modular object-oriented architecture that permits advancements in technology to be realized for future ease of maintenance. Software agents, as discussed in Section 2.1, are inherently modular by the fact that they can operate autonomously, independent of any other software. Their communication interface defines the behavior they can exhibit and share with other software or software agents, and it is the means by which they can change their environment. This ability to decouple the agent and its processing from other

elements of a software application represents an ideal level of modularity. However, this alone does not reflect the modularity that *IMAGES* represents, as elaborated in Section 3.1. The modularity of the *IMAGES* architecture is achieved at two levels: the architectural and the functional levels.

Architectural modularity is derived from three distinct attributes that are characteristic of CORBA based architectures: first, by decoupling the gauges from the application they are to instrument; second, by providing a client side interface to an application, that facilitates communication with the gauges; and third, by using a distributed inter-process communication technology, such as OLE, to enable the communication to work. This leads to three key benefits to this architectural model. The first benefit is the elimination of the application's dependency on the gauges. In so doing, any change to the gauges becomes transparent to the application, both in functionality and interface. The second benefit is the feasibility to offer remote observing and monitoring capabilities. The third benefit is the ability to load the gauges only when they are needed. In fact, an application can be integrated or completely decoupled from *IMAGES* by simply toggling a flag on or off, either at design-time or run-time, as suggested in Section 5.2.

Functional modularity is achieved by designing around the use of software agents. Since software agents have autonomous capabilities, they can offer functionality that is independent of the other software components. By separating functionality in this manner, *IMAGES*' functionality can be defined by the software agents that are available at run-time.

The third and most critical goal, is to ease its integration into existing applications, making it a viable advanced software enhancement. By providing a simple set of three rules for integrating *IMAGES* gauges into an application, integration is made easy. The three rules that an application must follow, as detailed in Section 5.3, are:

1. it must include the `ImagesCI` header or unit file for the *IMAGES* Client Interface, depending on the application's implementation language (see Section 5.2)
2. it must invoke the `ImagesInitialize()` function during the initialization of the application (see Section 5.2)
3. it may invoke the `ImagesGaugeUpdate()` function anywhere a gauge is to be updated (see Section 5.2)

These three steps define the requirements for integrating *IMAGES* gauges into an application.

Intelligent Multi-Agent Based Virtual Gauges (*IMAGES*) has been shown to exhibit a high degree of modularity for harnessing future technological advancement within each sub-component throughout the overall system. Its ease of maintenance extends to its ability to be easily integrated into (existing) simulation systems and environments. Its intelligent agent driven behavior enhances the usability and viability of such a tool by helping the user to decide the output characteristics for visualizing behavior generation output. Also, its ability to be customized can satisfy the demands of many users. As such, *IMAGES* is a modular, easy to integrate, agent assisted, and customizable tool for measuring, observing, and monitoring application variables.

Before addressing the question of the next steps for enhancing *IMAGES*, a review of *IMAGES*' current stage of development is prudent. Presently, *IMAGES* offers the following list of features:

- seven gauge types from which to choose from,
- automatic gauge-type selection based on the variable's unit of measure,
- automatic conversion between units of measure,
- both synchronous and asynchronous monitoring of behavior generation,
- logging and playback of measured simulation output variables,
- eight visual and eight aural, fully customizable, warning indicators for four threshold levels,
- adaptive scaling and run-time editing of the range,
- automatic calculation of common statistical information,
- communication support for multi-application conversations,
- run-time gauge configuration both indirectly via the conversation between the application and the gauge, and directly via the gauge's user interface; and
- a stealth mode that allows background operation with warnings occurring at the time of threshold breaches.

The existing functionality offered by *IMAGES* is more than adequate to provide an advanced instrumentation solution for applications and simulation environments alike.

Table 6-1 summarizes the implementation status of the *IMAGES* software, as of the end of the second quarter of 1998. The client-side application interface, the IGS tool, and the server's user interface and gauge manager are implemented. In progress is the implementation of the supervisor

agent, audio agent, the unit of measure conversion knowledgebase, the gauge's interface and its behavior. The server's network communication agent, statistics agent, and unit of measure conversion agent are designed. The remaining sub-components, the server's repository agent and interface management agent functionality are defined, and may not be designed or carried out due to time constraints. Furthermore, these sub-components are not critical to the proof of concept. The *IMAGES* implementation is on a PC, using Microsoft Visual C++.

Component	Subcomponent	Status
Client Interface	Interface definition	*implemented*
	Interface behavior	*implemented*
Server	HMI	*implemented*
	Gauge Manager	*implemented*
	Supervisor Agent	*partly implemented*
	Network Communication Agent	*designed*
	Audio Agent	*partly implemented*
	Repository Agent	*defined*
	Interface Management Agent	*defined*
	Statistics Agent	*designed*
	Unit of Measure Conversion Agent	*designed*
	Unit of Measure Conversion Knowledgebase	*partly implemented*
Gauges	HMI	*integrated*
	Interface definition	*partly implemented*
	Gauge Behavior	*partly implemented*
IGS Tool	HMI	*implemented*
	Gauge Configuration	*implemented*
	Project Manager	*implemented*
	Gauge Association Agent	*partly implemented*
	Gauge Association Knowledgebase	*implemented*

Table 6-1 Software status

6.2 Future Research

I have identified several enhancement opportunities for future consideration. Enhancements that require little effort include, but are not limited to: synchronous/real-time monitoring of behavior generation; data playback controls (e.g. play, rewind, fast forward, etc.); and providing a means (i.e. an editor) for visual customization and creation of new gauges. In the present interface process, the client application must explicitly request and terminate a conversation with *IMAGES*. One enhancement would be to have a single method of updating a gauge for a specific simulation output variable that automatically sets up a conversation, and consequently a gauge if one does not already exist. However, these are nevertheless relatively minor enhancements compared to the following list of possibilities that require more research and development: Other opportunities that require more research and development are:

- support of application interoperability by allowing the output of one application to become the input for another, thus facilitating simulation program interoperability. This would involve scheduling and parallel computing complexity. Software agents employed to adapt to the computational requirements for many interoperating applications could enhance the servicing of the applications, particularly in a distributed environment where the medium for communication is a multi-segment LAN or the Internet
- support of the DARPA Knowledge Query and Manipulation Language (KQML) (Finin et al, 1993), and Knowledge Interchange Format (KIF) (Genesereth and Fikes, 1992), or the Coordination Language (COOL) (Barbuceanu and Fox, 1995), software agent protocols tunneled over the existing OLE automation interface
- support of multiple database formats as an alternative to the existing single repository
- support of re-sizeable and interface customizable gauges at run-time
- simplification of the second stage of the integration process. This can be done by integrating IGS into the simulation environment, or by providing an OLE automation interface for the *IMAGES* Gauge Specification (IGS) tool and enabling OLE automation capabilities in the simulation environment
- enhancement of the IGS with the capability of extracting variables from source code (i.e. C/ C++) without requiring either programmer intervention or an external tool, and
- remote monitoring of the variables using additional LAN and WAN protocols with the ability to suggest an optimal route dynamically. This enhancement would increase scalability and

flexibility; however, it may diminish gauge response due to potential network latency

Finally, the concept of virtual gauges represents not only the decoupling of the back-end interface from an application, but also the dynamic ability to connect a gauge to an application without imposing programming effort on the client application. This additional characteristic is the impetus for my work with *IMAGES*. *IMAGES* will be used as a stepping stone toward the goal of giving the user the ability to point and click (i.e. specify a "hot-link"), wherever a gauge is wanted in a graphical representation of an application or simulation model. With the least guidance from the user in perhaps selecting among several possible variables at the *hot-link*, an enhanced *IMAGES*, or its predecessor, would be automatically employed to manage the visualization of the measured variable. This ultimate goal brings me one step closer to my ideal vision of viable intelligent agent-based virtual gauges as competent and trusted instrumentation assistants to users.

References

Abdullah, Bashar and Tuncer I. Ören. 1995. "Integration of Visual Intelligent Software Agent Gauges (VISAG) into a Simulation Environment." In: Proc. 1995 Summer Computer Simulation Conference, Ottawa, ON., July 24-26, 1995. SCS, San Diego, CA., pp. 269-274.

Abdullah, Bashar and Tuncer I. Ören. 1997. "Enhancement of a Simulation Environment with IMAGES (Intelligent Multi-Agent Based Virtual Gauges)." In: Proc. 1st World Congress on Systems Simulation, Singapore, Sept. 1-3, 1997. SCS, San Diego, CA., pp. 359-363.

Andreoli, Jean-Marc, and François Pacull, and Remo Pareschi. 1997. "XPECT: A Framework for Electronic Commerce." *IEEE Internet Computing* 1:4 (July-Aug.): 40-48.

Barbuceanu, Mihai, and Mark S. Fox. 1995. "COOL: A Language for Describing Coordination in Multi Agent Systems." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, California: AAAI Press, pp. 17-24.

Barbuceanu, Mihai, and Mark S. Fox. 1996. "The Architecture of an Agent Building Shell." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 235-250.

Burke, Robin D., Kristian J. Hammond, and Benjamin C. Young. 1997. "The FindMe Approach to Assisted Browsing." *IEEE Expert Intelligent Systems & Their Applications* 12:4 (July-Aug.): 32-40.

Castelfranchi, Cristiano. 1995. "Commitments: From Individual Intentions to Groups and Organizations." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, California: AAAI Press, pp. 41-48.

Chaib-Draa, B., B. Moulin, R. Mandiau, and P. Millot. 1992. "Trends in distributed artificial intelligence." *Artificial Intelligence Review* 1:6 (June): 35-66.

Date, C.J. 1990. "An Introduction to Database Systems." vol.1, 5th ed., Reading, MA.: Addison-Wesley Inc., pp. 525-560.

Finin, Tim, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, Donald McKay, James McGuire, Richard Pelavin, Stuart Shapiro, Chris Beck. 1993. "DRAFT Specification of the KQML Agent-Communication Language plus example agent policies and architectures." The DARPA Knowledge Sharing Initiative External Interfaces Working Group. June 15, 1993.

Fisher, Michael. 1996. "Representing and Executing Agent-Based Systems." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 307-323.

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, Addison-Wesley, 1995.

Generic Logic Inc. 1998. GLG Toolkit: Real-Time Dynamics for C/C++ and Java. Brighton, MA, URL: http://www.genlogic.com/java.html

Genesereth, Michael R. and Richard E. Fikes. 1992. "Knowledge Interchange Format Version 3.0 Reference Manual." Report Logic-92-1. Logic Group. June 1992. Computer Science Department, Stanford University, Stanford, CA. 94305.

Genesereth, Michael R. and S.P. Ketchpel. 1994. "Software Agents." *Communications of the ACM* 37:7 (July): 48-53 and 147.

Gray, N. A. B. 1994. Programming with Class: A Practical Introduction to Object Oriented Programming with C++. Chichester, England, John Wiley & Sons Ltd., 1994.

Guha, R.V. and D.B. Lenat. 1994. "Enabling Agents to Work Together." *Communications of the ACM* 37: 7 (July): 127-142.

Haley Enterprise Inc. 1991. Data-Driven Backward Chaining. Sewickley, PA. Reprinted from Proc. of the Second Annual CLIPS Conference, NASA Johnson Space Center, Houston, TX. Sept. 1991. URL: http://www.haley.com/

Hollan, James D., Edwin L. Hutchins, Timothy P. McCandless, Mark Rosenstein, and Louis Weitzman. 1987. "Graphic Interfaces for Simulation." Advances in Man-Machine Systems Research. vol.3, 1987. Editor: William B. Rouse, JAI Press, pp. 129-163.

Hyperception Inc. 1997. "Virtual Instrumentation." 31 March, 1997. URL: http:// www.hyperception.com/j98htm/prod03.htm

Iglesias, Carlos A., José C. González, and Juan R. Velasco. 1996. "MIX: A General Purpose Multiagent Architecture." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 251-266.

JavaSoft Inc. 1996. JavaBeans 1.01 API Specification. Oct. 10 1996. URL: http:// www.javasoft.com/docs/beans.101.pdf

Kautz, Henry A., Bart Selman, and Michael Coen. 1994. "Bottom-up Design of Software Agents." *Communications of the ACM* 37:7 (July): 143-147.

Krogh, Christen. 1996. "The Rights of Agents." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 1-16.

Krulwich, Bruce. 1997. "Automating the Internet: Agents as User Surrogates." *IEEE Internet Computing* 1:4 (July-Aug.): 34-38.

Law, Averill M., and W. David Kelton. 1991. Simulation Modeling & Analysis. 2nd ed. New York, N.Y.: McGraw-Hill Inc., pp. 522-574.

Luck, Michael, and Mark d'Inverno. 1995. "A Formal Framework for Agency and Autonomy." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, 12-14 June, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 254-260.

Maes, P. 1994. "Agents that Reduce Work and Information Overload." *Communications of the ACM* 37: 7 (July): 31-40 and 146.

Malheiro, Benedita, and Eugénio Oliveira. 1996. "Consistency and Context Management in a Multi-Agent Belief Revision Testbed." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 361-375.

MathWorks Inc. 1997. MATLAB: Accelerating Innovation and Development. Natick, MA., URL: http://www.mathworks.com/products/matlab/

Microsoft Press. 1997. Automation: Programmer's Reference. Redmond, WA: Microsoft Press, pp. 3-12.

Moffat, David, and Nico H. Frijda. 1995. "Where there's a *Will* there's an Agent." Lecture Notes in Artificial Intelligence vol. 890. In: Proc. of Intelligent Agents - Agent Theories, Architectures, and Languages. ECAI-94 Workshop. Amsterdam, The Netherlands. Aug. 8-9 1994. Editors: Michael J. Wooldridge and Nicholas R. Jennings. Berlin, Germany: Springer-Verlag, pp. 245-260.

Norman, D.A. 1994. "How Might People Interact with Agents." *Communications of the ACM* 37:7 (July): 68-71.

Norman, Timothy J., and Derek Long. 1995. "Goal Creation in Motivated Agents." Lecture Notes in Artificial Intelligence vol. 890. In: Proc. of Intelligent Agents - Agent Theories, Architectures, and Languages. ECAI-94 Workshop. Amsterdam, The Netherlands. Aug. 8-9 1994. Editors: Michael J. Wooldridge and Nicholas R. Jennings. Berlin, Germany: Springer-Verlag, pp. 277-290.

Numaoka, Chisato. 1995. "Introducing the Blind Hunger Dilemma: Agents' Properties and Performance." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California. June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 290-296.

OMG. 1996. The Common Object Request Broker: Architecture and Specification. v2.0. July 1996. URL: http://www.omg.org/corba/corbiiop.htm

Ören, T.I. 1994. "Artificial Intelligence in Simulation." Annals of Operations Research 53 (1994): 287-319.

Pressman, Roger S., 1987. Software Engineering: A Practitioner's Approach. 2nd ed. New York, N.Y.: McGraw-Hill Inc., pp. 19-29, 50-53, and 216-238.

ProtoView Development Corp. 1994. ProtoView Interface Component Set (PICS). Dayton, NJ., URL: http://www.protoview.com/activexsuite/default.htm

Rao, Anand S., and Michael P. Georgeff. 1995. "BDI Agents: From Theory to Practice." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California. June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 312-319.

Redmond III, Frank E. 1997. DCOM: Microsoft Distributed Component Object Model. Foster City, CA: IDG Books Worldwide Inc., pp. 3-23.

Rodriguez, Lawrence. 1997. "Java: The Next Generation - Java Beans." *Java Developer's Journal Online* 1:6 (Jan.): http://www.sys-con.com/java/javajan/beans.htm

Sánchez, J. Alfredo, Flávio S. Azevedo, and John J. Leggett. 1995. "PARAgente: Exploring the Issues in Agent-Based User Interfaces." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 320-327.

Sen, Sandip. 1997. "Developing an Automated Distributed Meeting Scheduler." *IEEE Expert Intelligent Systems & Their Applications* 12:4 (July-Aug.): 41-45.

Sichman, Jaime S., and Yves Demazeau. 1995. "Exploiting Social Reasoning to Deal with Agency Level Inconsistency" In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 352-359.

Sun Microsystems. 1996. JavaBeans Component APIs for Java. Jul. 1996. URL: http:// java.sun.com/products/jdk/1.1/docs/api/Package-java.beans.html

Tambe, Milind, and Paul S. Rosenbloom. 1996. "Architectures for Agents that Track Other Agents in Multi-Agent Worlds." In: Proc. Intelligent Agents II - Agent Theories, Architectures, and Languages. IJCAI'95 Workshop (ATAL). Montréal, Canada, Aug. 19-20, 1995. Editors: M. Wooldridge, J. P. Müller, and M. Tambe. Berlin, Germany: Springer, pp. 156-170.

Taylor, Richard N., Kari A. Nies, Gregory A. Bolcer, Craig A. MacFarlane, Gregory F. Johnson, and Kenneth M. Anderson. 1994. Supporting Separations of Concerns and Concurrency in the Chiron-1 User Interface System. UCI Technical Report 94-12. Department of Information and Computer Science, University of California, Irvine, CA., 92717-3425, Northrop Corp., Pico Rivera, CA., March 11, 1994.

Youssefmir, Michael, and Bernardo Huberman. 1995. "Resource Contention in Multiagent Systems." In: Proc. First International Conference on Multi-Agent Systems. San Francisco, California, June 12-14, 1995. General Chair: Victor Lesser. Menlo Park, CA.: AAAI Press, pp. 398-403.

Appendices

Appendix A:Gauges of IMAGES A1
Appendix B:IMAGES User-System Interface. B1
Appendix C:Integration Example With A Windows Application. C1
Appendix D:Development Requirements D1
Appendix E:Glossary of Terms E1

Appendix A: Gauges of *IMAGES*

The *IMAGES* gauges are based on the GMS instrumentation toolset available from *Global Majic Software*[1]. The GMS toolset provides the gauge interface and customization facilities, while *IMAGES* provides both the software agent automation, as described in section "2.1 Introduction", and the virtual attributes, as described in section "3.1 Introduction".

On the following pages of this appendix is a sample set of gauges that can be used and customized with the *IMAGES* system. There is a selection of seven basic gauge types: circular and arc dials (see Figure A-1), odometer (see Figure A-2), digital LED display (see Figure A-3), toggle indicator (see Figure A-4), horizontal and vertical bars (see Figure A-6, and Figure A-8), plotter (see Figure A-7), and a thermometer (see Figure A-8). Each gauge type can be customized, resulting in countless variations of gauge interfaces (see Figure A-5). The samples provided here are snapshots in time of the gauges in operation. Some of the gauges are shown with their default attributes, while others, such as the vertical and horizontal bar variations, are customized to reflect a particular application for the gauge.

---

1. GMS can be contacted via the internet at the URL address http://www.globalmajic.com or by e-mail at gms@globalmajic.com. I do not endorse GMS or its products and I am not an employee of GMS.

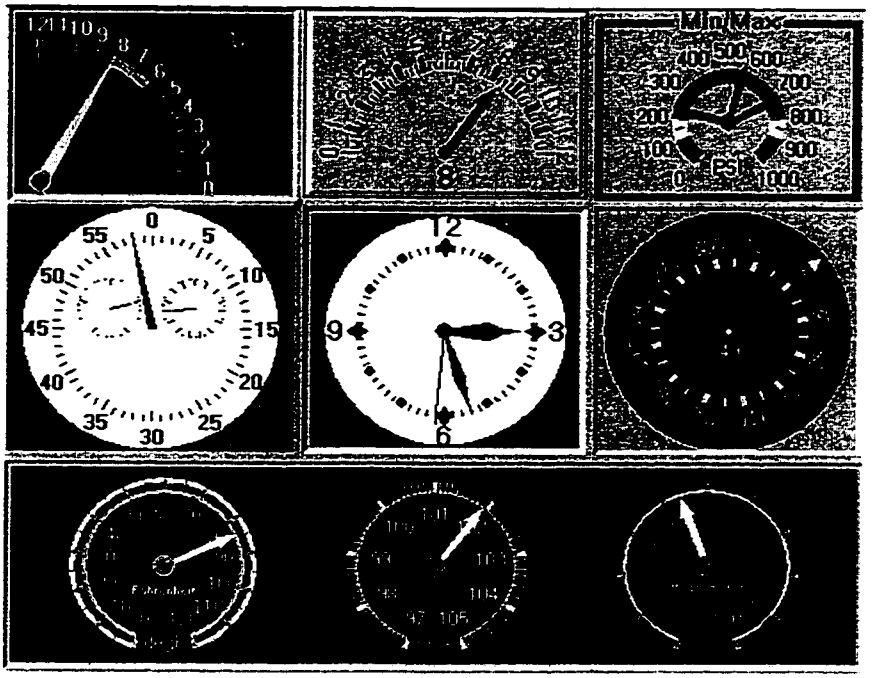


Figure A-1 Sample application specific circular and arc dials.

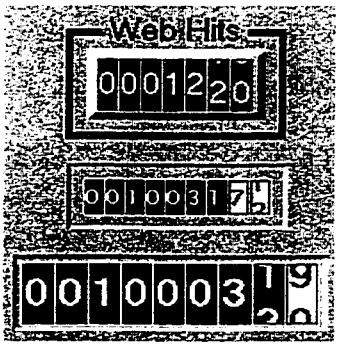


Figure A-2 A sample of odometers.

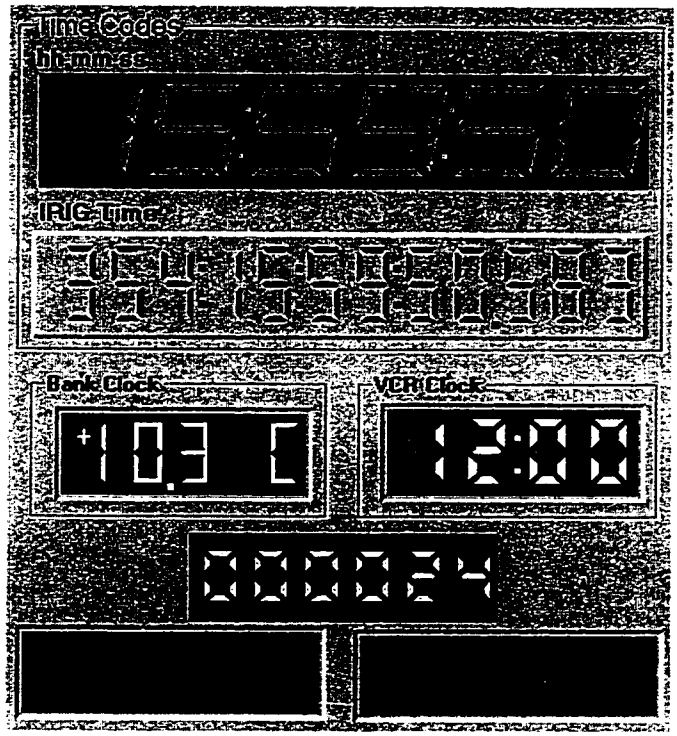


Figure A-3 A sample of digital LED displays.

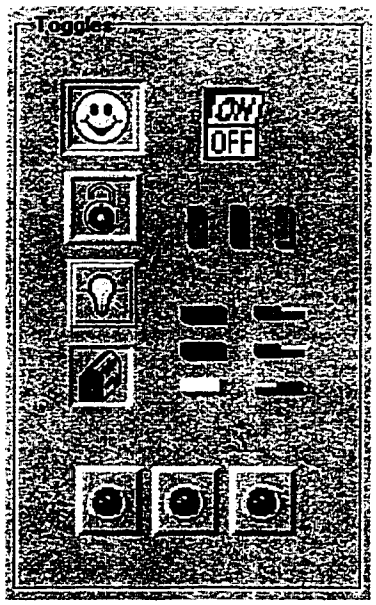


Figure A-4 A sample of toggle indicators (i.e. on/off).

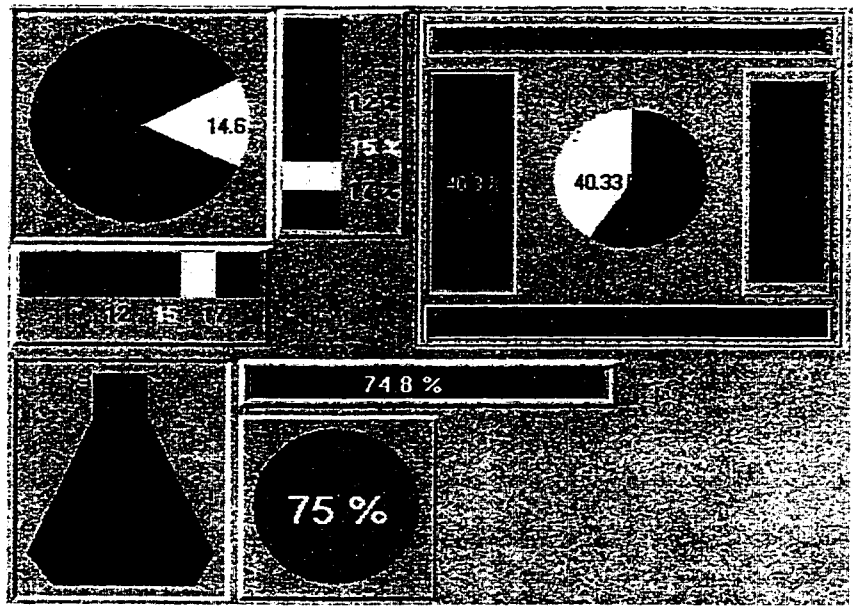


Figure A-5 Both standard and user customized percentage indicators.

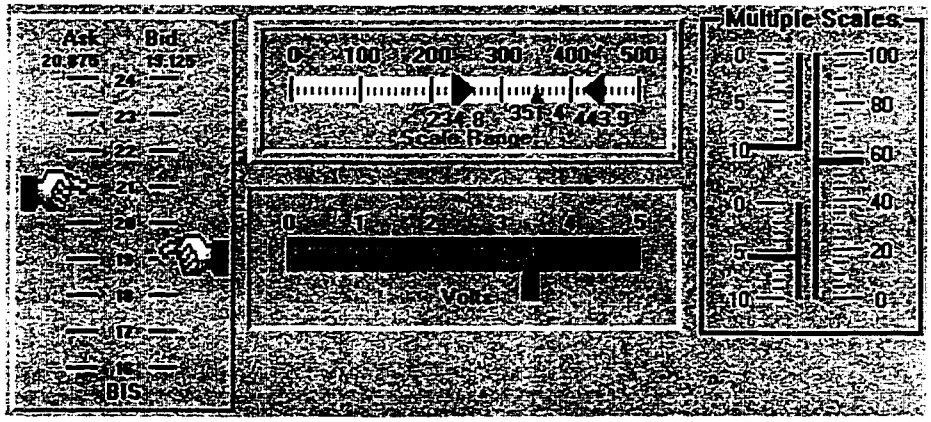


Figure A-6 A sample of horizontal and vertical bars.

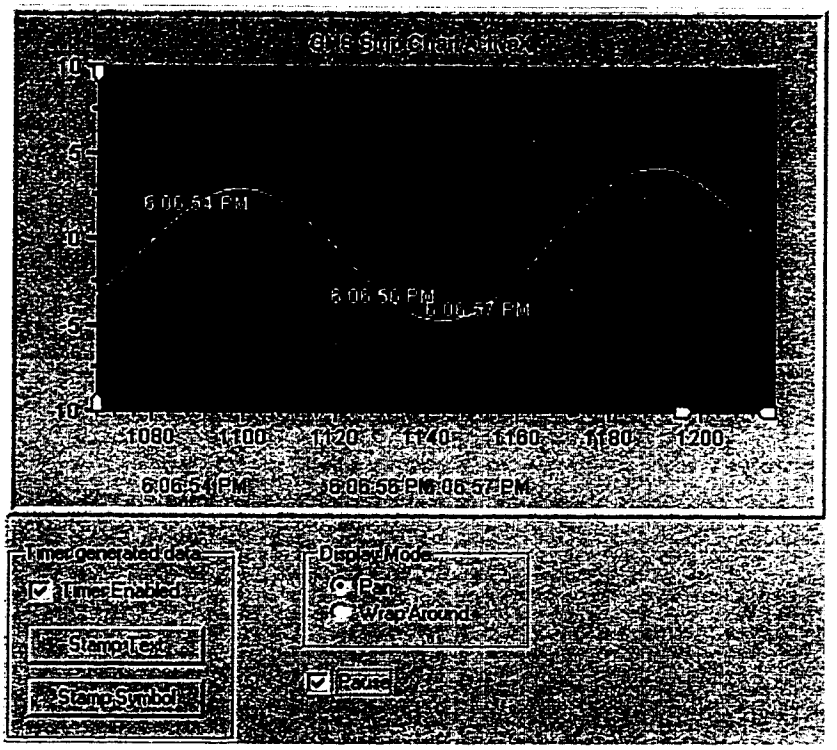


Figure A-7 A sample of the plotter and its interactive controls.

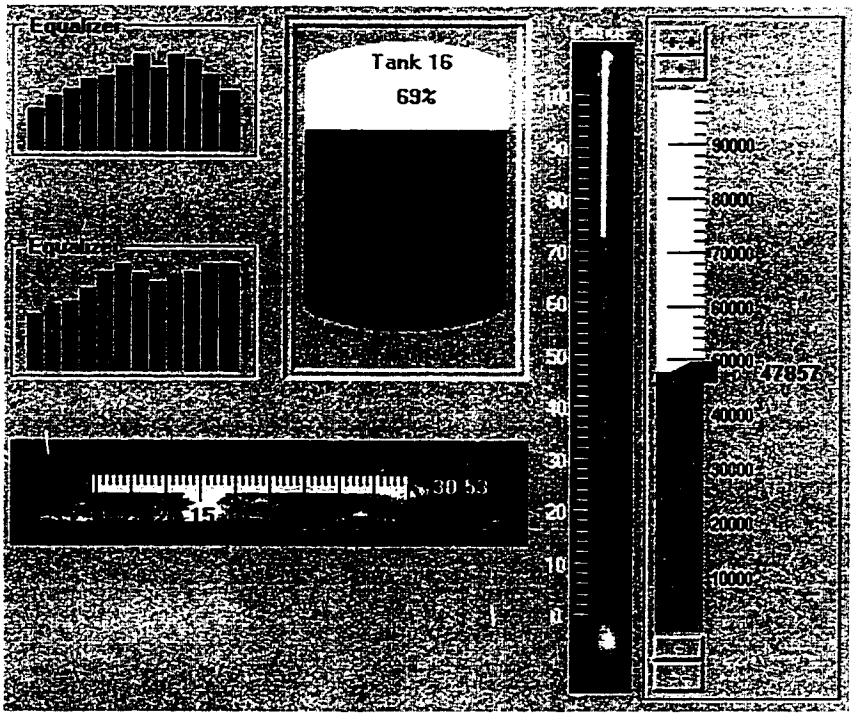


Figure A-8 A sample of horizontal and vertical bar variations.

Appendix B: *IMAGES* User-System Interface

There are three visible components in the *IMAGES* system. They are the Gauge Specification (IGS) tool, the server, and the gauges. The gauge types and a sample of the variations that can be achieved with them have already been presented in "Appendix A: Gauges of IMAGES". Presented here is the *IMAGES* specific configuration interface.

B.1 IGS Tool

Figure B-1 illustrates the IMAGES Gauge Specification (IGS) tool user-interface showing two projects ('MyProject01' and 'TestFile2'), the IGS preferences, all available audio files, and supported gauge types for a specific user (Developer).

Figure B-2 shows the preferences that are applicable to the use of the IGS tool. One of the most important features in the IGS tool is the facility that enables a user to indicate whether or not they prefer suggestions for gauge associations, and furthermore, whether their associations should be used to build up the gauge type associations knowledgebase. Consequently, the IGS tool has access to the the gauge association agent to deliver this capability.

Figure B-3 demonstrates a project with numerous variables. Some variables are associated with default gauge configurations, while some are explicitly specified in the project's variable list (VAR) file.

In Figure B-4, the IGS tool user-interface shows the attribute groups common to all variable configurations: the interface, range and thresholds, warning indicators, log file or repository, knowledgebase for unit of measure conversions, statistics calculations and equation editor. The basic or minimum configuration for the variable selected from on the left-hand-side pane is shown in the right-hand-side pane.

Figure B-5 shows the gauge interface attribute group.

Figure B-6 shows the range and thresholds attribute group.

Figure B-7 shows the warning indicators attribute group.

In Figure B-8, the IGS tool presents the log file or repository attribute group. Only the three ASCII options, indicated in the form, are currently supported by *IMAGES*.

As one of its capabilities, the unit of measure conversion agent automatically prefixes user specified basic units of measure. For example, if the unit *second* is added to the unit of measure conversion knowledgebase, the agent automatically configures to allow measurement of millisecond, microsecond, nanosecond and so on. A full set of SI prefixes that are applied to user specified units of measure are listed in Table B-1. The unit of measure conversion interface in the IGS tool is shown in Figure B-9.

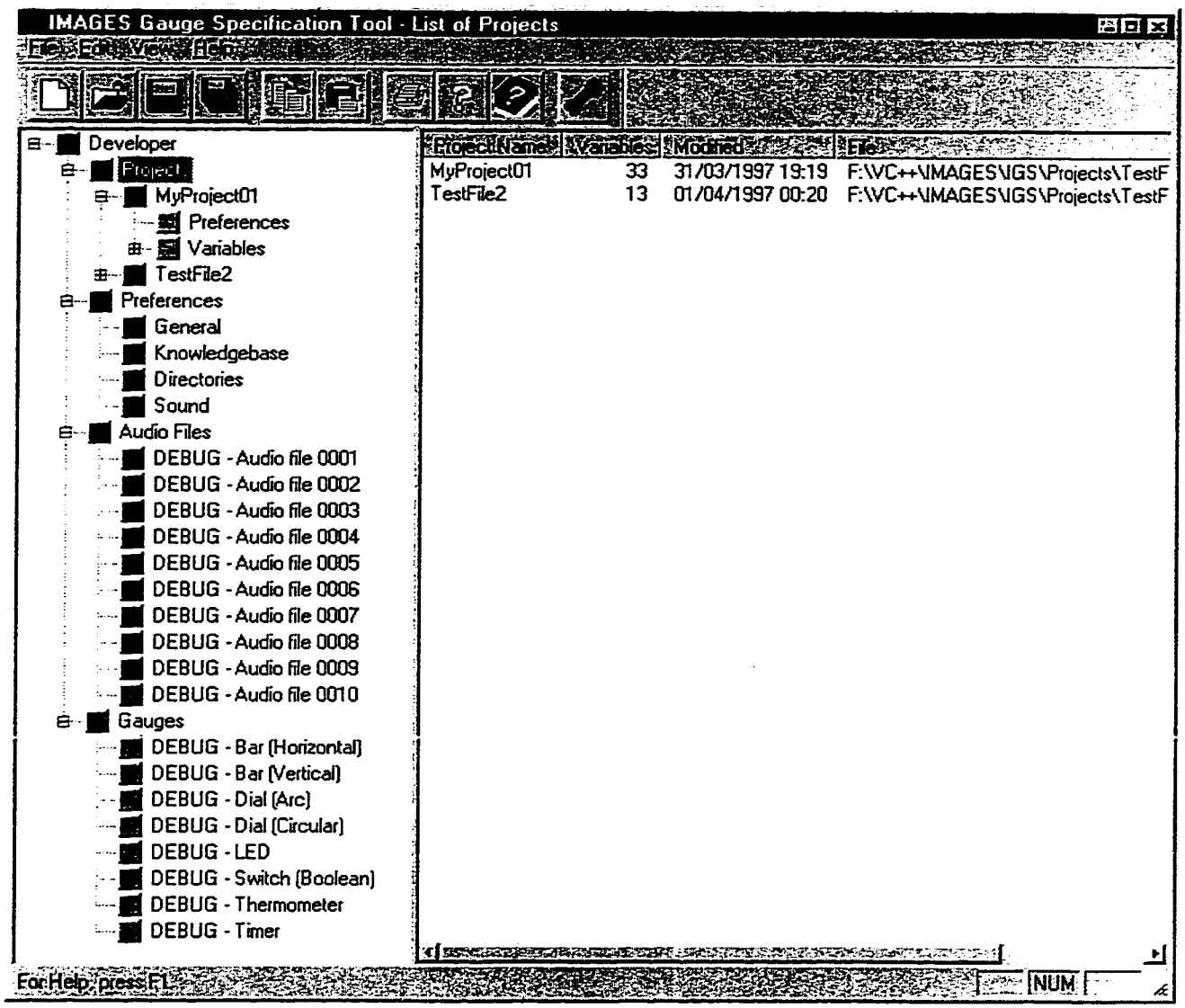


Figure B-1 The IGS tool user-interface showing two projects.

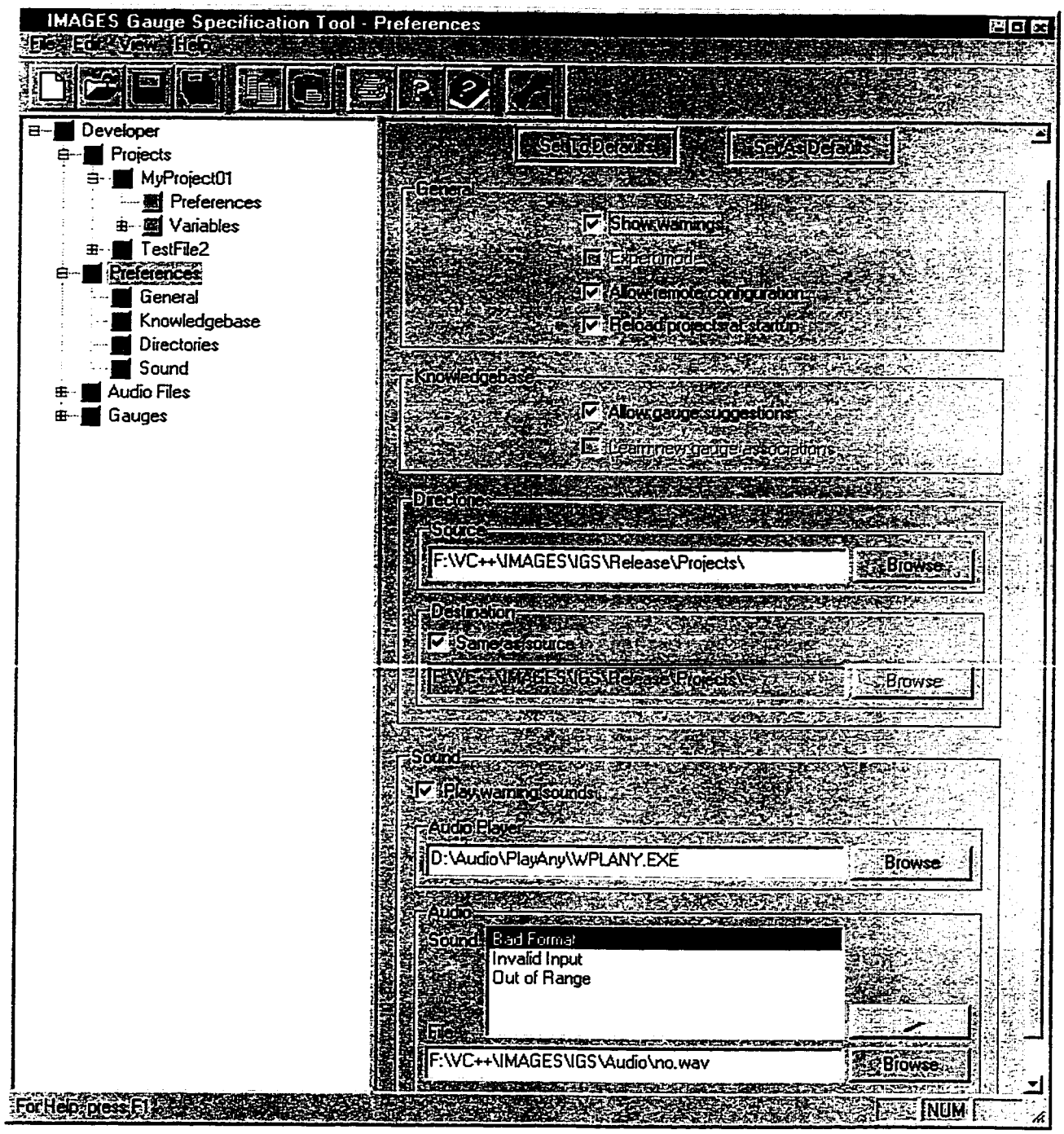


Figure B-2 The IGS tool preferences.

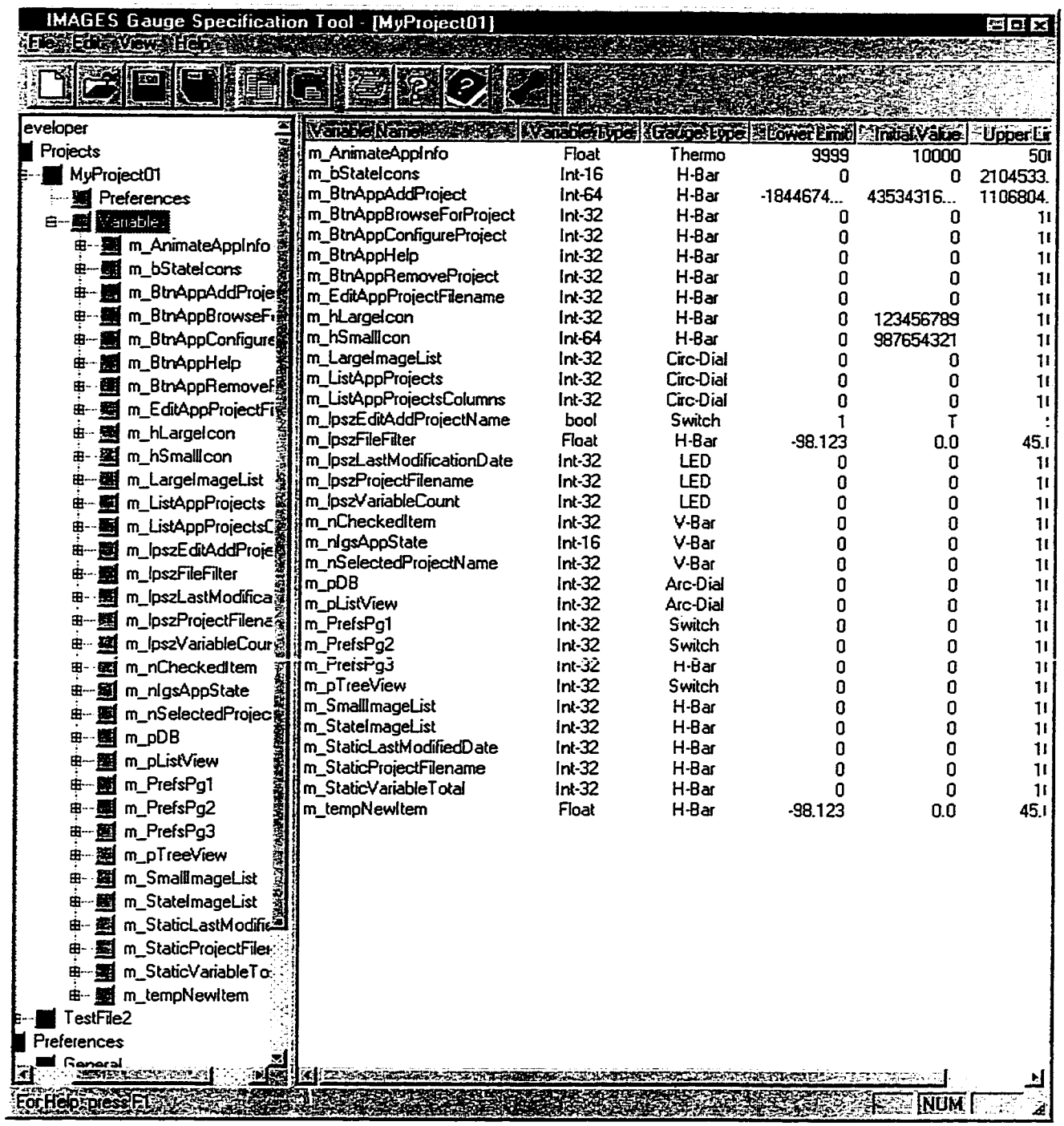


Figure B-3 The IGS tool listing the variables of a project.

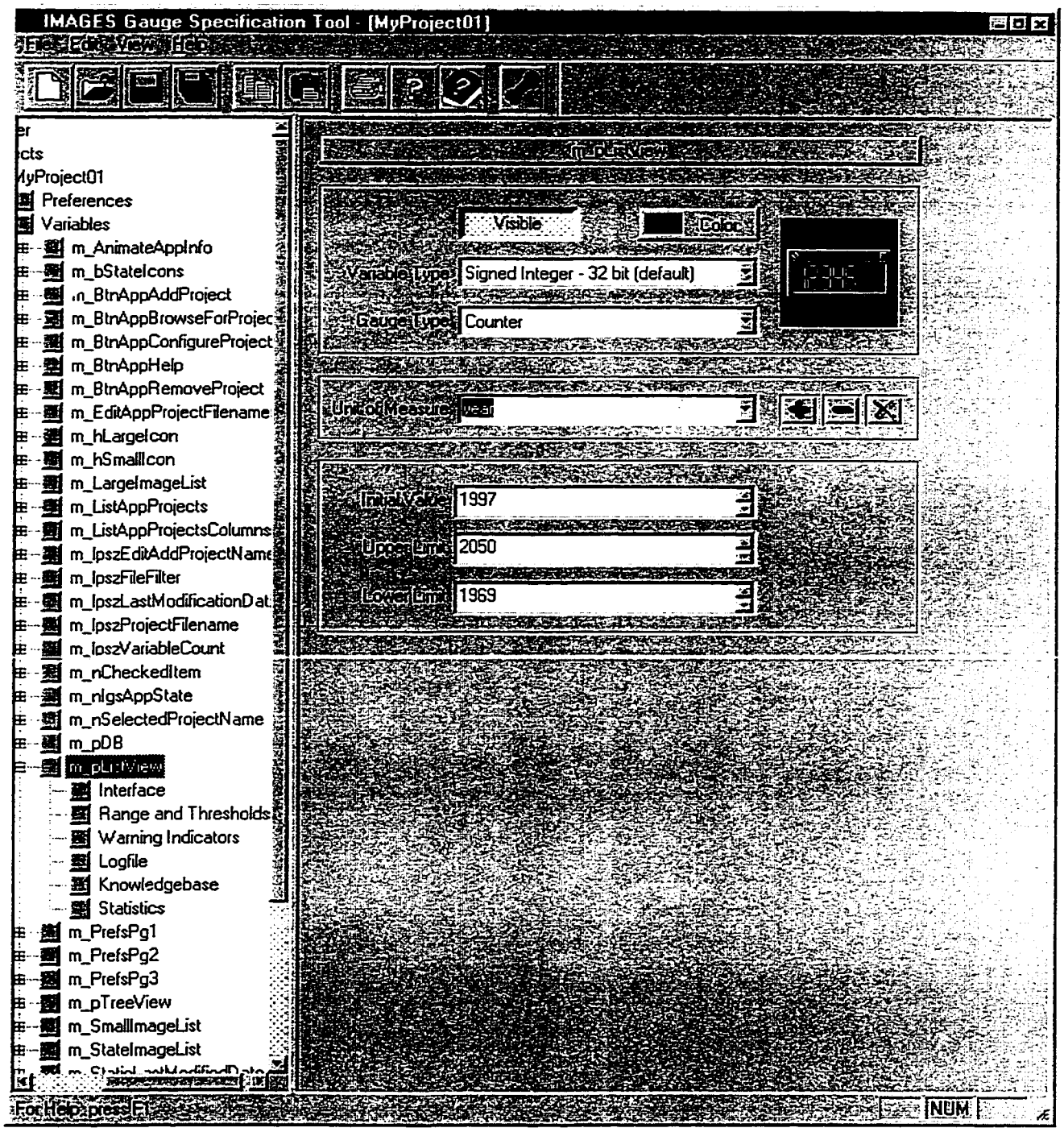


Figure B-4 The IGS tool common variable configuration attributes.

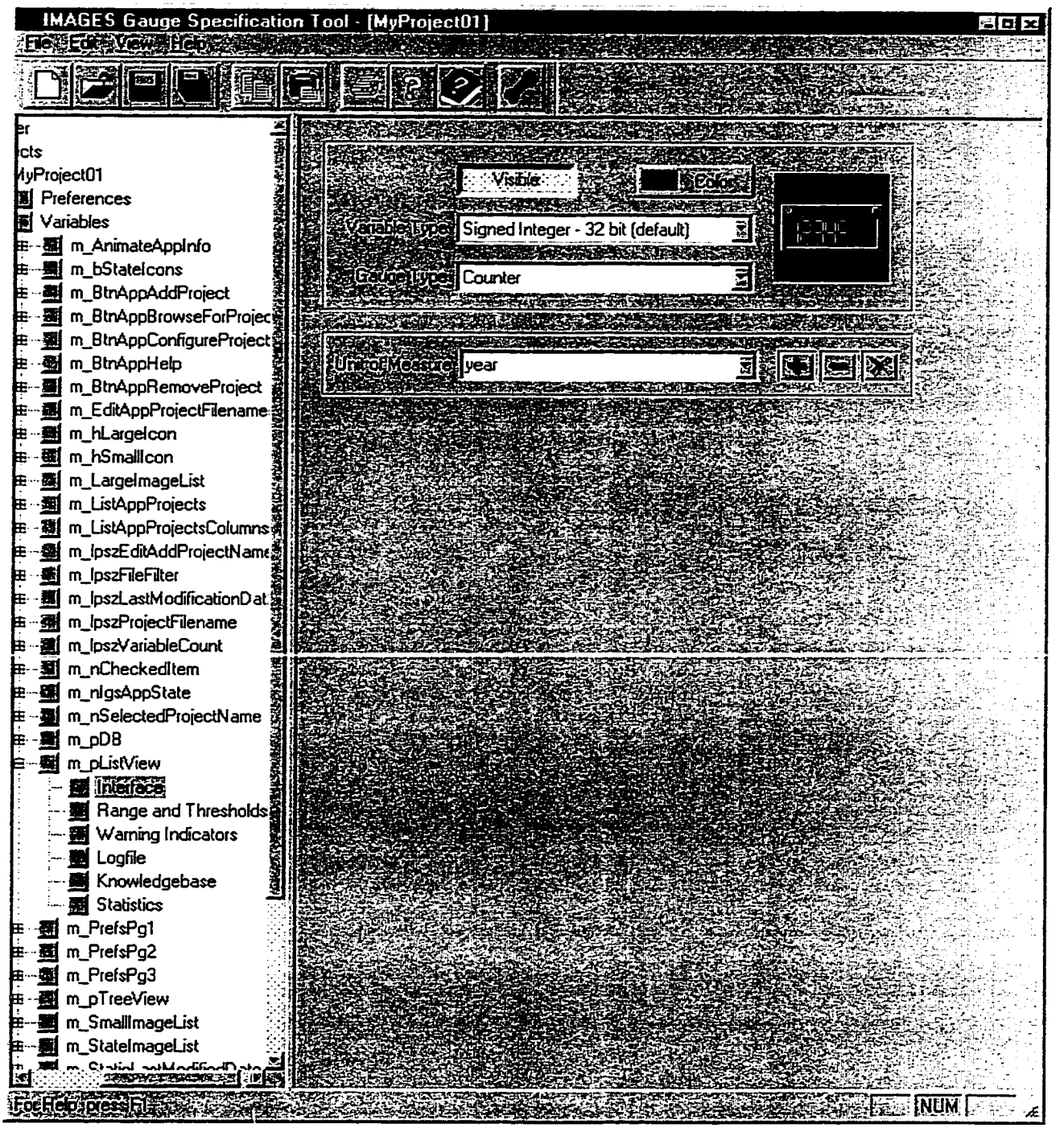


Figure B-5 The IGS tool gauge interface attributes.

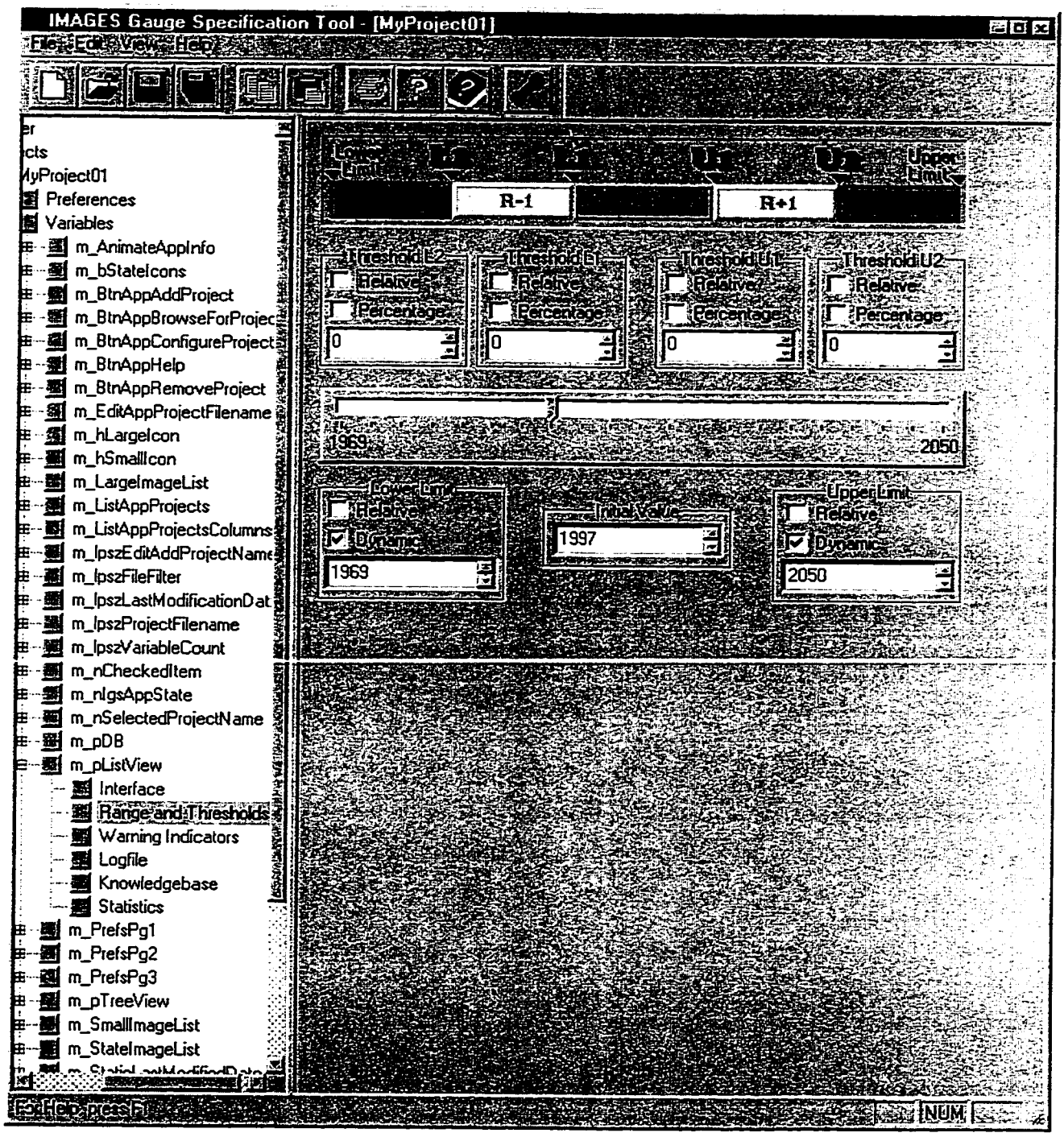


Figure B-6 The IGS tool range and thresholds attributes.

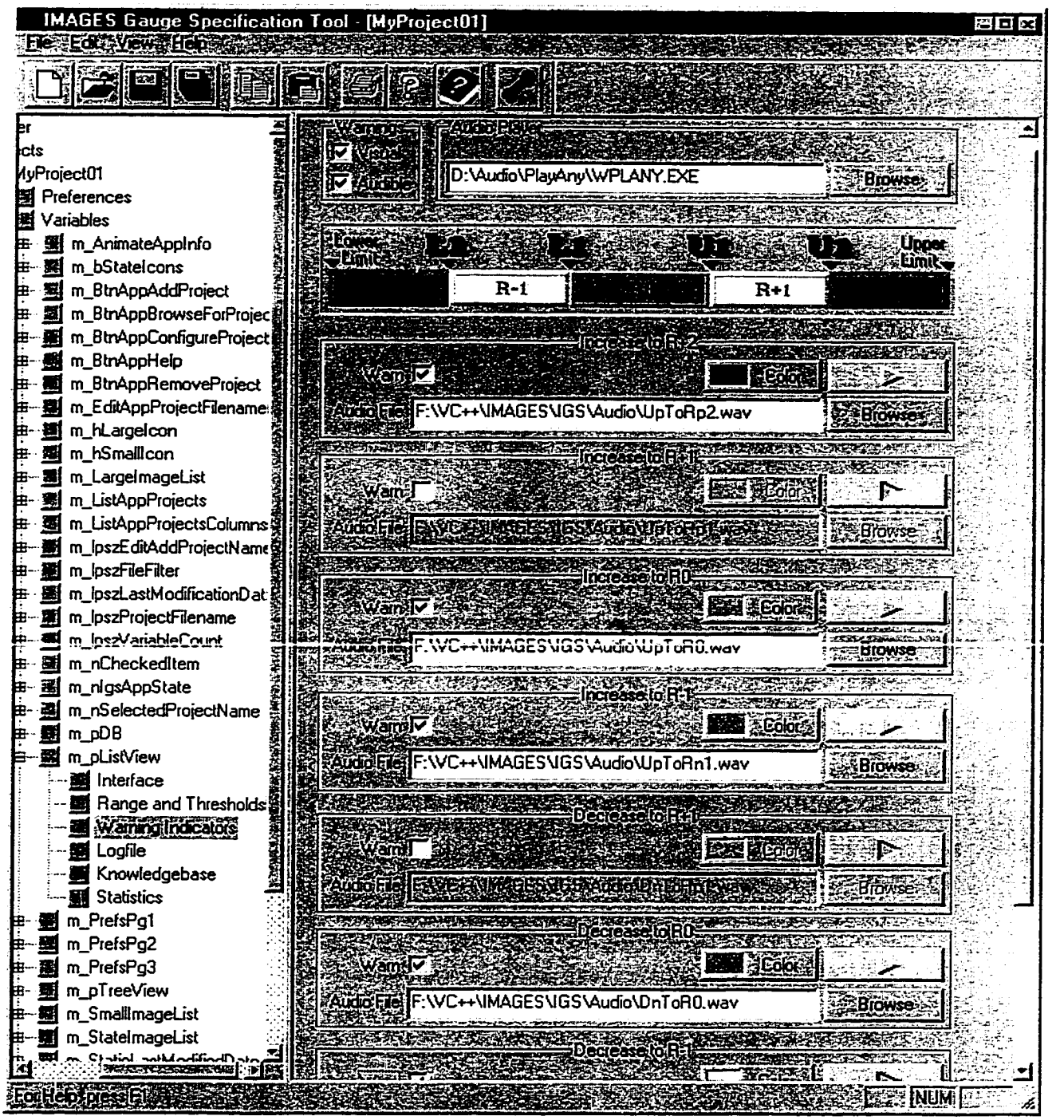


Figure B-7 The IGS tool warning indicators attributes.

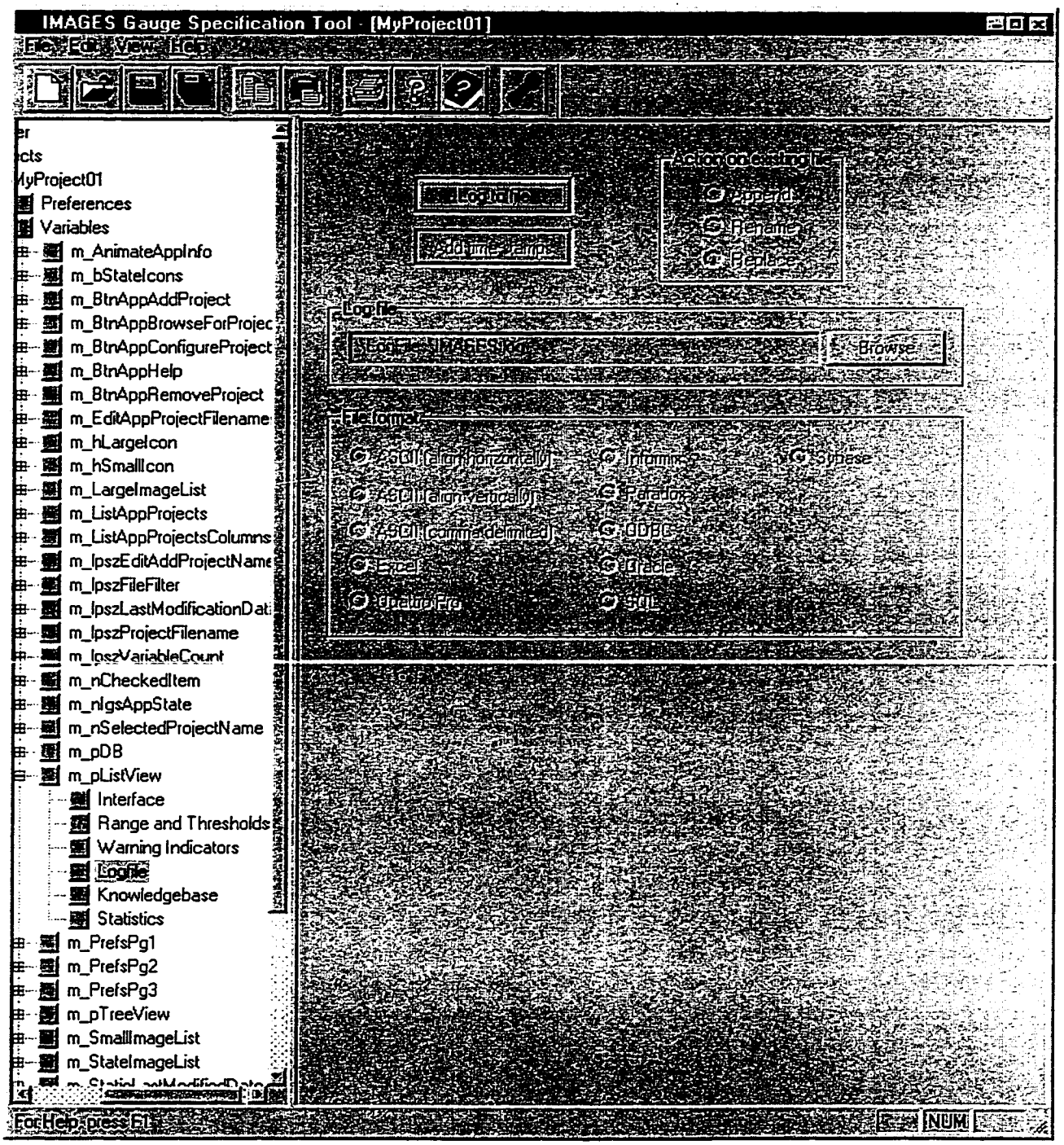


Figure B-8 The IGS tool repository attribute group.

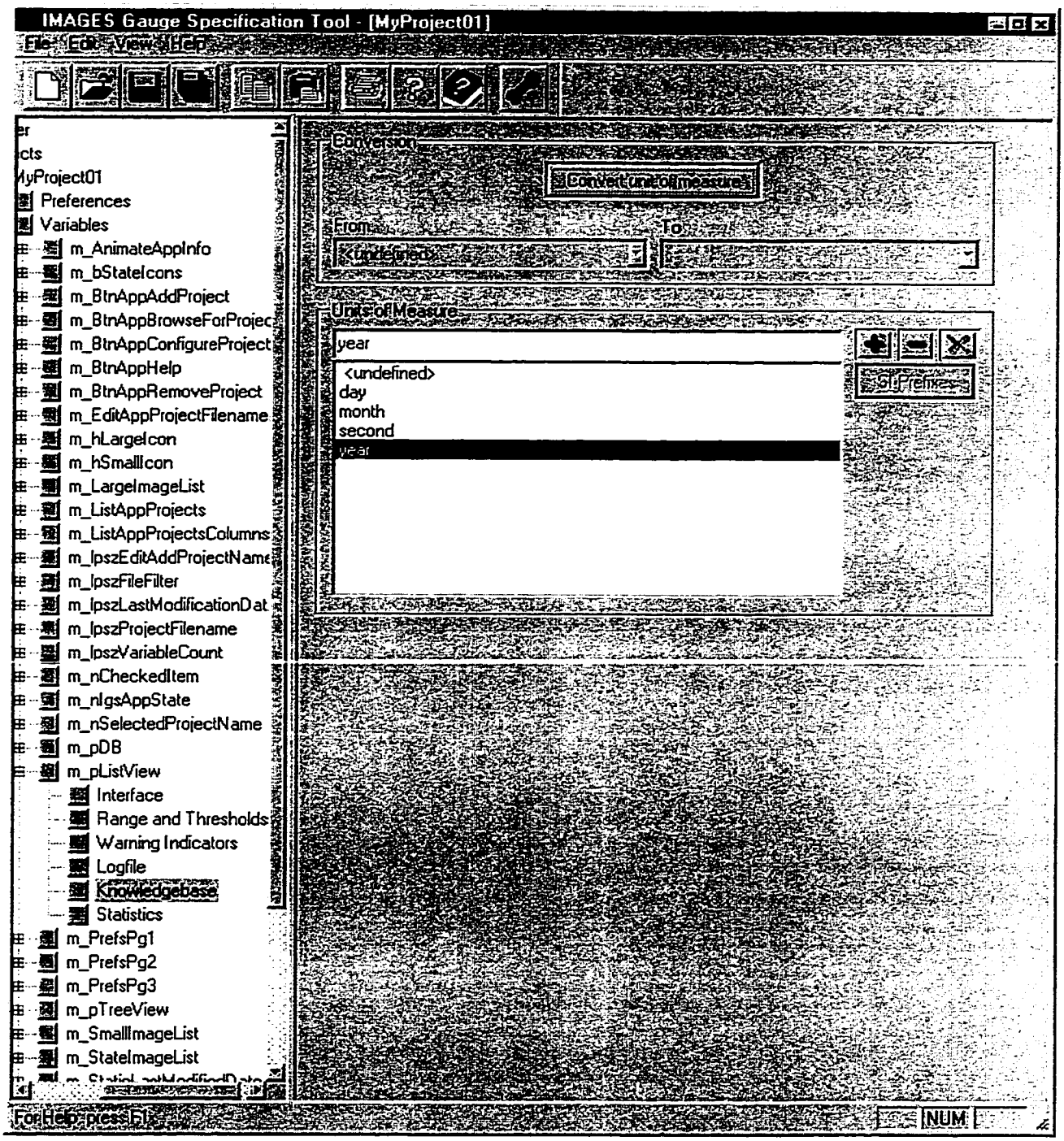


Figure B-9 The IGS tool unit of measure conversion attributes.

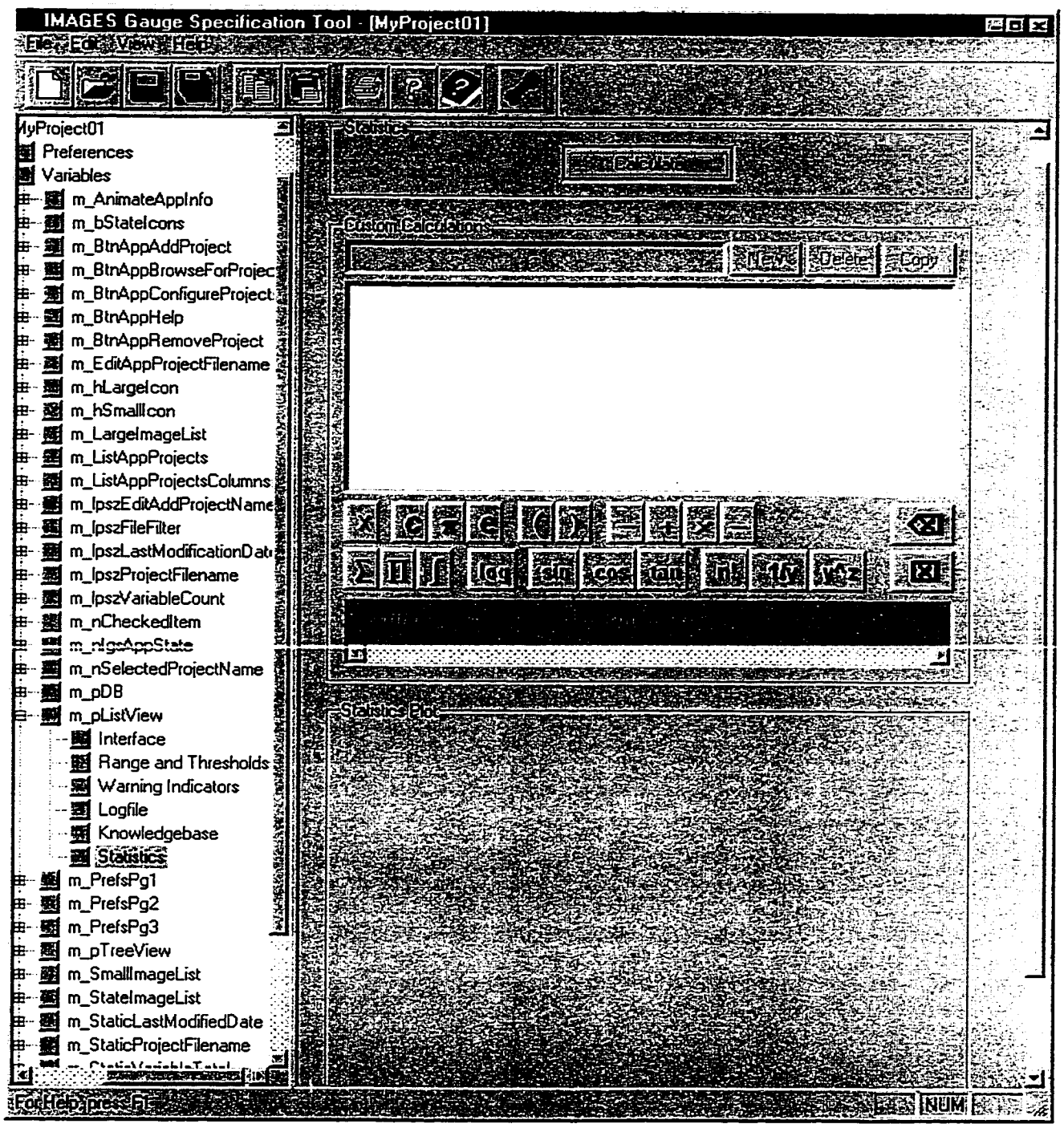


Figure B-10 The IGS tool statistics calculations and equation editor.

Power	SI Prefix	Power	SI Prefix
10^{-18}	*atto*	10^{18}	*exa*
10^{-15}	*femto*	10^{15}	*peta*
10^{-12}	*pico*	10^{12}	*tera*
10^{-9}	*nano*	10^{9}	*giga*
10^{-6}	*micro*	10^{6}	*mega*
10^{-3}	*milli*	10^{3}	*kilo*
10^{-2}	*centi*	10^{2}	*hecto*
10^{-1}	*deci*	10^{1}	*deca*

Table B-1 Unit of measure SI prefixes

In Figure B-10, the *IMAGES* Gauge Specification (IGS) tool user-interface shows the statistics calculations and equation editor attribute group. The variables n, x and y are the number of measured samples, the measured variable value and the measurement time stamp respectively, allowing for time based calculations.

Figure B-11, B-12 and B-13 present an example of the mean of a measured variable. Figure B-11

$$z = \frac{\sum_{i=1}^{n} x_{y_i}}{n}$$

Figure B-11 Mathematical notation for the mean of a measured variable.

illustrates the expression in mathematical notation. Figure B-12 illustrates the sequence of buttons that are to be selected in the IGS tool's equation editor to specify the mean expression. Neither the summation index, nor the summation interval are explicitly specified. When the summation or product operators are used, the statistics agent automatically sums or multiplies, respectively, over

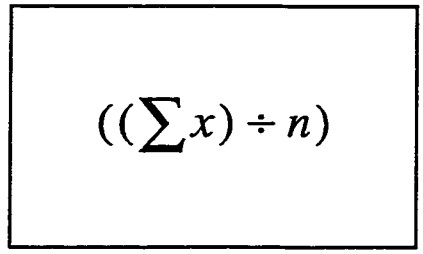


Figure B-12 IGS tool button sequence to specify the mean expression.

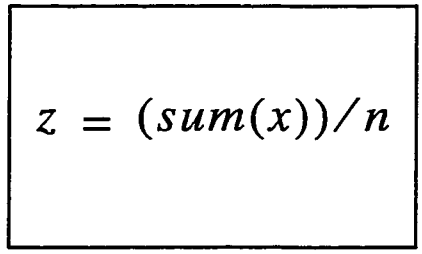


Figure B-13 The mean of a measured variable expression as it would appear in the IGS tool's statistics preview window.

the entire set of sampled values. Figure B-13 illustrates the expression as it would appear in the IGS tool's preview window.

Figure B-14, B-15 and B-16 present the average measurement over a measured time interval.

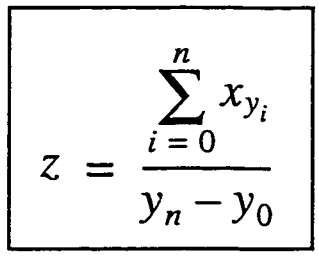


Figure B-14 Mathematical notation of the expression for the average measurement over a measured interval.

Figure B-14 is the expression in mathematical notation. Figure B-14 illustrates the sequence of

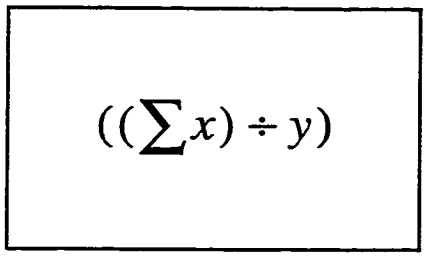


Figure B-15 IGS tool button sequence to specify the expression for the average measurement over a measured interval.

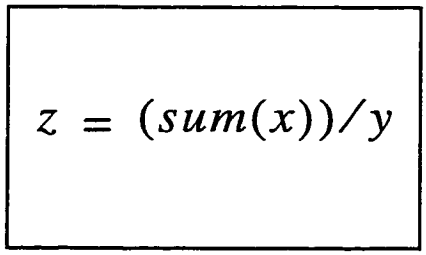


Figure B-16 The average measurement over a time interval expression as it would appear in the IGS tool's statistics preview window.

buttons that are to be selected in the IGS tool's equation editor to specify the expression. On the right is how the equation would appear in the preview window.

B.2 Server

The *IMAGES* Server can be viewed as the hub of the entire *IMAGES* functionality. That is, communication between the client application and the gauges occurs via the server. As a result, the server has the ability to analyze the protocol communication between the client application and the gauges. Also, the server can monitor the status of each gauge as a means for diagnosing problems with the *IMAGES* setup and overall component communication.

In Figure B-17, the server's user interface is illustrated. Visually, it shares many characteristics

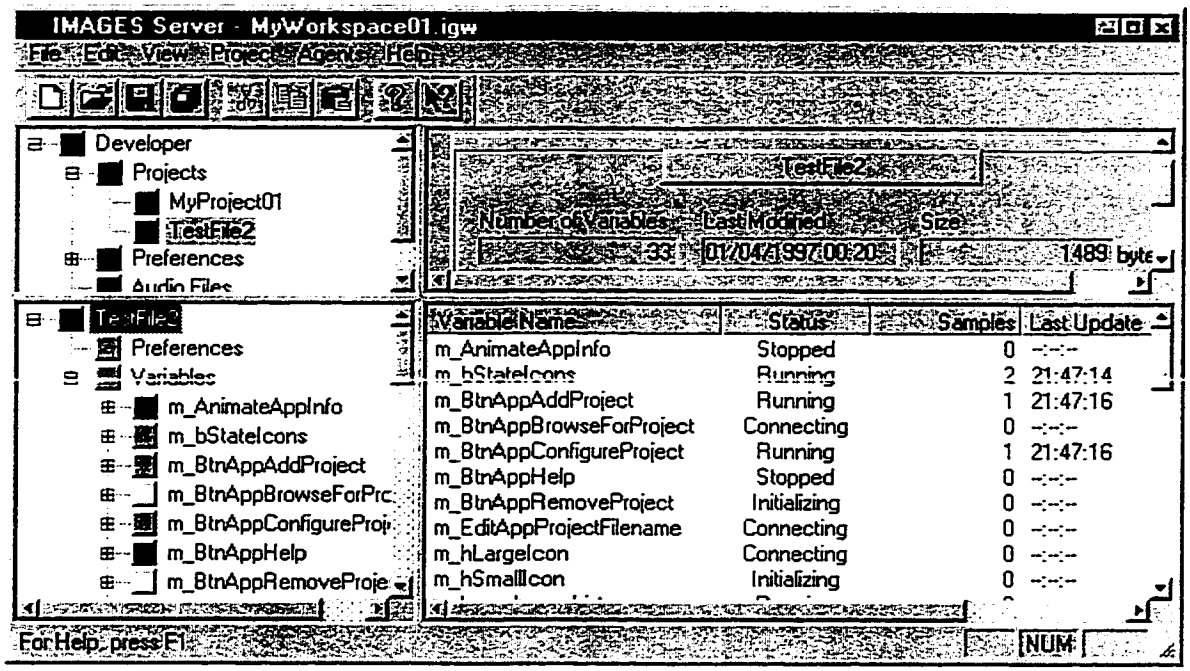


Figure B-17 The *IMAGES* Server main user interface window.

with the IGS tool, in that components are listed in the left pane(s), and their attributes are edited in the right pane(s) via component specific forms. The difference is that the server does not permit the modification of any attributes in a project. The server boasts monitoring features for inter-component specific information, such as gauge states and their connection status to their client application source. The information is presented in two forms. The first is a visual summary indicated by a color specific icon adjacent to each variable in the left hand pane (i.e. bottom-left hand pane in Figure B-17). The second is a set of detailed tables of attributes, such as the status of the communication between the client application and the gauge that represents a particular

variable, the number of updates issued and the last time an update occurred for each gauge, and the total time that a gauge has been connected to a client application. Some of this information is visible in the bottom right-hand pane of Figure B-17.

Appendix C: Integration Example With A Windows Application

The example presented in Section "5.5 An Integration Example", walked through a design process for integrating the *IMAGES* gauges with a simulation environment. Here, I present an example of integrating a native 32-bit Windows application with *IMAGES* gauges at the implementation level. Although this is a test tool application, the exact same process can be followed to integrate *IMAGES* with a simulation environment or any Windows based application. This example assumes a Visual C++ development environment; however, the implementation can be repeated for most Windows development environments such as Visual Basic and Delphi. The application that I am presenting is the test client used to validate the *IMAGES* system. To minimize bias, the application was implemented first, prior to any component of *IMAGES* having been designed. It supports all the basic data types that are currently supported by the Microsoft Visual C++ compiler, and the most common basic data types in Visual Basic and Delphi development environments. The application makes no presupposition of functionality implemented by the *IMAGES* system. It can issue gauge updates manually or continuously, using either user specified values or randomly generated values. The user interface, illustrated in Figure C-1, is terse, but sufficient to elucidate the implementation procedure.

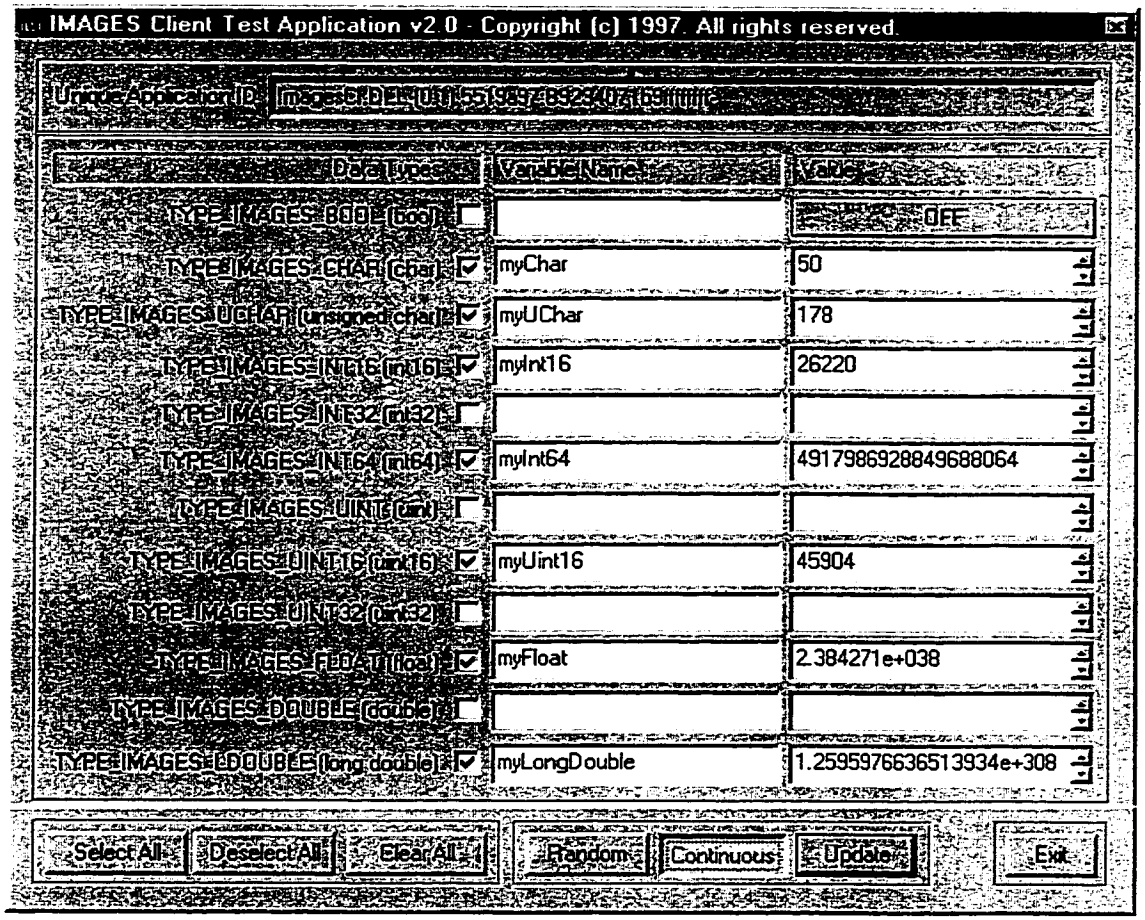


Figure C-1 Client Application User Interface.

There are three requirements and three steps to complete the implementation process. The first requirement is to place the `ImagesCI.h` header file (see Figure C-2) into a directory that is accessible by the compiler. The second is to ensure the `ImagesCI.lib` library file is accessible during linking. The third requirement is to ensure the ImagesCI.DLL library file is located in at least one of the following locations:

- Windows `system` (Win95) or `system32` (WinNT) directory (eg. `C:\WinNTW\System32\`);
- Windows root directory (eg. `C:\WinNTW\`);
- same directory as the executable client application (eg. `\MyClientApp\Release\`); or,
- any other directory in the *`PATH`* environment variable.

The first step toward integration is to add an #include statement, for the ImagesCI.h header file, into your root inclusion file. For example, the Visual C++ application wizard generates an

```
//*****************************************************************************
//
// ImagesCI.h
//
//===============================================
// DESCRIPTION
//
// IMAGES Client Interface Dynamic Link Library header file.
// This file must be included by an application as a prerequisite
// to enable integration with the IMAGES system at runtime.
//*****************************************************************************
#ifndef __IMAGES_CLIENT_INTERFACE_DYNAMIC_LINK_LIBRARY__H__INCLUDED__
#define __IMAGES_CLIENT_INTERFACE_DYNAMIC_LINK_LIBRARY__H__INCLUDED__

//===============================================
// IMAGES API Declarations
// These macros handle invocations within the client application source code.
//===============================================
#if defined( IMAGES )
#define ImagesGaugeUpdate(VariableName)      GaugeUpdate( #VariableName , VariableName )
#define ImagesInitialize(IgcFilename)        ClientInterfaceInitialize( IgcFilename )
#else
#define ImagesGaugeUpdate(VariableName)
#define ImagesInitialize()
#endif

//===============================================
// Storage variable types supported by a gauge.
//===============================================
typedef bool           TYPE_IMAGES_BOOL;      //                0 to 1
typedef char           TYPE_IMAGES_CHAR;      //              128 to 127
typedef unsigned char  TYPE_IMAGES_UCHAR;     //                0 to 255
typedef unsigned short TYPE_IMAGES_USHORT;    //                0 to 65,535
typedef long           TYPE_IMAGES_LONG;      // -2,147,483,648 to 2,147,483,647
typedef unsigned long  TYPE_IMAGES_ULONG;     //                0 to 4,294,967,295
typedef int            TYPE_IMAGES_INT;       // System dependent
typedef unsigned int   TYPE_IMAGES_UINT;      // System dependent
typedef __int16        TYPE_IMAGES_INT16;     //          -32,768 to 32,767
typedef __int32        TYPE_IMAGES_INT32;     // -2,147,483,648 to 2,147,483,647
typedef __int64        TYPE_IMAGES_INT64;     // -9,223,372,036,854,775,808 to
                                              //  9,223,372,036,854,775,807
typedef unsigned short TYPE_IMAGES_UINT16;    //                0 to 65,535
typedef unsigned long  TYPE_IMAGES_UINT32;    //                0 to 4,294,967,295
typedef float          TYPE_IMAGES_FLOAT;     //         3.4E-38 to 3.4E+38   ( 7 digits)
typedef double         TYPE_IMAGES_DOUBLE;    //        1.7E-308 to 1.7E+308  (15 digits)
typedef long double    TYPE_IMAGES_LONGDOUBLE; //       1.2E-4932 to 1.2E+4932 (19 digits)

//===============================================
// Storage type aliases
//===============================================
typedef __int64        TYPE_IMAGES_V_INT;
typedef long double    TYPE_IMAGES_V_REAL;
typedef bool           TYPE_IMAGES_V_BOOL;
```

Figure C-2 The ImagesCI.h header file.

```
//=================================================
// Declaration of the overloaded GaugeUpdate() method.  This method
// is invoked by the client application in order to update associated
// IMAGES gauges via the IMAGES Server.
//=================================================
#define  DECLARE_GAUGE_UPDATE(ImagesDataType) \
         bool FAR PASCAL EXPORT GaugeUpdate( const char*fp_lpszVariableName,
                                             ImagesDataTypefp_GaugeValue )

bool FAR PASCAL EXPORT ClientInterfaceInitialize();

DECLARE_GAUGE_UPDATE( TYPE_IMAGES_BOOL );          // IAG_NPBD_N
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_CHAR );          // IAG_NPBDD
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_UCHAR );         // IAG_NPBDE
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_INT16 );         // IAG_NPBDF
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_INT32 );         // IAG_NPBDH
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_INT64 );         // IAG_NPBD_J
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_UINT );          // IAG_NPBDI
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_UINT16 );        // IAG_NPBDG
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_UINT32 );        // IAG_NPBDK
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_FLOAT );         // IAG_NPBDM
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_DOUBLE );        // IAG_NPBDN
DECLARE_GAUGE_UPDATE( TYPE_IMAGES_LONGDOUBLE );    // IAG_NPBDO

#endif // !defined(__IMAGES_CLIENT_INTERFACE_DYNAMIC_LINK_LIBRARY__H__INCLUDED__)
```

Figure C-2 (cont'd) The ImagesCI.h header file.

StdAfx.h file which contains MFC header files that are not modified. This header file would be a good place to add the `#include <ImagesCI.h>` statement (see Figure C-3).

The second step is to add the `ImagesInitialize()` statement in the initialization thread of the application, adhering to the rules detailed in Table 5-2. At run-time, when the `ImagesInitialize()` statement is executed, the gauges are created based on the specified IGC filename. In the test client application source code, shown in Figure C-4, and Figure C-5, the `ImagesGaugeUpdate()` statements are placed in a message handler which is invoked whenever a variable to be measured, observed, and monitored is modified.

In Figure C-5, the ImagesGaugeUpdate() statements are used to handle user updates to the gauges made either manually or programmatically whenever either a timer expires or the next update has completed and a new message is posted. This results in a continuous invocation of the OnGaugeUpdate() message handler. The latter two options raises the issue of flow control, which

```
////////////////////////////////////////////////////////////////////////////
//
// StdAfx.h
//
// Include file for standard system include files, or project specific include files that are
// used frequently, but are changed infrequently
//
////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_STDAFX_H__8F6B31C7_05DC_11D1_83B7_0020AFA326C9__INCLUDED_)
#define AFX_STDAFX_H__8F6B31C7_05DC_11D1_83B7_0020AFA326C9__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN            // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>             // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <limits.h>             // System data type limits (eg. INT_MAX)
#include <float.h>              // System data type limits (eg. DBL_MAX)


#define INT64_MAX9223372036854775807
#define INT64_MIN -INT64_MAX - 1

__int64 __cdecl atoint64( CString pszValue );

#include "..\ClientInterface\ImagesCI.h"
#include "..\ClientInterface\ImagesCIdebug.h"

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_STDAFX_H__8F6B31C7_05DC_11D1_83B7_0020AFA326C9__INCLUDED_
```

Figure C-3 Example file to include the ImagesCI.h header (highlighted).

this test client application does not handle gracefully. Ordinarily a client application is expected to wait for an acknowledgment of the previous update for a particular variable before generating the next update.

The third and final step is to compile and link the application with the ImagesCI.lib library. There

```
//*****************************************************************************
//
// ImagesTestClientDlg.h : header file
//
//*****************************************************************************
#if !defined(AFX_IMAGESTESTCLIENTDLG_H__8F6B31C5_05DC_11D1_83B7_0020AFA326C9__INCLUDED_)
#define AFX_IMAGESTESTCLIENTDLG_H__8F6B31C5_05DC_11D1_83B7_0020AFA326C9__INCLUDED_

        :
#include <ImagesDataTypes.h>
#include "RandomNumberGenerator.h"
        :
#define IMAGES_VAR_TYPE_INDEX_INT32    2
#define IMAGES_VAR_TYPE_INDEX_DOUBLE 6
        :

/////////////////////////////////////////////////////////////////////////////
// CImagesTestClientDlg dialog
class CImagesTestClientDlg : public CDialog
{
        :
// Dialog Data
        :
        BOOL      m_bChkInt32;          // Flag to generate Int32 data
        BOOL      m_bChkDouble;         // Flag to generate Double data
        :
        CButton   m_ChkInt32;           // Checkbox for Int32 data type
        CButton   m_ChkDouble;          // Checkbox for Double data type
        :
        CString   m_lpszEditInt32;      // Int32 data value
        CString   m_lpszEditDouble;     // Double data value
        :
        CString   m_lpszEditVarInt32;   // Int32 variable name
        CString   m_lpszEditVarDouble;  // Double variable name
        :

// Implementation
protected:
        // Generated message map functions
        //{{AFX_MSG(CImagesTestClientDlg)
        :
        virtual void OnGaugeUpdate(); // This method is invoked whenever the "Update" button is pressed or
                                      // we generate a WM_GAUGE_UPDATE message for continuous updates.
        :
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
        :

// ATTRIBUTES
private:
        :
        TYPE_IMAGES_INT32     m_nValueInt32;
        TYPE_IMAGES_DOUBLE    m_fValueDouble;
        :
};

#endif // !defined(AFX_IMAGESTESTCLIENTDLG_H__8F6B31C5_05DC_11D1_83B7_0020AFA326C9__INCLUDED_)
```

Figure C-4 Example uses of the ImagesGaugeUpdate() statements.

```
//*****************************************************************************
//
// ImagesTestClientDlg.cpp : implementation file
//
//*****************************************************************************

#include "StdAfx.h"
        :

        :
BEGIN_MESSAGE_MAP(CImagesTestClientDlg, CDialog)
        //{{AFX_MSG_MAP(CImagesTestClientDlg)
        :
        ON_BN_CLICKED(IDC_BTN_RANDOM, OnBtnRandom)
        ON_BN_CLICKED(ID_GAUGE_UPDATE, OnGaugeUpdate)
        :
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

        :
        :

//=================================================
//
// MSG_HANDLER(WM_COMMAND, ID_GAUGE_UPDATE)
//
// This method is invoked when the "Update" button is pressed or the WM_COMMAND
// with an ID of ID_GAUGE_UPDATE is sent or posted to the message dispatcher queue.
//
void CImagesTestClientDlg::OnGaugeUpdate()
{
        UpdateData( TRUE );     // Get the values from the edit controls
        ConvertStringToValue_v();

        // Update the IMAGES gauges. If the target was not built with IMAGES,
        // this block of code does not generate any function calls to the IMAGES
        // Client-Side Application Interface.
        :
        if (m_bChkInt32)        { ImagesGaugeUpdate( m_nValueInt32 ); }
        if (m_bChkDouble)       { ImagesGaugeUpdate( m_fValueDouble ); }
        :

        // Kick the next interval to update the gauges.
        if (m_bChkFrequency) {  // Continuous mode - generate new values each iteration
                PostMessage( WM_COMMAND, BN_CLICKED, IDC_BTN_RANDOM );
                PostMessage( WM_COMMAND, BN_CLICKED, ID_GAUGE_UPDATE );
        }
}

        :
        :
```

Figure C-5 Example use of the ImagesGaugeUpdate() statements .

are no special settings requirements for the build environment to compile and link with the *IMAGES* Client Interface header and library files. However, a preprocessor flag (see Table 5-1), is required to enable or disable *IMAGES* for the simulation program or application.

This example demonstrates both the simplicity and the flexibility that is harnessed by using the *IMAGES* system for measuring, observing and monitoring the behavior of a simulation model or application program.

Appendix D: Development Requirements

The following list comprises the minimum requirements for integrating the *IMAGES* client-side application interface with a client application:

- Windows 95 or Windows NT (Intel platform) v4.0 or later operating system
- OLE v2.0 (or later) compliant services
- 6Mb (-50%/+100%) free hard disk space
- Windows 32-bit programming language (eg. Delphi, Visual Basic, Visual C++, etc.) with support for DLL run-time linking
- 32-bit compiler & linker for Windows 95 or Windows NT (Intel targets)
- Win32 API v4.0 (or later)
- 24Mb RAM (32Mb recommended).

The following optional requirement would exploit the maximum capabilities of *IMAGES*:

- a network of two or more Windows NT v4.0[1] based computers supporting distributed COM for remote OLE automation.

---

1. As of 4Q97, current commercially available versions of Windows 95 do not support distributed COM (DCOM), which is required to support OLE-automation over a network.

Appendix E: Glossary of Terms

Term	Description
API	Application Programming Interface
Atomic Object	An object or class definition that serves a single purpose to perform a single task. For example, an integer class could be used to perform operations on integers. This is in contrast to a complex object.
Common Object Request Broker Architecture	An architecture developed by the Object Management Group (OMG) to standardize distributed inter-application communication.
Complex Object	An object or class definition which may be composed of multiple atomic objects, with a purpose of performing multiple tasks. For example, a linked list object can perform many tasks, such as dynamically sizing the list, handling multiple data types, and inserting, removing and searching entries. This is in contrast to an atomic object.
CORBA	See Common Object Request Broker Architecture.
Gauge	An instrument used for measuring and observing a quantitative or qualitative value.

Term	Description
Gauge Scope	The scope of a gauge is defined by the location of either the ImagesInitialize() or ImagesGaugeCreate() API function and the ImagesGaugeUpdate() API function. In the case of the ImagesInitialize() function, the scope of a gauge is always global, so that all variables associated with a gauge must be unique. Gauges created by the ImagesGaugeCreate() function have a scope equivalent to a variable which is declared at the same location. In this case, the gauge may be locally unique.
GestCell	A simulation environment. See also M-Gest.
IGC	See *IMAGES* Gauge Configuration File
IGS	See *IMAGES* Gauge Specification Tool
IMAGES	See Intelligent Multi-Agent Based Virtual Gauges
IMAGES Gauge Configuration File	This type of file contains the configuration information for all the gauges that are defined prior to running the instrumented application. It is automatically generated when a project is created and managed by the IGS Tool, but can also be created manually. The file extension is IGC. For the format of this file type refer to the IGC template file (IMAGES/Templates/IGC.template). See also *IMAGES* Gauge Specification Tool.
IMAGES Gauge Specification Tool	A tool developed as part of the *IMAGES* system to assist a user in configuring and associating gauges with project variables.
IMAGES Server	The run-time application that manages and coordinates the communication among all supervisor agents as well as acts as the proxy between the client-applications and the gauges.
Intelligent Multi-Agent Based Virtual Gauges	A software solution for distributed intelligent monitoring of simulation model behavior generation and software programs in general.
Object Linking and Embedding	A Microsoft Common Object Model (COM) based interface for building distributed compound documents.
OCX	See OLE Custom Control Extension

Term	Description
OLE	See Object Linking and Embedding
OLE Custom Control Extension	OLE automation controls that are based on the Microsoft Common Object Model (COM) specification for inter-application communication.
Scope of a gauge	See Gauge Scope
Sockets	Sockets are a session layer communication facility for establishing logical connections between two end-points in a TCP based network.
TCP	See Transport Control Protocol
Transport Control Protocol	A protocol that operates at the transport layer of a communication network. It carries data, presented by the session layer, across the network. The network layer protocol that is predominantly used in today's network is IP (i.e. TCP/IP.) IP routes data across a network between two end-points. IP stands for internet protocol. It is the prime routing protocol used on the global internet (for world wide web traffic).
Variable list file	A client project specific file containing a list of variables that are to be instrumented with *IMAGES* gauges at run-time. This file is only used by the IGS tool in assisting the user perform gauge association and configuration. See Chapter 5 for a detailed description of the file format. This file is distinguished with a filename extension of VAR.
Winsock	The Microsoft Windows version of sockets. See also Sockets