The Mixed Reality Simulation Platform (MIXR)

Douglas D. Hodson[1], **David P. Gehl**[2]
Air Force Institute of Technology, WPAFB, OH, USA[1]
emails: doug@sidechannel.net[1], gehldp@earthlink.net[2]

Abstract— *The Mixed Reality Simulation Platform (MIXR) is an open-source software project designed to support the development of robust, scalable, virtual and constructive, and stand-alone and distributed simulation applications. Its most common use case is to support the development of executable simulation applications used to assemble real-time, interactive, distributed virtual environments (DVEs).*

MIXR's core infrastructure is architected to favor the development of applications that can be executed in a deterministic manner to meet real-time interaction/response time requirements, yet provide a degree of configuration flexibility for scenario definition, I/O, and standard interoperability interfaces (e.g., DIS, HLA, etc.).

Within the military, MIXR has been used as a basis develop applications that include mixtures of Live, Virtual, and Constructive (LVC) entities; more generally known as "mixed reality" simulations [1].

This paper describes the modeling and simulation software platform itself, and how it partitions code to take advantage of multi-core/multi-CPU PCs to support the development of these type applications.

Keywords: MIXR, DVE, LVC, mixed reality, simulation

1. Introduction

The Mixed Reality Simulation Platform (MIXR) is an open-source software project that originated in the Department of Defense (DoD). It has been used to support the development of numerous distributed applications in multiple domains, including Live, Virtual, and Constructive (LVC) simulations.

MIXR defines a high-level organizational pattern that provides a structure for simulation applications (sometimes called *simulators*). In essence, the software provides a blueprint for the developer to customize in order to ease the creation of simulation applications. It leverages traditional object-oriented (OO) software design principles while blending real-time system concepts so that human and/or hardware interaction requirements can be met.

By providing abstract representations of system components (i.e., abstract classes), models at different levels of fidelity can be intermixed in such a way as to optimize run-time performance. For virtual and mixed reality simulations, the abstract representations of systems enable a developer to tune applications to run efficiently so that human and/or hardware (in-the-loop) interaction latency deadlines are satisfied. For purely constructive simulation applications (where interaction deadlines are non-existent or of less concern), more detailed, possibly processor intensive, system models can be selected.

The software leverages the Model-View-Controller (MVC) pattern by partitioning functional components into separate packages. MVC concepts are taken a step further in this domain by providing *views*, such an abstract network interface to support specific interoperability solutions; examples include the Distributed Interactive Simulation (DIS) protocol and the High Level Architecture (HLA).

Specific applications using the software are numerous and include current and future fighter and bomber platform simulators, Unmanned Aerial Vehicle (UAV) Ground Control Stations, Integrated Air Defense Systems (IADS), futuristic battle managers and more; for additional information see [2] and [3].

2. Abbreviated History

The genesis of the MIXR software code base can be traced to the late 1980s when it was written in the C programming language and executed on a Commodore Amiga 1000 (yes, a Commodore Amiga!). Because C doesn't directly support the OO programming paradigm, the code base defined an OO-like infrastructure to support programming from this perspective. In the early 1990's, the C-based OO system was converted to C++, where applications were developed and executed on Silicon Graphics (i.e., SGI) workstations. The transition away from Silicon Graphics workstations to personal computers (PCs) occurred in 1997.

Initially, the code base had no official name associated with it; that changed in 2002 when it was named the Enhanced Air-to-Air, Air-to-Ground Linked Environment Simulation (EAAGLES); later updated to mean the Extensible Architecture for the Analysis and Generation of Linked Simulations. The update removed domain-specific terminology such as "air-to-air" and "air-to-ground" to emphasize the more general purpose modeling and simulation capabilities the software is designed to support.

In 2003, "EAAGLES" became more visible within the DoD community when an EAAGLES-based fighter simulator was shown and used at the Interservice/ Industry Training, Simulation and Education Conference (I/ITSEC); the world's largest modeling, simulation, and training conference. Even though the generic fighter simulator was developed in only a few months, it performed flawlessly; a testament to both the design of the simulator and the underlying software framework.

In July of 2006, a significant subset of the original EAAGLES code was released into the public domain; it became what is known as OPENEAAGLES. At the same time, a website was set up to provide information, documentation, and releases. In 2009, the book "Design & Implementation of Virtual and Constructive Simulations Using OPENEAAGLES" was published [4]. Since then, a steady stream of releases has been posted.

In 2017, OPENEAAGLES was renamed to MIXR for a number of reasons. These include:

- providing a better alignment of the project name with the domain of interest; i.e., the development of mixed reality simulation applications. See [1] and [5] to understand the relationship between mixed reality simulations and LVC simulations,
- explicitly shifting away from using the term “framework” to “platform” to highlight efforts to address the following issues: from a software perspective, exposing MIXR’s functionality and capability to better support alternative programming paradigms (as opposed to traditional OO-style inheritance through subclassing) and secondly, from a modeling perspective, MIXR’s inherent capability to address higher-level distributed simulation “platform-wide” interoperability concerns as described by the conceptual interoperability model [6], [7],
- serving as an indication that the project is transitioning away from a “traditional C++” code base in favor of a “modern C++” code base to improve quality, capability, and understandability, and to reduce complexity.

Today, the MIXR code base is quite stable in terms of the features it provides and capabilities it exposes. As a result of its relatively long development history, it has accumulated lots of “baked-in” knowledge which resulted from its use in supporting the development of a large number of simulation applications. The current focus is on refining and improving it in ways to make it even more flexible and accessible to a wider audience.

3. Terminology

Over the past few decades, a number of equivalent terms have arisen to describe real-time, distributed simulations that create a shared *virtual* world in which humans and/or hardware interact. These include, *distributed virtual environments* (DVEs), *networked virtual environments* (NVEs) and *distributed virtual simulations* (DVS). Today’s common practice is to characterize these systems more generally as *"mixed reality"* environments or simulations.

A common architectural characteristic of these systems is the asynchronous execution of several largely autonomous standalone simulation applications that interact with each other by exchanging data through a network to create a shared illusion of a virtual world (in military terms, a “synthetic battlespace”). In a military domain, the means to exchange data between applications is defined by interoperability protocols, of which several are published as IEEE standards; for example, the Distributed Interactive Simulation (DIS) and the High-Level Architecture (HLA).

Because humans and/or hardware are included as part of the represented system of interest (i.e., “in-the-loop”), additional requirements in the form of interactive response times arise, which classify the entire apparatus as a real-time system. Trade-offs must be made concerning responsiveness and fidelity when constructing real-time systems. The MIXR code base was designed from scratch considering these requirements, so that trade-offs between deterministic execution performance and model fidelity can be made.

4. Software Classification

A framework is a set of cooperating classes that make up a reusable design for a specific domain [8], [9]. A framework is customized to a particular application by creating application-specific subclasses of abstract classes from the framework [10]. A toolkit is a set of related and reusable classes that provide useful, general-purpose functionality. They are the OO equivalent of subroutine libraries [10].

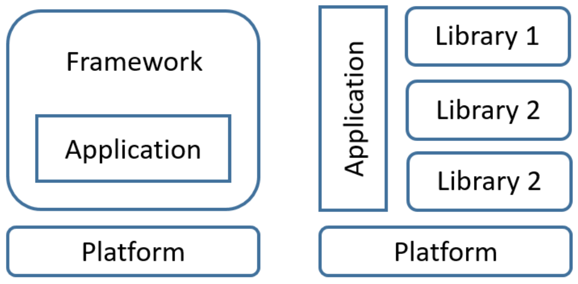

Fig. 1: Software Organization

As Figure 1 shows, an application built using a framework is created by extension; which, in C++, usually means subclassing classes via inheritance. Whereas, an application that simply uses library functionality (i.e., toolkits) does not exhibit the same kinds of dependencies (i.e., specifically coupling).

While MIXR is strongly organized as a framework in which applications are developed by extending classes, the goal is to position it to feel and act more like a general purpose modeling and simulation platform. In other words, expose more of its functionality without having to subclass classes.

In either case, MIXR is not an executable application like, say, Microsoft Word or even a typical game engine. It doesn’t define a `main()` function; it leaves those details up to the developer which is in full control of how applications are assembled and configured.

MIXR is written in C++ and partitioned into packages that serve as functional toolkits (i.e., libraries) for the developer to use as needed. For example, it defines a graphics toolkit which can be used to facilitate the development of operator/vehicle interfaces and displays.

As organized, the software enables the creation of a diverse set of simulation applications. Derived simulation applications can often be run stand-alone or within a distributed networked environment. Distributed applications that interact with each other using standard protocols such as DIS and/or HLA can be set up almost “out of the box”.

5. Virtual Simulation Requirements

Simulations that interact with human participants (or hardware) must respond within prescribed deadlines (latency or response times). A simulation that does not respond like the system it is intended to represent will frustrate the operator and may skew the simulation results. Software systems faced with this requirement fall into the category of a real-time system. Real-time systems are designed and organized so that time-critical (often periodic) tasks can meet their deadlines.

Two standard approaches to scheduling tasks are priority-based and foreground/background systems. Priority-based designs assign a priority to each task in the system. The task with the highest priority that is ready to run is executed first. The scheduling of the task resides with the operating system.

In a foreground/background system, the application controls the scheduling of tasks. Foreground tasks are executed with the help of a jump-list, or a managed list of pointers to functions (tasks). Tasks are executed one after another as defined by the list order. Aperiodic events and background tasks receive processing time after all the “highest priority” tasks in the list have finished.

MIXR is organized as a modified foreground/background system. Instead of managing a jump-list (or a list of functions to process), thread execution paths are interwoven into the design of the class hierarchy. It is specifically designed to take advantage of multi-core/multi-CPU PCs which allow the creation of high priority foreground threads. Because multiple processors are available, reliable execution of high priority time-critical foreground threads is assured by general-purpose operating systems such as Windows and Linux.

It should be emphasized that MIXR-based applications execute code as a cycle (or frame-based) simulation; not as a discrete-event simulation. This approach satisfies the requirements for which MIXR is designed; namely, support for models of varying levels of fidelity including detailed physics-based and digital signal processing models.

6. Real-Time Simulation Platform

MIXR was written in C++ because:

- even though most real-time systems are developed in C for performance reasons [11] and object-oriented languages tend to be viewed with skepticism, we believe the flexibility C++ provides outweighs any potential performance issues,
- C++ is portable and compilers exist on virtually every platform. This allows developers to build MIXR-based applications on any of the major popular operating systems (Windows, Linux, etc),
- C++ is flexible in terms of supporting multiple programing paradigms,
- it is desirable to define memory management so it does not interfere with the overall performance of the application. Therefore, the use of `new` and `delete` operators is preferable to uncontrolled garbage collection.

While it is beyond the scope of this paper to discuss every class defined within the MIXR code base, a few classes deserve attention to provide insight into the overall software architecture.

Object : The *Object* class serves as the C++ “system” object for the MIXR code base. Unlike other OO languages (for example Java or Ruby), C++ does not provide a system object. C++ also does not provide native garbage collection. While lacking these two features could be viewed as a negative when comparing the native features of various languages, being able to design this functionality can be an important advantage for a domain that consists of executable applications that must meet real-time requirements. For example, if the developer is writing an application in which control over potentially time-consuming memory management operations is of little concern, the code base provides smart pointers to automatically manage the creation and deletion of objects. If, on the other hand, the application has time constraints to meet (i.e. a real-time system), the uncontrolled creation and destruction of objects might lead to performance problems.

One of *Object*’s capabilities is to provide a simple reference counting system for the memory management of all objects. *Object* provides access to this system so that a developer can manually control and tune performance-oriented applications if issues arise in, for example, the real-time processing of modeled radio frequency (RF) emission packets or infrared radiation (IR) geometry information.

The other subtle but important aspect of providing a system object appears in the form of type checking. The presence of a system object, and the derivation of all classes from it, enables the dynamic casting of objects. It also avoids the pitfalls associated with untyped functions and classes. MIXR’s coding standards explicitly prohibit the use of void pointers for this very reason.

Component : In OO programming, a container class is a class of objects that contain other objects. The MIXR component class is that and much more. *Component* is a container for other components. *Component* also defines a basic messaging system that is used throughout the code base.

From the outset, MIXR was designed to facilitate the creation of simulation applications that execute in real-time and/or interact with a human participant. Applications with time constraints and latency/response deadlines typically identify and separate time-critical tasks from non-time-critical tasks; for example, separating the execution of an aerodynamic model at a specific frequency from writing data to a hard disk, or printing a document.

In MIXR, this separation is facilitated by two methods in the component class. Code that needs to execute in a time-critical manner (usually mathematical calculations) is placed in an overridden virtual `updateTC()` (update time-critical) method. Code that can be run in a non-time-critical manner is placed in the overridden virtual `updateData()` method.

This organization of code has a number of advantages:

- since the time-critical code is clearly separated from background code, applications can be designed to meet performance requirements,
- all the code (time-critical and background) associated with a model is logically located within the same class.

As in Figure 2, one can represent an instance of a simulation application as nothing more than a tree of *Component*s. A call to the top (or root) of the tree's `updateTC()` method, will automatically execute every subcomponent's `updateTC()`. In other words, every component will execute the code of its children. This process continues until the entire tree has been processed. The same process takes place for the background code.

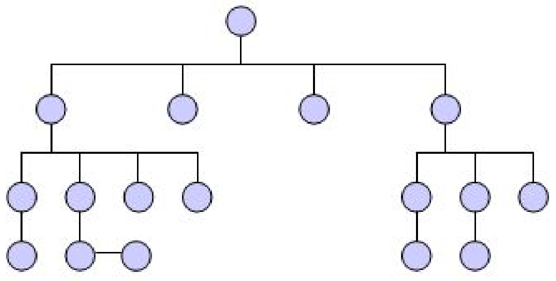
Fig. 2: Component Tree

The MIXR coding standard spells out rules to follow when writing code in `updateTC()` (e.g., no blocking I/O calls). These rules mimic understood guidelines associated with real-time system development.

7. Application Structure

A developer using MIXR as a basis for a simulation typically builds an application by either using existing classes (i.e., models) or extending them to add detail. Finally, the developer must define `main()` for the application.

The user-defined `main()` usually performs these tasks:

- reads a configuration file that defines and creates an object hierarchy. MIXR provides a parser to do this; it defines a simple context-free scheme-like input language, that is easy to extend,
- creates and sets up threads to be executed. For non-real-time applications (e.g., purely constructive applications) a single thread maybe all that is needed. For a virtual simulation with time-critical code, both foreground and background threads are usually created,
- executes the simulation by calling `updateTC()` and `updateData()` as required. If the application is a virtual simulation, high-priority thread(s) are assigned to process foreground tasks.

Because the developer fully defines `main()`, a variety of execution scenarios can be addressed. In practice, the user-defined `main()` tends to be short; most of the application code consists of new or extended classes.

Station : MIXR applications are typically structured as shown in Figure 3. Thinking in terms of a tree of components, the class *Station* resides at the top, or the root node. Every other component is a subcomponent of *Station*. *Station* connects models to views (e.g., graphical displays, I/O devices and controls, interoperability networks).

Simulation : A *Station* owns an instance of a *Simulation* which manages a list of players (i.e., entities or platforms) and keeps track of simulation time, which includes the cycle, frame, and phase that is currently being processed.

MIXR is a frame-based system, as such, delta time is passed as an argument to `updateTC()` so proper calculations involving time can be performed. Having models rely on delta time for calculation means the frequency of the entire system can change without having to change each and every model. Additional time-related information is recorded in terms of cycles (16 frames or sometimes called a major frame) and phases. Phases sequence the flow of data throughout a model. Four phases are defined:

- Dynamics – Player or system dynamics including aerodynamic, propulsion, and sensor positions (e.g., antennas, IR seekers) are updated.
- Transmit – R/F emissions, which may contain datalink messages, are sent during this phase. The parameters for the R/F range equation, which include transmitter power, antenna pattern, gains, and losses, are computed.
- Receive – Incoming emissions are processed and filtered, and the detection reports or datalink messages are queued for processing.
- Process – Datalink messages, sensor detection reports,

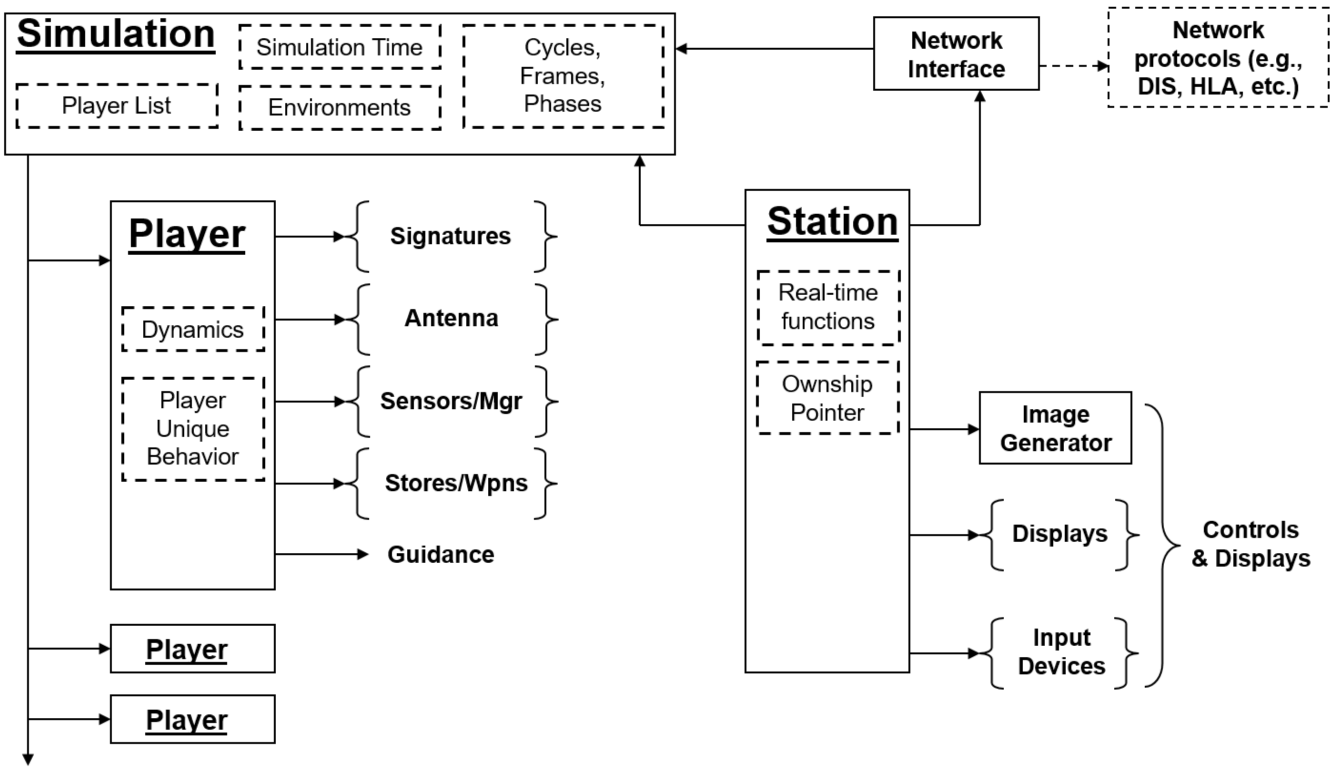

Fig. 3: Application Structure

and tracks are processed. State machines, onboard computers, shoot lists, guidance computers, autopilots or any other player or system decision logic are updated.

Player : A *Player* is a component that adds dynamics and other unique behaviors. Some components that can be attached to *Player* include signatures, antennas, sensors, and stores. Derived air and ground players are included within the code base.

Network Interface : An abstract interoperability network interface is defined so specific protocols can be incorporated, such as *DIS* to support interoperability with other distributed simulation applications. This network interface automatically creates new players in the player list. As far as the simulation is concerned, these players are like any other.

8. Graphics

The platform defines several graphics toolkits for the development of operator/vehicle interface displays. The graphics toolkits are based on OpenGL for all primitive drawing, thus making it compatible with virtually any platform.

The foundation for graphics drawing is contained in the *graphics* package. It contains classes for drawing graphics objects such as bitmaps, input/output fields, fonts, polygons, readouts, textures, and others.

As Figure 4 shows, the graphics architecture defines hierarchical relationships between the *Graphic*, *Page* and *Display* classes.

The *Graphic* class encapsulates attributes associated with a graphic such as color, line width, flash rate (for graphics that flash), coordinate transformations, vertices and texture coordinates, select names and scissor box information. Since *Graphic* is a component, it can contain other graphics.

The *Page* class defines a “page” of graphics that can facilitate the creation of Multi-Function Displays (MFD) where specific page transition events need to be defined. The *Display* class defines all the resources available for drawing such as fonts, the color table and both the physical and logical dimensions of the display viewport. Finally, open source GUI toolkits (such as Glut, Fox, FLTK, and Qt) can be used to provide a rich interface.

MIXR graphic classes ease the development of operator/vehicle displays and leverage open source GUI toolkits, but they are not intended to replace visual scenegraph oriented displays (such as a heads-up display or an image generator). The overarching philosophy of MIXR is to avoid reinventing what is already well done by other applications and/or packages.

Higher level MIXR toolkits that use this structure include the instrument library which includes dials, buttons, gauges, meters, pointers, and countless other fully functional instruments, along with simple maps.

All of the graphical toolkits are independent of the simulation modeling environment. Models don't include any special knowledge of graphics, and graphics include no special knowledge of models. The code that connects the two resides within the *Station* class.

Through an *ownship pointer* in the *Station* class, the controls and displays of any player can be switched at any time. Switching from player to player is useful for observing simulation interactions from different perspectives.

All of the graphics classes are derived from *Graphic* which is derived from *Component*. Being a component, all time-critical code can be written into the `updateTC()` method and background processing can be written into the `updateData()` method. Sometimes, in real-time system development, it is desirable to set graphics drawing to an even lower priority than other background processing. Therefore, another method within the *Graphic* class is defined that serves as a placeholder to do actual OpenGL graphics drawing.

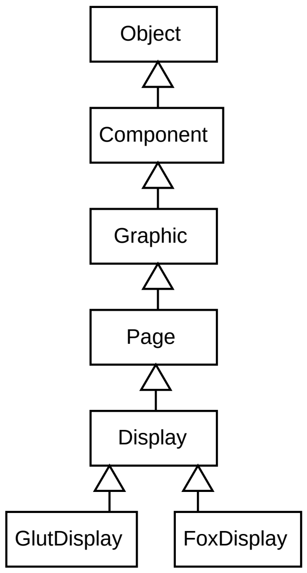

Fig. 4: Graphics Class Hierarchy

A sample application included in with the MIXR distribution illustrates basic graphics by drawing a *worm* that moves around the screen and *bounces* off the walls. Code for this example is organized as follows. All mathematical calculations for the position, speed, and direction of the worm are performed in `updateTC()`. All the work to set up what to draw is done in `updateData()`. The actual drawing of the graphic is performed by *Graphic*'s `draw()` method.

Organizing code this way enables the application developer to determine how to execute the code and to define threads to meet response time requirements. For the example just presented, a thread is set up to execute time-critical mathematical calculations associated with *worm* movement in real-time, and in a non-time-critical manner the operating system (or Glut in this case) draws the worm during idle times.

9. Device I/O & Linkage

MIXR abstracts I/O devices and complete linkage systems so that a hardware interface appears to the application as a single (unified) device that presents a number of analog (axis) and digital (button) values as shown in Figure 5. This linkage package has interface code for several I/O devices including joysticks and USB-based devices. It has also been extended to interact with other commercial I/O cards and data acquisition devices.

Once the device is initialized, a call to the virtual `receive()` method, defined in the *IODevice* class, obtains the latest values from the device. Information about button transitions can be determined as well as setting deadbands on analog inputs.

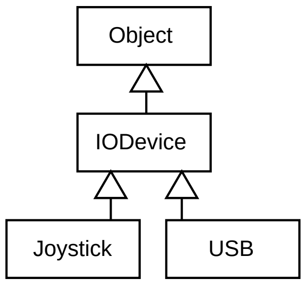

Fig. 5: Device Class Hierarchy

The *Station* class defines how axes and buttons are *connected* to the models and views of the simulation application.

10. Example Applications

A MIXR software distribution consists of both code to build new applications and examples that demonstrate how to use some of its inherent functional capabilities. For an extended description of some early military products, see [2]. It details an early version of a fighter cockpit, a Ground Control Station and a Group Command Post.

11. Opportunities

For almost four decades, the MIXR code base has improved and evolved in terms of both structure and provided functionality. From its earliest beginning when it was written in C with a developer created OO-like system to mainstream C++; from a Commodore Amiga 1000 to current generation PCs, it has proven itself to be a useful and capable software development product in which a wide range of simulation applications have been created.

Given that, we don't consider MIXR "finished" - in fact, we believe with the advent of Modern C++ in 2011 (which was the first major language update in nearly 13 years), new opportunities for the code base are in store. For example, C++'s standard library is growing and providing cross-platform solutions for threading, high-resolution clocks and atomic operations of which MIXR needs to fulfill its requirements; all of these aspects can be leveraged to both reduce the size and complexity of the code base. Also, with adoption or mainstreaming of graphics processing units (GPUs) as included capabilities within a typical PC, other opportunities present themselves in terms of optimizing parallel data-centric operations.

Favored programming paradigms and idioms have also changed; *excessive* inheritance is discouraged for a number of reasons including 1) easy misuse, such as use for reuse (i.e., "implementation inheritance"), 2) to reduce (or loosen) tightly coupled code and 3) the possible refactoring of inheritance relationships to satisfy new use case requirements.

Because MIXR has existed for a considerable time, it could be argued that its inheritance structure is reasonably *stable* and well defined; although the desire to model and simulate different, potentially technologies might alter that view.

12. Finally

MIXR is open-source and freely available; it along with a set of examples can be downloaded from `www.mixr-platform.org`.

References

[1] D. D. Hodson, "Military simulation: A ubiquitous future," in *2017 Winter Simulation Conference (WSC)*, 2017, pp. 4024–4025.

[2] D. D. Hodson, D. P. Gehl, and R. O. Baldwin, "Building distributed simulations utilizing the EAAGLES framework," *Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*, 2006.

[3] "MIXR Platform website," http://www.mixr-platform.org, accessed: 2018-06-23.

[4] D. M. Rao, D. D. Hodson, M. S. Jr, C. B. Johnson, P. Kidambi, and S. Narayanan, *Design & Implementation of Virtual and Constructive Simulations Using* OpenEaagles. Linus Publications, 2009.

[5] D. D. Hodson and R. R. Hill, "The art and science of live, virtual, and constructive simulation for test and analysis," *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, vol. 11, no. 2, pp. 77–89, 2014.

[6] A. Tolk, *Engineering Principles of Combat Modeling and Distributed Simulation*, 1st ed. Wiley Publishing, 2012.

[7] A. Tolk and J. Muguira, "The levels of conceptual interoperability model," *Fall Simulation Interoperability Workshop*, 2003.

[8] P. L. Deutsch, "Design reuse and frameworks in the smalltalk-80 system," in *Software Reusability, Volume II: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis, Eds. Addison-Wesley, 1989, pp. 57–71.

[9] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, no. 2, pp. 22–35, 1988.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] P. A. Laplante, *Real-Time Systems Design and Analysis*. Wiley-Interscience, 2004.

Author Biographies

DOUGLAS D. HODSON is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications. He is also a DAGSI scholar and a member of Tau Beta Pi.

DAVID P. GEHL is employed by L3 Link Training & Simulation. He has over 45 years of experience in human-in-the-loop simulation and training for human factors engineering research including extensive knowledge in pilot/operator-vehicle interfaces, aircraft system models (aerodynamics, radars, weapon delivery, navigation, visual systems, etc.), and real-time system development. Previously, he served as the primary architect for the Extensible Architecture for the Analysis and Generation of Linked Simulations (EAAGLES) simulation framework. He received a B.S. in Computer Science in 1979 and a M.S. in Systems Engineering in 1986 from Wright State University.