

Martin R. Stytz, Ph.D.
Sheila B. Banks, Ph.D.
Larry J. Hutson
Eugene Santos, Jr., Ph.D.
Virtual Environments Laboratory
Artificial Intelligence Laboratory
Wright-Patterson AFB, OH 45433
mstytz@afit.af.mil
mstytz@acm.org
sbanks@afit.af.mil

An Architecture to Support Large Numbers of Computer-Generated Actors for Distributed Virtual Environments

Abstract

A variety of challenges exist in the design of systems that can be used to host a wide variety of computer-generated actors (CGAs) that possess believable behaviors. The challenges arise in the areas of system architecture and design, knowledge-base design, decision-making mechanisms, and the distributed virtual environment (DVE) network interface. These challenges are especially significant if the DVE is to be used for training, because accurate training is essential to the ready application of training experience to real-world situations. The project described in this paper was undertaken to improve the quality of threat CGAs in DVEs utilized for aircrew training. In this paper, we describe the system and the reasons for its genesis. We present the system requirements, system architecture, component-wise decomposition of the system design, and structure of the major components of the decision mechanism. We conclude with a summary of our results to date and recommendations for further research.

1 Introduction

The Joint Synthetic Battlespace (JSB) proposed within the Department of Defense Modeling and Simulation Master Plan (available at <http://www.dmsomil>) requires a distributed virtual environment (DVE) wide consistent threat environment to achieve a useful mission rehearsal, training, and test and evaluation capability. One of the obstacles to achieving large-scale, complex distributed virtual environments is the difficulty and expense involved in inserting large numbers of believable actors into the DVE. Human-controlled actors are costly in terms of both hardware and human time due to the large numbers of human operators required to control actors in the DVE at run-time. However, little relief to these problems has been forthcoming. To date, computer-generated actor (CGA) systems have proven to be expensive to implement, expensive and challenging to modify, and lacking in realistic behaviors. Because the Department of Defense is moving toward the use of Distributed Mission Training (DMT) for aircrew training, approaches to mitigate the costs associated with current CGA implementations are needed. A more-flexible CGA architecture designed for training needs, cost containment, and software modification is needed. For CGAs to be useful in training, they must exhibit a broad range of skills, display competency and realism in their behaviors, and comply with current doctrine. To minimize costs, a single computer host should insert a number of CGAs of various types into the environment and coordinate the activities

Report Documentation Page				Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE 1998		2. REPORT TYPE		3. DATES COVERED 00-00-1998 to 00-00-1998	
4. TITLE AND SUBTITLE An Architecture to Support Large Numbers of Computer-Generated Actors for Distributed Virtual Environments				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute Technology, Virtual Environments Laboratory, Wright Patterson AFB, OH, 45433				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT see report					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 29	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

of its own actors with the CGAs that are controlled by other computer hosts. Because of the rapid rate of change in DVE technologies and the ever-expanding set of performance objectives for any CGA, the system must be modifiable at reasonable cost. To address these needs, we undertook the development of a CGA application for distributed mission training threat systems. Because of our ready access to military DVEs, we chose to develop and refine our concepts within military DVEs that are geared toward Air Force aircrew DMT needs.

In a DVE that uses CGA systems to provide threats for aircrew training, each of the CGAs must exhibit realistic levels of fidelity to all the other actors operating in the DVE, and they must also interact with human-operated and computer-controlled actors in a realistic fashion. Achieving these goals using current systems for Air Force aircrew training in DVEs is not currently possible for two reasons. First, each manufacturer of primary aircraft simulator training systems has devised a simulator-specific threat system and made modeling decisions that generally support only a specific customer organization for a select few predetermined threats. This traditional threat simulation approach is expensive and leads to ongoing difficulties in maintaining threat currency as intelligence updates are made, new weapons are introduced, and new theaters of operation are identified. These simulator-specific systems tend to be brittle and unmaintainable and cannot be used in a military training DVE without substantial investment in software development. Second, the threat-system interaction on a distributed network must be coordinated, but the individualized nature of current threat systems precludes the possibility of introducing coordinated threats. Our work, the Distributed Mission Training Integrated Threat Environment (DMTITE) project, was undertaken to address these two issues.

The DMTITE project is identifying the requirements for a distributed-threat environment and building a demonstrator United States Department of Defense (DoD) High-Level Architecture (HLA) compatible system to provide realistic threats for aircrew training. The DMTITE system will be used within large-scale, HLA-based DVEs to insert a variety of accurate and highly realistic threats into the DVE for aircrew training. To be

a suitable system, DMTITE must provide a distributed-threat environment comprised of surface threats, air threats, and jamming systems. To achieve these objectives, DMTITE must have threat models and knowledge bases for interaction with every appropriate entity in the DVE. A key element of the system is the provision of realistic behaviors and multiple skill levels for the threat systems. To further improve the fidelity of the threat portrayal, we incorporate realistic sensor models, aerodynamics models, and weapons models into DMTITE for each threat system. Decision making can be accomplished using a variety of decision-making techniques such as fuzzy logic or case-based reasoning. Each DMTITE system must operate autonomously and also be able to cooperate with other DMTITE systems for the DVE to portray a coordinated threat environment. Figure 1 illustrates the anticipated way in which DMTITE will be used.

As shown in Figure 1, three locations (Eglin, Luke, and Tyndall Air Force Bases) are participating within this DVE. Each location has two dedicated DMTITE systems that are used to insert threats into the DVE. Four manned aircraft trainer systems for aircrew training are colocated with the DMTITE systems at each base. Two additional DMTITE systems in the DVE are responsible for inserting RADAR threats and RADAR jamming into the DVE. Each DMTITE can insert a variety of threat systems into the DVE, and the performance of each DMTITE system can be varied to portray a variety of operator skills and tactics.

To help us achieve the requirements mentioned above and to aid us in developing the prototype, we devised and refined a software architecture for DMTITE that naturally supports variety in performance for a given type of CGA and also allows us to organize and build vastly different CGAs within the same architecture. The architecture also had to help us uncover and refine system requirements as well. Because the requirements for DMTITE expanded over the course of the project and because CGA requirements in general are continuously changing, an evolutionary and exploratory approach to knowledge engineering, such as the Rapid Evolutionary and Exploratory Prototyping (REEP) methodology (Stytz, Adams, Garcia, Sheasby, & Zurita, 1997) is re-

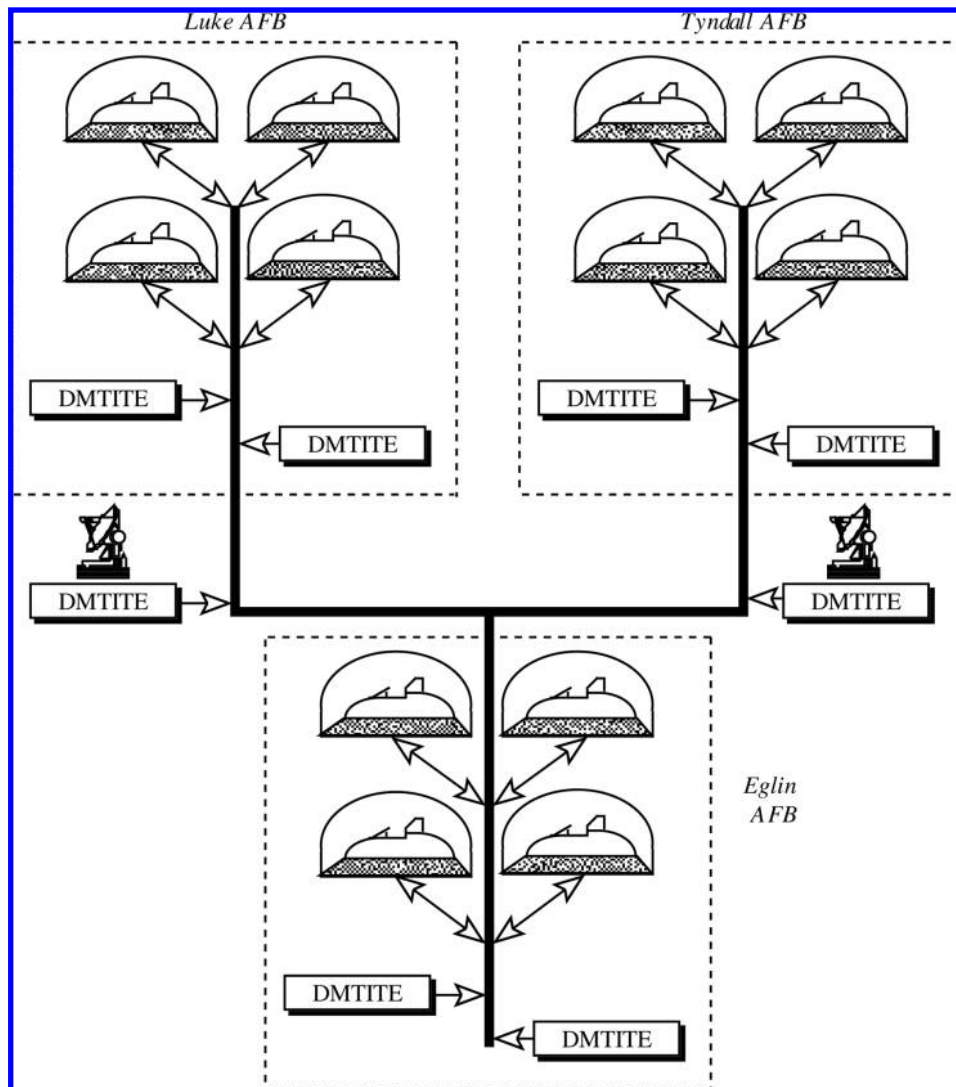


Figure 1. DMTITE operational concept for distributed training using distributed virtual environments.

quired. To support the REEP approach, our architecture consists of highly modular components in which interdependencies are well defined and minimized.

The next section presents a short discussion of background information for our project. Section 3 contains a description of the operational concept for the DMTITE system, its requirements, and the architectural implications of these requirements. Section 4 presents the simulation object model we developed for DMTITE. Section 5 contains a discussion of our use of containerization in DMTITE. Section 6 presents our architectural solution to the system requirements that have been levied against

DMTITE. Section 7 presents the system design. Section 8 contains a summary and presents some suggestions for additional work.

2 Background

This section discusses DVEs, CGAs, relevant projects, the DoD HLA, the Common Object DataBase (CODB) baseline for the DMTITE system architecture, and the REEP approach to system development. These topics form the groundwork for the DMTITE architec-

ture and its operational environment. Before turning to these topics, we will first define our terms. In a DVE, an *entity* is a component of a DVE whose state can change. For example, terrain whose appearance and features can be changed as a result of plowing, explosions, or traffic is an entity. Terrain that does not change is not an entity. An *actor* is an entity that moves with apparent intelligent purpose. Actors can be virtual (human controlled), constructive (traditional simulation controlled), live (derived from instrumented range data), or computer generated (controlled by a computer program, which may include artificial intelligence techniques). In a DVE, weather is not an actor, but a human-controlled aircraft avatar is an actor. A *host* is a computer system within a DVE that allows its human or computer user to control actors or entities within the DVE and / or to observe the actions of other virtual environment actors. *Human behavior modeling* is the process of making CGA behaviors realistic by developing models of the output of the human decision process. The most important aspects of developing the behavioral model for a CGA are *knowledge acquisition*, which is acquiring the information needed to effectively model human behavior within the DVE; *knowledge representation*, which is putting the knowledge base into a form that can be accessed readily and used for analysis and decision making during DVE operation; and building the *decision making apparatus* to perform decision making. Knowledge acquisition and knowledge representation jointly determine the information about the human mental models that is brought to bear on the CGA decision process.

Distributed Virtual Environments: The most widespread use of network technology for DVEs relies upon the current distributed interactive simulation (DIS) suite of standards (IEEE Standard 1278-1993) or upon the HLA. DIS was designed to link distributed, autonomous hosts into a real-time DVE via a network for data exchange. The data describe events and activities. DIS takes the concept of environmental distribution to its extreme; there is no central computer, event scheduler, clock, or conflict arbitration system. The HLA is a more comprehensive architectural approach: it describes the communication requirements, basic system require-

ments, and defines an object-oriented approach to defining a DVE. Stytz, Banks, and Santos (1996) present additional information concerning DIS and DVEs, as does Blau, Hughes, Moshell, and Lisle (1992) and Blau, Moshell, and McDonald (1993).

The High-level Architecture: Because of the difficulties encountered when attempting to reuse simulation software and to engineer participants for a DVE, the US Department of Defense has undertaken development of a High-Level Architecture (HLA) for DVE applications. A central goal of the HLA is to establish a framework that facilitates interoperability between simulations and models. A central architectural decision supporting this goal is the separation of application functions from communications functions. All application functions are managed by the host application software system while communications functions are managed by the run-time infrastructure (RTI). The RTI manages communication paths among executing applications and ensures that its application acquires the data that it needs and can make available to the DVE. The data that an actor needs from other actors in the DVE is identified by subscribing to the actors that can provide the data. The data that an actor provides to participants is published to the DVE, and actors can subscribe only to data that has been published. The RTI publish-and-subscribe mechanism minimizes the amount of data transmitted between applications to only the data that is required by the applications. The foundational papers for HLA are in the *15th Workshop on Standards for the Interoperability of Distributed Simulations* (Calvin & Weatherly, 1996; Dahman, Ponikvar, & Lutz, 1996; Fujimoto & Weatherly, 1996; Miller, 1996; and Stark, Weatherly, & Wilson, 1996).

One aspect of achieving HLA compliance for an application is the construction of its simulation object model (SOM). The SOM, in conjunction with the federation object model (FOM) for a simulation exercise (federation), specifies the object model template (OMT) required by all participants in a federation. The SOM is used to document key information about the software for generating an entity, such as the types of entities supported, the information that each entity can export to

the DVE, the information each entity requires from the DVE, and the types of interactions that each entity can participate in. The SOM also documents the information that an actor or actor superclass in an application requires to operate. This information is obtained by subscribing to remote actors in the DVE. Finally, the SOM documents the information that each actor and actor class can provide to a DVE and therefore can publish. Information is made available by subscription and publication. Other DVE actors can subscribe to classes or to class types in the application. By subscribing to a class, an actor is guaranteed that it will receive all updates for all of the data values for the subscribed class. A publishable class is a class that is instantiated within the application and that will make certain information about itself available to other actors within the DVE. Subscription can take place at the class, superclass, or subclass levels. For superclasses, information is only subscribable. Subclasses can publish and subscribe to information. The subclasses are refinements of their superclasses and inherit properties from them and also expand upon their properties as well. Only single inheritance is supported within a SOM class structure. The specification of a SOM for a system requires the definition of an object class structure table, an interaction class structure table, an attribute table, and a parameter table.

The object class structure table in the SOM defines the classes of DVE actors that a simulation application can support and the remote DVE actor classes whose instances and attributes represent potentially useful information to the simulation application's local actors. These classes of local and remote actors are defined by specifying the hierarchical relationships among the various classes in the HLA application. The object class structure table specifies the classes of CGAs that the application will support and interact with, such as aircraft, missiles, etc., as well as the specific instances of each type of class that are available, such as a F-15 fighter, Sparrow missile, etc. In this table, the application designer determines the actors that the application can make available and requires in any DVE in which it participates.

The second table required in a SOM is the interaction class structure table. In a SOM, an interaction is defined as an action that an actor can perform that can poten-

tially have some effect on another actor in the DVE. The interaction class structure table specifies the types of interactions in which each actor can participate. As in the object class structure table, the interactions are described hierarchically, thereby enabling inheritance to be used to specify interactions that are common to whole classes of actors. An important specification in the table is the interaction type supported by each interaction class. Each interaction can be one that an actor in DMTITE can initiate, sense, react to, or ignore. An actor can *initiate* an interaction if it has the capability to model the initiation of the interaction and can also call the HLA send-interaction service for the interaction when it is initiated. An actor can *sense* an interaction if it is capable of utilizing information about received interactions. An actor can *react* to an interaction if it has the capability of publishing those of its attributes that have been affected by an interaction and also updating these same attributes internally.

The third table required by a SOM is the attribute table. In the SOM, an attribute is a named portion of a class of actor's state whose values may change over time. The attribute table documents the data produced and consumed by each class of actor in the application. The individual actor attributes defined in this table can be subscribed to by an actor in a remote host or published to the rest of the DVE by a local actor. By subscribing to a class or subclass, a remote actor that requires only a limited subset of information for an actor will receive only the information that it requires, thereby conserving network bandwidth and processing power. Conversely, by specifying only the information that the actor requires to function, network bandwidth and remote-host processing is conserved because unneeded information is not generated or transmitted. The attribute table in a SOM defines the attributes for each of the classes in the DVE application. Updates to these attributes are the data that actually flows among the actors that compose a DVE. To specify the attributes for a class, the name of the object class for the data item must be specified (using the same name for the class that was defined in the object class structure table). In addition, the name of each attribute for the class, its data type, the cardinality of the datatype (if it is an array), and the units of mea-

sure for the data are also specified. The resolution and accuracy for the attributes specify how accurate the data must be. The resolution of an attribute is defined to be the attribute's maximum deviation from its true value (value at its generating host) that is permitted throughout the DVE. Accuracy is defined to be the condition required to be satisfied for the resolution requirement to be met. Additional information such as the data update type and update condition are also specified, these components of the table determine if an update to the data is possible, and if so under what conditions. The attributes for the classes are defined hierarchically and data types from superclasses are inherited by the specified subclasses.

The fourth, and final, table required in an HLA SOM is the interaction parameters table. The interaction parameters table is based upon the information contained in the interaction class structure table and the attribute table. The interaction types defined in the interaction class structure table are the basis for the interaction parameters table as it determines the specific interactions to be supported for each class and superclass in an application. The attribute table, on the other hand, contains the list of available parameters that can be supplied by an actor class or subclass for any interaction. The interaction parameters table presents each generic interaction and specific interaction that is required to be supported by the application.

Computer-generated Actors: Computer-generated actors (CGAs) that exhibit believable humanlike behaviors are crucial to achieving large-scale DVEs. Approaches to achieving realistic CGAs are described by Calder, Smith, Courtemanche, Mar, and Ceranowicz, 1993; Edwards and Stytz, 1996; Laird, Newell, and Rosenbloom, 1987, 1995; and Tambe, Johnson, Jones, Koss, Laird, Rosenbloom, and Schwamb, 1995. The run-time challenges for a CGA arise from the need to compute humanlike behaviors and reactions to a complex dynamic environment at a human-scale rate of time. However, the computational challenge is eased somewhat because there is no need to replicate the human decision process; instead, only the observable aspects of human decision making must be mimicked. In addition,

the CGA's behavior must be realistic and accurate enough so that other CGAs and human participants react to its behaviors as though the CGA's avatar were human-controlled.

For our purposes, the major components of a CGA are vehicle dynamics, behavior modeling, artificial intelligence, and software architecture. Vehicle dynamics are important because the actor should move through the virtual environment accurately whether the CGA is human or computer controlled. The vehicle dynamics for computer-controlled actors should never allow a human to identify it as a CGA. Human-behavior modeling requires the acquisition of domain-specific knowledge about the domain models that humans use and about the information that humans use in the decision-making process. For a CGA in a military virtual environment, human-behavior modeling requires incorporating doctrine, tactics, knowledge models, and training into the CGA.

Artificial intelligence addresses the problems associated with assessing and reacting to the environment based upon considerations like plans, assigned mission, the activities of other actors, the available domain knowledge, and the capabilities of the vehicle that the CGA must control. The artificial intelligence component ensures that the CGA pursues its goals, responds in a proper, humanlike manner based upon its knowledge base, develops plans based upon its knowledge base, and manages other tasks. The vehicle dynamics, behavior modeling, and artificial intelligence system components are brought together within the CGA software architecture component. A flexible CGA software architecture ensures extensibility to meet future CGA requirements.

Common Object DataBase and Rapid Exploratory and Evolutionary Prototyping: The DMTITE architecture is based upon the common object database (CODB) architecture described by Stytz et al. (1997). The common object database is a data-handling architecture that uses object classes, containerization, and a central run-time data repository to manage and route data among the major objects in an DVE application. The CODB holds the entire current state of the DVE and all the public information for each threat in

operation within the DMTITE application. The CODB architecture reduces the coupling in an application's software by reducing the amount of information that a class must maintain about other classes; all of the data that an entity requires to interact with the DVE is located in the CODB, and all of the data an entity must export is placed in the CODB. To acquire public data from other DMTITE application objects, an object need only access the container in the CODB where the information resides. The world state manager (WSM) portion of the CODB handles incoming and outgoing network traffic from a simulation application and also maintains the world state for all entities controlled outside of its host.

The rapid exploratory and evolutionary prototyping (REEP) methodology (Stytz et al., 1997), is a methodology that supports quick extraction and refinement of requirements, experimentation with alternative means for satisfying requirements, and rapid incorporation into the application of the solutions developed by successful experiments. Exploratory prototyping examines an implementation solution within the context of an operational solution, and significantly accelerates the ability to assess alternative implementation solutions. The intent of evolutionary prototyping is to allow successive revisions to the overall design and implementation without making major modifications to the system. The REEP process begins with the construction of an initial prototype of the application to satisfy baseline requirements.

Baseline DMTITE Architecture: The starting point for the DMTITE architecture, shown in Figure 2, is the common object database (CODB). This architecture was developed to support the operation of a single CGA in a DVE. In this architecture, the CODB, as a run-time data repository, is used to manage data transfer between the DVE and a single actor in DMTITE. In the baseline architecture, the network interface and network component is responsible for the transmission of information between DMTITE and the other computers that are instantiating the DVE. As each DIS protocol data unit (PDU) arrives, it is forwarded from the network interface software to the WSM. The WSM is responsible

for maintaining the state of the entire DVE. Therefore, the WSM takes incoming data, updates its information about the entity, and places the information in the container for transmission to the CODB. In addition, the WSM is responsible for dead reckoning entities in-between receipt of data for the entity. As a result, when the CODB requests an update to its information from the WSM, the WSM has a container holding the most-current information about the state of the DVE ready to be dispatched.

Once in the CODB, the data is made available to the actor's decision-making system, dynamics unit, and sensors. The dynamics unit and sensors components share a container to minimize the amount of data to be transported, and together comprise the physical dynamics component (PDC). The PDC contains the description of all of the physical attributes and properties of the individual CGA. In addition to the dynamics unit and sensors, in the baseline architecture the PDC component includes the actor-specific properties, performance capabilities, weapons load, damage assessment, and physical status. The PDC also computes physical state changes. For example, the dynamics unit uses the information in the CODB to compute the current velocity and orientation of the actor, and places the results of its computations back into the CODB for use by the decision-making system and for transmission to the rest of the DVE. The sensors component uses the information to determine the actors that are within range of the actor's sensor systems and places the result into the CODB for use by the decision-making system.

The decision-making system for the baseline architecture consisted of two components (Figure 2): a skills component (SC) and the active decisions component (ADC). The SC consists of those portions of the CGF (computer-generated force) that vary between individual entities within a type and class. This component serves to model the skills and ability of the operator of an entity. The ADC contains the intelligent decision-making processes and the knowledge required to drive them. The knowledge includes the overall mission, goals and objectives, plan generation, reaction time, and crisis-management ability. In the baseline architecture, the

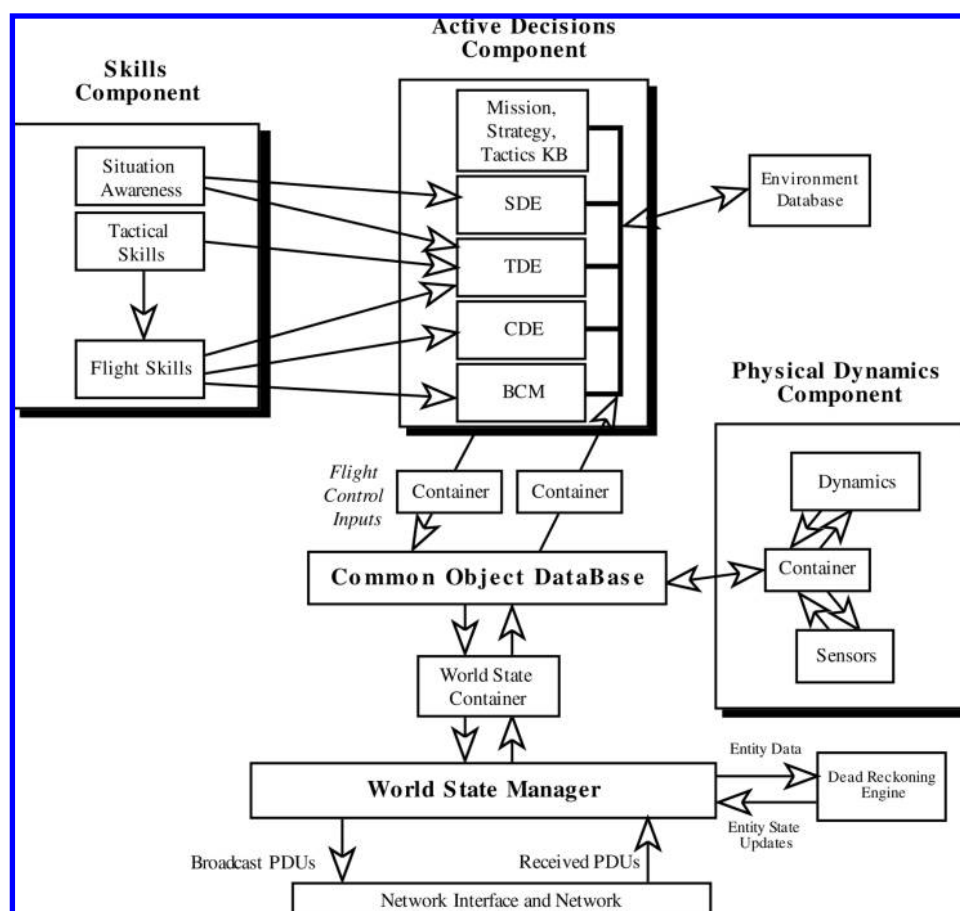


Figure 2. Baseline distributed training system application architecture showing the major architectural system components.

ADC has three reasoning engines: the strategic decision engine (SDE), tactical decision engine (TDE), and the critical decision engine (CDE). These engines perform long-term, near-term, and immediate-reasoning operations for the CGA, respectively. These decision engines (DEs) are described further in Stytz et al. (1996).

We separated the PDC, SC, and ADC components from the remainder of the CGF architecture and from each other to ensure that modifications to a component is isolated to the component and do not propagate throughout the entire system. The PDC is only responsible for the basic entity maneuver and sensing computations, and functions completely unaware of the status of the other system components. Likewise, the ADC is solely responsible for decision making and only knows

about the physical component's status based upon the data communicated to it via the CODB. In the baseline architecture, the SC is more closely tied to the ADC than the PDC because the ADC is responsible for computing control outputs for the entity based upon the modeled pilot's skills. The SC describes the pilot's ability to the decision-making component so that the decision can be appropriately constrained by the simulated pilot's abilities.

Using the information described above, the baseline architecture, and interviews with subject-matter experts, we defined the requirements for DMTITE in light of its concept of operations and the objectives for the application. The next section summarizes the results of this process.

3 DMTITE Requirements and Their Implications

As a prelude to describing our architectural solution and to support our SOM specifications and software design decisions, this section presents a discussion of the system requirements that DMTITE must satisfy. To clarify the connection between these requirements and our subsequent architecture and design, we also assess the implications of the requirements for the DMTITE software and knowledge-base architectures.

DMTITE Requirements: The requirements for DMTITE can be divided into several categories. These categories were derived from those used to specify the requirements for aircraft CGAs by Stytz, Banks, and Santos (1996). These requirements range from the software architecture that implements the CGA to the knowledge base used by the CGA to support its decision making. The origin of these requirements lies in the need to support a wide variety of aircrew training scenarios while also using a credible representation of the behavior for each modeled entity. Briefly, DMTITE should possess the capability to perform the following tasks.

- 1) hosting a wide variety of threat systems, such as anti-aircraft artillery (AAA), surface-to-air missiles (SAMs), jamming radars, acquisition radars, sound and infrared DVE actors systems, aircraft, and unmanned autonomous vehicles (UAVs),
- 2) ease of modifiability for its knowledge bases, decision-making systems, and networking services,
- 3) versatile decision mechanisms and modifiable actor behaviors,
- 4) hardware independence,
- 5) exterior (commercial) software independence,
- 6) a variety of threats at varying levels of fidelity determined by the training scenario being executed,
- 7) a variety of skills levels for each actor type for reasoning capabilities and performance skills,
- 8) terrain reasoning capability as part of operator emulation,
- 9) accurate modeling of real-world sensor outputs,

- 10) HLA compliance, and
- 11) capability to download scenarios from other DMTITE locations and reuse them.

The next few paragraphs delve into a select few of these requirements and assess their implications for the DMTITE project.

A variety of threat systems: For the DMTITE to achieve its objective for operational training, it must be capable of inserting a variety of threat systems into the DVE. One project goal is to separate the threat systems from the pilot training system and to distribute them across the network; therefore, every threat system must have at least a generic representative within DMTITE.

Modifiability: Modifiability is the ability to enhance existing CGA capabilities as well as perform software maintenance. This capability includes the need to rapidly expand a domain-specific knowledge base, the use of a flexible software architecture, the capability to operate on a variety of hardware, and independence from external software. The requirement for knowledge-base expandability addresses the need for the CGA to incorporate new strategies, tactics, and maneuvers as ally and opponent concepts change. The need for modifiability also addresses the need to maintain DMTITE in the field and supports improvement of the system's operation by permitting well-encapsulated changes to the knowledge base. A flexible software architecture likewise assists in readily adapting DMTITE to meet new performance, interface, and communication protocol requirements. Hardware independence addresses the need to be able to port DMTITE to more-capable computer hardware with minimum changes to the system's code. The need to remain independent from commercial, non-DMTITE software supports the need to remain independent of hardware and allows the system to take advantage of developments that can improve its performance.

High fidelity representations: High-fidelity representations in DMTITE are achieved by operating each CGA using accurate world representations (especially

terrain), dynamics for vehicle motion, sensor and weapon models, and models of human behavior. The world representations are based upon surface representations composed from primitive data elements organized within a hierarchical representation of the terrain data. However, because CGAs do not operate in isolation, their world representation must have a high-fidelity counterpart for manned systems as well as for other CGA systems. Correct vehicle dynamics ensures that the DVE actor only moves according to the capabilities of its real-world counterpart and does not exhibit unrealistic performance given the DVE's terrain, weather, and atmospheric conditions. Likewise, the weapons and sensor models for each DMTITE actor must use models with the same sensitivity, field of view, and range as used by their real-world counterparts. The requirement for sensor fidelity exists across all sensors, from the eyesight of the assumed operator of the CGA to the radar and infrared sensing systems used by the real-world counterpart of the CGA.

Adaptable decision mechanisms. Adaptable decision mechanisms give the CGA flexibility in dealing with situations that occur in the distributed virtual environment. Adaptable decision mechanisms permit the system to maintain robust, credible behavior for DMTITE actors under a variety of external circumstances, such as missing or inconsistent information, and at different levels of operator skill. This requirement ensures that each threat instantiated by DMTITE can operate effectively even when confronted by conflicting or incomplete information and when under system stress.

Threats at varying levels of fidelity and a variety of skills levels: The first component of this requirement speaks to the need to conserve computational power. Threat actors should be able to be instantiated at multiple levels of fidelity so that only those threats that require a high-fidelity representation use a high-fidelity model and are permitted to consume a correspondingly greater amount of available computational resources. Regardless of the level of fidelity, each threat actor should also be available in a range of skill levels. Multiple

skill levels allow the training to be tailored to the abilities of the human participants and provide a more realistic training situation because the opponents and allies exhibit a variety of capabilities. The skills can be realized by varying manual skills, by varying the range of options available to the decision-making component, by varying the knowledge about friendly and enemy tactics available to the decision-making component, and by permitting the decision-making component to forecast the impact of each available option on its ability to perform its mission.

Implications of these requirements: The above requirements have implications for system complexity, real-time performance, knowledge engineering, and scalability. We discuss these implications below.

The requirement to be able to instantiate a wide variety of threat systems within a single computer host indicates that the system must use general-purpose reasoning mechanisms coupled with threat-specific knowledge bases for reasoning about the state of the DVE and the actors within it. To conserve memory and minimize data handling, the architecture must allow the instantiated threats to share a single, shared representation of the DVE state, permit identical entities to share knowledge bases, and allow each threat to have a customizable, publish-and-subscribe profile for its interaction with the other DVE participants regardless of whether they are local or remote.

The requirement to achieve a modifiable system indicates that the system should be structured so that components are strictly isolated from each other and so that there is loose coupling among components of the system. By ensuring the isolation of system components, the architecture minimizes the impact of changes to the DMTITE application, and it also serves to retard architectural entropy. Two central consequences of the goal of achieving isolation and minimizing coupling are that data movement among components should be carefully managed within the architecture and that the programmer should be constrained to remain within the system's architectural approach. Modifiability can complicate the design because there must be a clean separation between

the knowledge representation and the decision mechanism.

The need for DMTITE to provide high-fidelity actor representations affects the system's architecture, knowledge base, and information flows. Because we require high-fidelity actor representations but must also conserve processing power, the architecture must support multiple levels of fidelity in the representations of the DVE terrain, actor vehicle dynamics, and simulated human behavior. Additionally, DMTITE must have access to different levels of detail of information so that the decision engine is not burdened with reasoning about high-detail information that is beyond its sensor range. The data flows must ensure that the information available to the decision-making mechanisms accurately models the type and quantity of information that the sensors in the actual vehicle would provide to a human operator in the real world. As regards the knowledge base, the design should encapsulate related items of knowledge within a single access unit and ensure the separation of unrelated knowledge components. Encapsulated knowledge permits the decision mechanisms to atomically access the information they require and also permits the designer to update the knowledge bases with minimal impact upon other information in the knowledge base.

The requirements for adaptability, multiple skill levels, and multiple levels of fidelity affect several aspects of the knowledge base and the decision-making components. First, the decision-making component must contend with incomplete information and uncertainty about the DVE because available information will be limited to that capable of being provided to the operator in the real world. The decision mechanism must be structured so that the amount of information considered when making the decision can be adaptively varied and so that additional possibilities can be considered as time and circumstances permit. Second, because the system requires general-purpose decision mechanisms, the knowledge-base component must be comprehensive enough to allow the decision mechanism to satisfy the requirements for multiple skill levels and multiple levels of fidelity. Finally, the need for multiple levels of fidelity affects the decision-making component in that one means of

achieving computational savings is to alter the type of reasoning performed. Therefore, the decision-making component of the DMTITE architecture must support the use of different reasoning systems for a given threat without requiring changes to the threat's knowledge base.

The next step we undertook in developing DMTITE was identification of the required inputs and outputs for the system based upon the types of interactions that each actor had to support. The next section summarizes the HLA-compliant simulation object model that resulted.

4 The DMTITE Simulation Object Model

The specification of the simulation object model (SOM) for DMTITE required the definition of an object class structure table, an interaction class structure table, an attribute table, and a parameter table. The development of the SOM is, in many ways, a refinement of the requirements identified in the preceding section. However, in the preceding section requirements were specified as high-level objectives. With the complete specification of the SOM, many of the interaction requirements and the fidelity provided by the system are determined. In our case, since we were developing the system from scratch, we were able to address the desired interactions in detail and to use this specification to guide the capabilities developed in each of the components as well as to assist in determining the features that the DMTITE architecture should exhibit.

The first table defined for DMTITE was the object class structure table, depicted in Table 1. In Table 1 and the other SOM tables, the items in the farthest lefthand column are the superclasses for DMTITE; the subclasses of a class are below and to the right. The subclasses are refinements of their superclasses and inherit properties and subscription and publication requirements from them and can also expand these properties as well. In Table 1, we specified all of the classes of actors and the subclasses of actors that DMTITE can instantiate for an exercise. For example, Table 1 shows that DMTITE has a superclass called AAA. As subclasses of the AAA class, two additional classes are defined: the generic AAA and

Table 1. DMTITE Simulation Object Class Structure Table

Class	Type
AAA	generic AAA
	ZSU-28 AAA
SAM	generic SAM
Radar	generic radar
	generic fixed radar
	generic mobile radar
Jamming radar	generic jamming radar
UAV	generic UAV
Aircraft	generic fighter aircraft
	generic bomber aircraft
	generic Wild Weasel aircraft
Missile	generic radar-guided missile
	generic infrared-guided missile
	generic laser-guided missile
	generic radar-homing missile

the ZSU-28. Data about actors in both of these subclasses can be accessed by other actors by subscribing to information about these types of actors, and, when instantiated, these DMTITE actors publish information to the DVE.

The object class structure table documents the types of actors that are available within DMTITE and the publish-and-subscribe, information-transfer relationships among these actors and the remainder of the actors in the DVE. These information-transfer relationships are the basis for the information documented in the class attribute table defined for DMTITE. However, before the class attribute table can be defined, the interaction class structure table must be specified.

The DMTITE interaction class structure table, illustrated in Table 2, specifies the interactions required of each DMTITE actor. Table 2 contains a portion of the

Table 2. Abstract of the Interaction Class Structure Table for the DMTITE Simulation Object Model

Interaction	Interaction class	Interaction type
radar_ broadcast	friendly_radar- _broadcast	initiate_friendly- _broadcast
		friendly_broadcast- _state_update
air-to-ground_ attack	opponent_missile- _attack	terminate_friendly- _broadcast
		opp_missile_launch
		opp_missile_state
		opp_missile_de- coys_deploy
air-to-air_ attack	opponent_ rocket_attack	opp_missile_impact & detonate
		opp_rocket_launch
		opp_rocket_state
		opp_rocket_impact & detonate
	opponent_missile- _attack	opp_missile_launch
		opp_missile_state
		opp_missile_impact
		& detonate

interaction class structure table of the SOM for DMTITE. Due to space limitations, we do not include all of the information that the complete table requires, but it is all present in the actual SOM. The basic type of

Table 3. *A Portion of the Attribute Table for the DMTITE Simulation Object Model*

General object class	Derived object class	Attribute name	Data type	Cardinality	Units	Resolution
Radar		Radar name	char	30	N / A	N / A
		frequency	long	N / A	KhZ	N / A
		range	short	N / A	meters	N / A
		power	long	N / A	watts	N / A
		beam width	float	N / A	degrees	.5 degrees
		beam height	float	N / A	degrees	.5 degrees
		sweep rate	float	N / A	degrees / sec	N / A
		location	long	3	meters	10 meters
		elevation limits	float	N / A	degrees	.25 degrees
		angular accuracy	float	N / A	degrees	.25 degrees
	generic mobile radar	total fuel	short	N / A	litres	liter
		remaining fuel	short	N / A	litres	liter
		direction of motion	float	3	N / A	N / A
		speed	short	3	km / hour	.5 km

interaction is described in the leftmost column, and its subclasses are placed in columns to the right. With the simulation object class structure and the interaction class structure tables defined, we can now turn to the definition of the types of interactions that DMTITE can participate in and the information that must be transferred to allow these interactions to occur. As the first step in defining the data content of the interactions, the data produced and consumed by each class must be defined and documented in the attribute table.

The attribute table in a SOM defines the data attributes in each of the classes in an application. Updates to these attributes are the data that actually flows among the actors that compose a DVE. We have found that it is generally easiest to first specify the attributes for the superclass(es) in the attribute table and then to specify attributes of the subclasses for each superclass. In our experience, the attribute table is a crucial table because it determines the scope of the data that can be used in an interaction. The omission of an attribute here will not only affect the specification of the interactions for an

actor but will also result in additional code development later in the project to rectify the mistake. A portion of the attribute table for DMTITE is presented in Table 3, and illustrates how the superclass and subclass definitional process occurs. In Table 3, the superclass we defined first is the radar class. Within this class, ten attributes are identified that are common to all of the subclasses. Note that several attributes have no resolution; this commonly occurs and is permitted in the HLA specification when the associated attribute is an enumerated type. Within the radar class, one subclass is specified, the generic mobile radar class. In this class, four additional attributes are specified. Therefore, in conjunction with the attributes specified for the superclass, every actor of the generic mobile radar class is required to publish fourteen attributes.

The last table to be specified is the interaction parameters table. A portion of this table for the DMTITE SOM is presented in Table 4; this excerpt specifies the parameters that an opponent radar must supply in an interaction with a DMTITE actor. The parameters re-

Table 4. A Portion of the Interaction Parameters Table for the DMTITE Simulation Object Model

Generic interaction	Interaction class	Parameter name	Datatype	Cardinality
radar_receive				
opponent_radar_broadcast				
	initiate_opponent_broadcast			
		radar name	char	30
		frequency	unsigned long	N / A
		range	short	N / A
		power	long	N / A
		beam width	float	N / A
		beam height	float	N / A
		sweep rate	float	N / A
		location	long	3
		elevation limits	float	N / A
		angular accuracy	float	N / A
	opp_broadcast_state_update			
		radar name	char	30
		frequency	unsigned long	N / A
		range	short	N / A
		power	long	N / A
		sweep rate	float	N / A
		location	long	3

quired for two interactions are specified: one interaction initiates a radar broadcast, and the other interaction supports a state change by the same radar. For example, Table 4 of the DMTITE SOM specifies that for an opponent radar to initiate a broadcast, it must supply ten parameters to any DMTITE actor that has subscribed to this particular type of DVE information. By virtue of simply subscribing to this interaction subclass, any DMTITE actor that needs information related to an opponent radar broadcast initiation is guaranteed to receive the information from the opponent host radar simulation system. Our practice is to place a parameter in this table at the most general level that is possible, thereby allowing other actors in the DVE to obtain useful information about our actors, and vice-versa, without requiring subscription to a multitude of more-specific interactions. This approach is by no means a standard one, and represents our best estimate of how to exchange information within the DVE at the lowest computational and

network bandwidth cost. At the present time, an argument can be made that the parameters should be specified at the lowest level in the hierarchy to which they apply, thereby allowing a remote actor to specify only the interactions that it needs, and thereby only receive the information it requires. We do not find this argument to be persuasive, but it will not be resolved until further testing of the HLA is completed.

The completion of the SOM provided us with a comprehensive guide to all of the data required to be transported from or to each actor in DMTITE. The specification of the four tables for the generic actor classes alone required forty pages. Upon examination, the tables demonstrate that massive amounts of data must be moved between each actor and the network interface, and that this movement posed a potential performance bottleneck for the system. While we remained confident that our initial baseline architecture could be adopted to meet this challenge, the amount of data to be trans-

ported forced us to reexamine our approach to containerization within DMTITE. We turn to a discussion of this aspect of DMTITE in the next section.

5 Containerization Within the DMTITE Architecture

One of the issues we had to address in DMTITE was moving the remote entity data from the network to each DMTITE actor via the world state manager (WSM) and the CODB. The data movement between the WSM and the CODB has the greatest volume in the system (numbers of bytes per second) and must be performed efficiently to achieve our performance objectives. Four issues are addressed in this component of the design. The first issue is that the system is expected to function within large-scale DVEs and will have to receive, process, and manage large amounts of data received over the network. Unless properly managed, these elementary tasks can consume a significant percentage of the computing resources for the host. Therefore, the incoming and outgoing data management tasks must be accomplished in a manner that is efficient from the viewpoint of computing resources while also allowing each of the actors in the DMTITE application to have access to current data about the state of the entities in the DVE.

The second issue is that each of the actors in DMTITE can have different requirements for DVE data, and the SOM presented in the preceding section allows for this eventuality. In current HLA thinking, each of the actors in a system can specify the subset of DVE state data that the actor needs to function properly. Reciprocally, the actor must furnish to all the other members of the DVE the data these remote actors require about the DMTITE actor so that the other actors are able to function properly. These data-interchange requirements are specified in the FOM for the DVE. While specifying data requirements for a single entity on a single machine is a straightforward process, challenges arise in satisfying these needs in a situation in which many actors reside on a single host. The first of these challenges is the need to support customized data trans-

fers among actors and the DVE without consuming an inordinate amount of computational resources by either the actor or the network portion of DMTITE. We believe that each of the actors should not need to be aware of the FOM requirements for the DVE that DMTITE is participating in; otherwise each of them would have to assemble specialized outgoing messages for each of the different information requirements imposed upon it by the other actors in the DVE. Instead, each actor should be allowed to place all of the required outgoing information into a single message, and some other portion of the system should manage information customization. This same approach should also be applicable to actor-specific transfers of data from the DVE to each of the actors in DMTITE. The second of these challenges is the need to be able to efficiently adapt to changes in a FOM, because DMTITE will be required to participate in a wide variety of HLA-based DVEs. At the actor level, changes in the FOM will be reflected in the data transferred between an actor and the DVE. We need these changes to be well encapsulated and transparent to the remainder of the actor's software and to most of the network interface software as well.

The third issue that arises from the need to support HLA is the desire to be able to readily adapt to changes in the HLA standard. We expect that there will continue to be changes to the HLA that affect the actor code and their interface to the DVE. Our desire is to isolate the actor from HLA changes as much as possible.

A fourth issue is the need to support our REEP approach to design and implementation of DVE applications. Support for REEP requires that the major system components be isolated from each other and that the actor code be separated from the DVE interface code.

As a result of examining these requirements, we refined our approach to containerization to better support the transport of data between the major components of DMTITE, as shown in Figure 3. We use containers to move data in structured groupings between the components. Within DMTITE, there is a container between the WSM and the CODB for every major entity type, such as actors, phenomenology, and electromagnetic emissions. Each container is, in turn, composed of pallets, and each pallet is composed of slots. For example, in

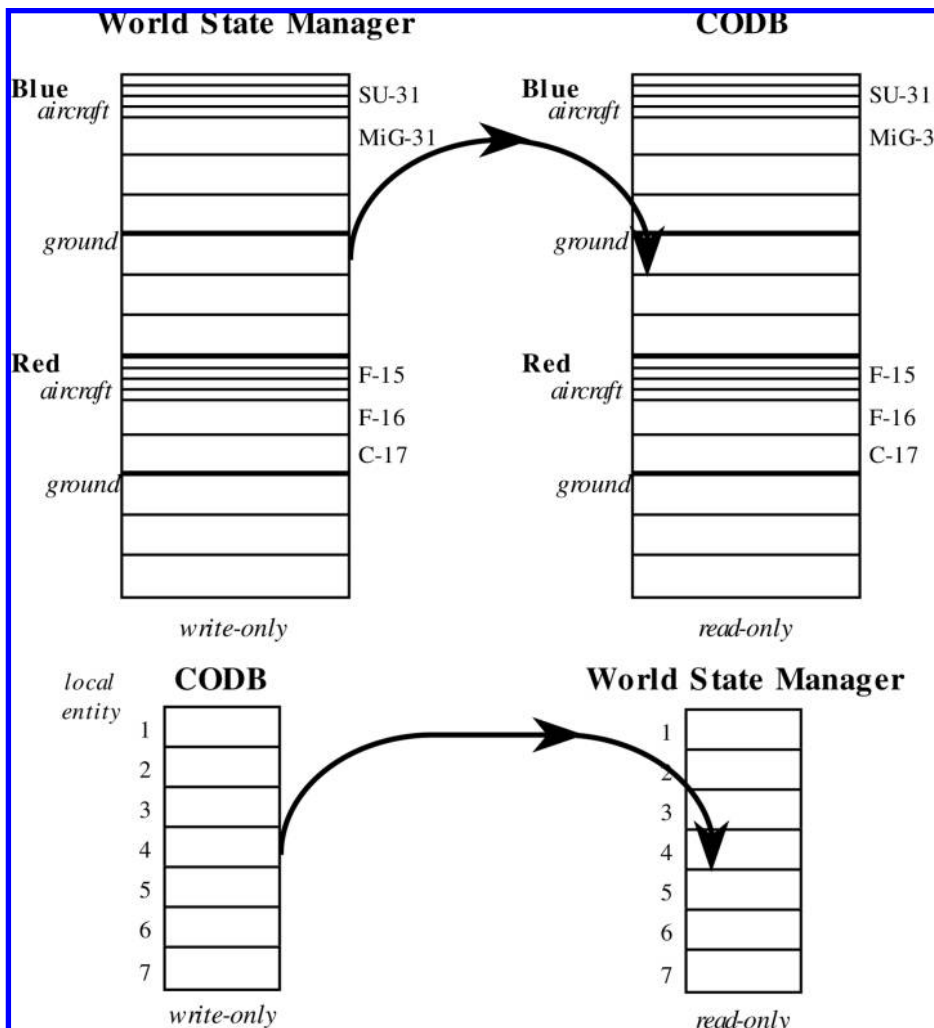


Figure 3. The use of containers to transfer data between the CODB and the world state manager.

the actor container shown in the top half of Figure 3 there are two major pallets, one for Blue (friendly) forces and one for Red (opponent) forces. Recall that we are building a threat system, so from the point of view of the threat system, the friendly forces are composed of non-US aircraft and systems and the Red forces are composed of US aircraft and systems. Within each of these two pallets are additional pallets, one each for ground and aircraft actors. And within the Red aircraft pallet there are three additional pallets, one for F-15, F-16, and C-17 aircraft. In each of these pallets are the slots that are used to hold state information for each actor of that type within the DVE, the type being Red Aircraft

F-15, Red Aircraft F-16, and Red Aircraft C-17. Each type of actor (air, land, surface, subsurface, and space) as well as each subtype for each type has a dedicated, pre-allocated portion of the container, its slot. Each pallet has a pre-allocated number of slots or subpallets. We allocate an identical amount of memory for each actor, even though in some instances this approach leaves some data space unused for an individual actor's slot in the container, and even in some of the pallets. We view this an acceptable tradeoff because for our purposes the objective is to minimize the cost of moving data between the WSM and the CODB.

Because the size and structure of the container are

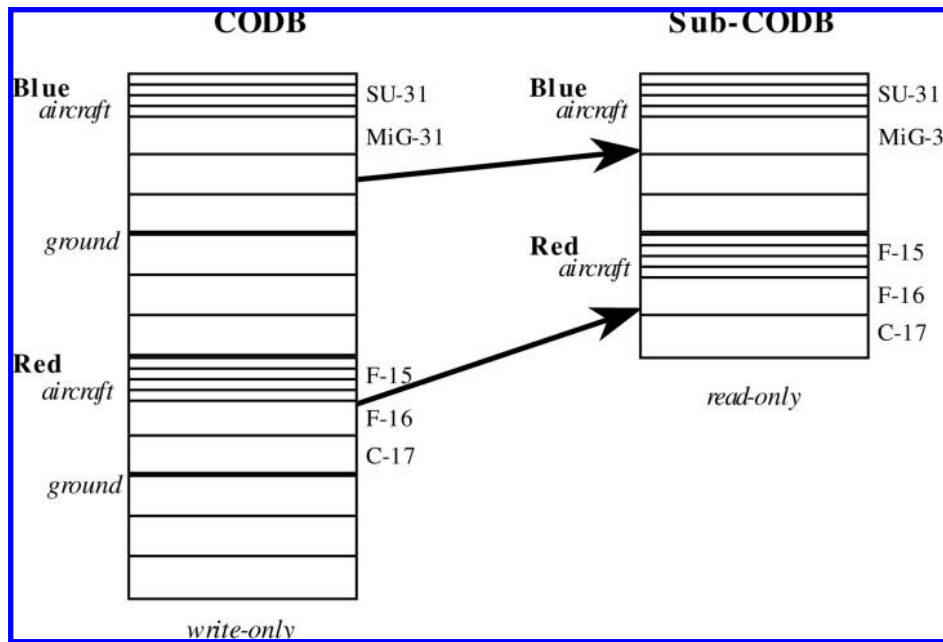


Figure 4. Repackaging data in the CODB for a sub-CODB for use by actors within DMTITE.

static, a single operation can move the data from the WSM to the CODB. To further minimize the cost of the operation, we move the data between the WSM and CODB using a double-buffering scheme. Double buffering allows the WSM to maintain the state of the DVE without concern about contention with the other system components for the data structures. As a result, we can maintain an accurate description of the DVE within the WSM while permitting the major components of the system to access data in the CODB whenever they require the data. And, conversely, each actor has convenient access to the data it needs, and the CODB can also prepare the outgoing container for the WSM when filling the container does not unduly interfere with the performance of the CODB's other duties. In this way, we can decouple the operation of these two data-intensive components of DMTITE.

Once the data is in the CODB, the actors can access the data or it can be repackaged into sub-CODBs for use by groups of actors with common data requirements, as shown in Figure 4. Figure 4 shows that a subset of the data in the CODB, the DVE's aircraft, have been repackaged and transmitted to a sub-CODB in a container for use by a set of actors that have a common set of data

requirements. Generally, repackaging would be performed when there are enough DMTITE actors with common data requirements so that the expense of repackaging is offset by the time saved by not transmitting the data to each actor individually. These sub-CODBs also can have their own methods attached to them, thereby allowing for some specialization in the data supplied to its user-actors. The repackaging is performed by methods within the CODB class, and the data is placed into an outgoing container that is transmitted to a sub-CODB. If all of the actors that access a given sub-CODB have identical data requirements, then we do not move the data from the container to a sub-CODB; instead the actors access the single container directly. In this case, we use a counter attached to the container to ensure that all of the actors serviced by the container do actually get a chance to read the container before a refreshed container is requested from the CODB.

When an actor must transmit data to the DVE, the actor uses a container to transmit the data to the CODB, as illustrated in Figure 3. For an individual actor container, the container has the same format as its slot in the CODB. If the container is shared among several actors, then each of the actors has an assigned slot to fill with its

data. Once the container is ready to be dispatched, the CODB signals the WSM to accept the data. Each actor exports all of the data required by the SOM for each container update: we rely upon the WSM and the HLA run time interface to manage repackaging the exported data to meet the FOM requirements.

With the specification of the data to be moved—and having refined our use of containerization to accommodate the large amount of data to be transmitted among the major system components—we can now turn to a discussion of the DMTITE architecture.

6 The DMTITE Architecture

In this section we describe our architectural solution to the DMTITE requirements. The solution incorporates containerization and the CODB, and addresses the data handling issues raised by the DMTITE SOM. This section opens with an overview of the architecture and the proceeds to a discussion of the data flow within the system and our approach to achieving the desired data flow within the architectural constraints.

Architectural Overview: The architectural solution presented in Figure 5 is based upon the generic single CGF DMTITE architecture outlined in Figure 2. The main architectural components of the generic CGF architecture are maintained in the DMTITE architecture; however, there are a few key differences. The architecture uses the CODB, and has provision for multiple, specialized, sub-CODBs that can be used to provide information to a select subset of the actors hosted in a DMTITE system. These sub-CODBs are shared by their actors, and have the same protection mechanisms and containerization associated with them as the main CODB. As in the generic architecture, the main CODB and WSM combination serves to transmit data to the DVE and to receive data into DMTITE. The CODB and all of its sub-CODBs are used to store and forward state information from actors hosted by DMTITE to the DVE through the WSM. The WSM is responsible for interfacing with the HLA RTI. These two components work together to ensure that each DMTITE satisfies its

DVE FOM requirements by consolidating the output from the actors and then transmitting the actor state data to the rest of the DVE in a manner that satisfies the FOM. Within an individual DMTITE system, each actor threat type shares a knowledge-base set that was assembled specifically for its actor type. However, while the knowledge bases for a specific actor type are shared by all of the actors of that type at run-time, not all of the actors utilize all of the knowledge base's information. The information accessed from the knowledge bases is determined by the fidelity level and skill level required of each actor instantiation. Currently, the knowledge bases are read-only at run-time. The decision mechanisms within each threat actor are instantiated along with the actor and are not shared between instantiations. Because knowledge bases are shared between threats, the difficult and expensive knowledge-base construction process must be accomplished only once, but the burden of achieving different levels of fidelity and skills falls upon the decision mechanisms and their use of the knowledge bases.

Even though the architecture in Figure 5 resembles that of Figure 2, a few words of explanation of the differences between it and Figure 2 are in order. The operation of the network interface and network, WSM, and the dead-reckoning engine are the same as that for the basic architecture. Once the DVE state information reaches the CODB, the data is repackaged into outgoing containers for either individual actors or for a single actor class. This repackaging is accomplished by an object manager. Once the DVE state reaches a sub-CODB, the data is dispatched from there in containers to the actors serviced by the sub-CODB. The containers that depart the CODB or a sub-CODB for an actor are customized for the actor, and contain only the DVE information required by the actor as specified in the SOM. There are two components of the threat database for each actor type: the environment database and the mission, strategy, and tactics database. The environment database for each actor type contains the specification of the terrain and other static portions of the DVE in visible wavelengths as well as the wavelengths used by the actor's sensors. The other portion of the threat database contains the information about the individual actor's mis-

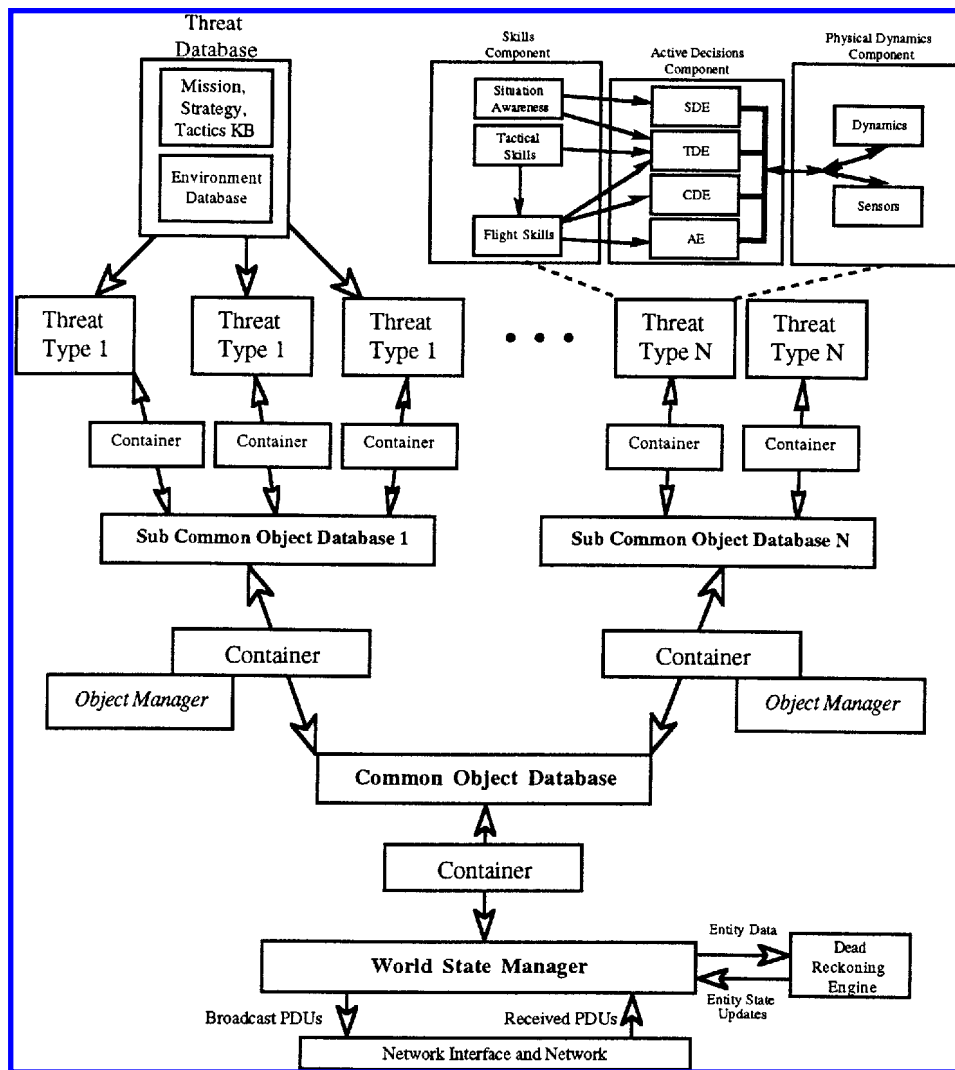


Figure 5. DMTITE system architecture.

sion, the tactics for the threat type, and the strategies to be employed by the threat type. Within each actor threat, the skills component remains as described previously for the individual actor case. The only change to the active decisions component was the replacement of the basic control module by the arbitration engine (AE). The function of the AE is to determine which of the decision engine outputs should be used as the actor's next action. We changed the name of this component to reflect the fact that its decision making became more complex so that the system could better select the output to use and also because the AE can employ a variety of de-

cision-making mechanisms when arbitrating over the outputs from the other decision engines in the ADC. The functionality of the physical dynamics component was unchanged.

When an actor has computed its new state, this information must be provided to the other actors in the DMTITE application at the host as well as to the other actors in the DVE. To accomplish this data transfer, the actor places its state information into a container that is then dispatched to the CODB (possibly via a series of sub-CODBs) for relay to the WSM. Once the new state data is in the CODB, the actor data is passed on to the

WSM for transmission to the DVE and is also repackaged into the next outbound container to leave the CODB for the actors in the local DMTITE application.

DMTITE Data Flow and Data Filtering: Within the DMTITE HLA operational environment, each system has perfect knowledge about the state of all of the entities in the DVE, which is an inaccurate portrayal of the real-world operational environment. The inaccuracy arises from the fact that known sensor limitations are not imposed upon the data; therefore, each sensor has, in effect, unlimited visibility coupled with unlimited sensitivity. As a result, the CGAs within DMTITE could have knowledge about the state of the DVE and the entities in it that is unavailable to their real-world counterparts under similar environmental and sensor conditions. To address this issue and increase the fidelity of the operation of DMTITE actors within the DVE, each threat within a DMTITE system has its information restricted by filtering the incoming data so that the CGA threat operates only upon a realistic set of information. Figure 6 illustrates how this is accomplished within the DMTITE architecture.

Figure 6 presents our approach to DVE data filtering for a single actor. Whenever a threat application acquires DVE state information, the information must always come through the CODB. However, before the actor operates upon the information in its container, the information is filtered by sensor models. The sensor models restrict the information provided to the actor so that the information available to the actor matches that provided to a real-world counterpart system. After filtering, the environment data is used in conjunction with the knowledge bases by the decision engines (DEs) for their decision-making computations. The AE takes the results from each DE and decides which of the results should be used as the output. The decisions from the AE are then forwarded to the CODB where other threat components within DMTITE and in the DVE acquire the outputs for their computations.

The above model for data filtering is acceptable for a single-actor system, but the requirement to host more than one actor in a DMTITE necessitates a modified approach to data filtering when multiple actors are using

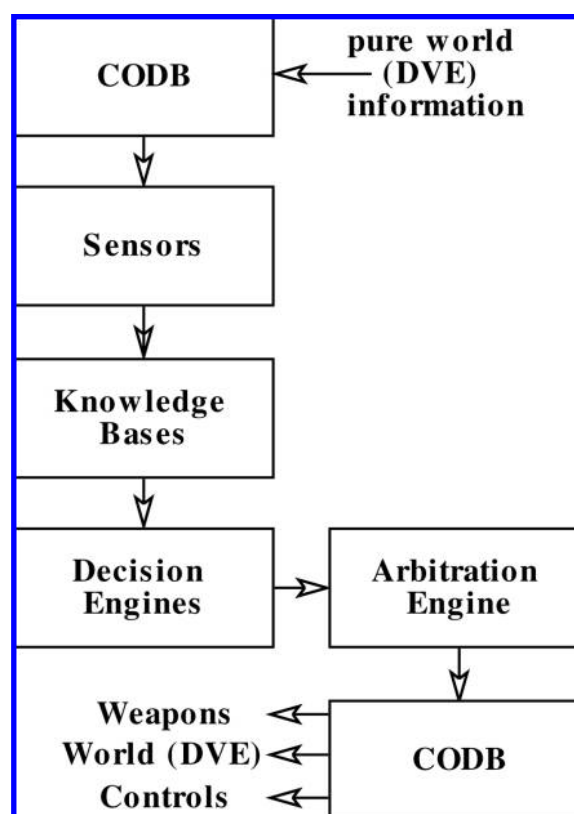


Figure 6. Distributed virtual environment data flow through the application for an individual DMTITE threat.

the dynamic environment data in a single CODB. Our model, as shown in Figure 6, places all of the information that an actor requires for decision making into the actor's knowledge base. The difficulty arises from the fact that in DMTITE the dynamic environment data destined for an actor type is transferred to the actors within a single container, which is then operated upon by a single sensor model before it is placed into the knowledge base. By virtue of the knowledge base being shared by all of the actors of a single type, the output of the sensor model is then shared by all of the actors of a type, even though the actors do not have identical sensor input requirements. Therefore, we modified our model for the information flow from the DVE to each individual actor. This new information flow model is depicted in Figure 7.

As shown in Figure 7, the transfer of data from the CODB to the actor is now mediated by new operations.

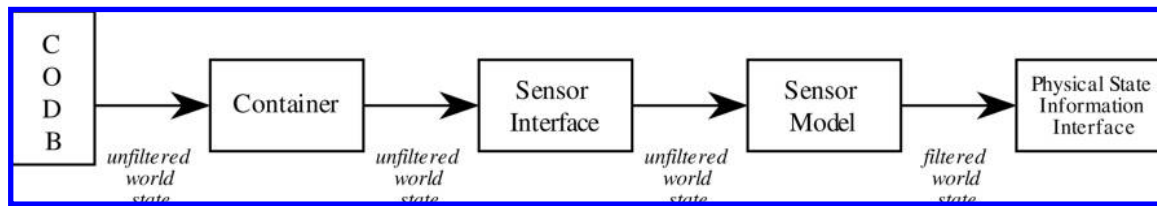


Figure 7. Processing of world state data using actor-specific sensor models within DMTITE.

As before, the information from the CODB required by an actor's SOM is dispatched via a container to the actor. However, instead of the actor accessing the container directly, the actor accesses the data after it has been processed and placed in the physical state information interface. Recall that each CODB has methods attached to it to manage the packing and unpacking of the container. We built upon this method-based access to implement filtering. In our revised approach to DVE state filtering, actor data access is initiated upon the arrival of a new container from the CODB.

The container unpackaging software, called the sensor interface, extracts data from the container and forwards it to the sensor models, called the physical component, for processing. However, this approach to DVE state filtering is not complete because it ignores processing that may have to be performed to support decision making when phenomenology and weapon systems (both friend and foe) must be considered in the decision-making process. These considerations resulted in additional change to the architecture, as shown in the diagram in Figure 8.

Figure 8 presents the final view of the DMTITE architecture. In this figure we emphasize the data flow and sensor interaction aspects of the architecture. This view of the architecture illustrates how the architecture supports the data flows and information-filtering objectives presented in Figure 5 and Figure 7. To keep the diagram as uncluttered as possible and to better illustrate the data flow for sensor filtering, the containers and CODB have been omitted from this diagram, but they are still present in the architecture. In the revised DMTITE architecture, we model sensor data filtering and the actor's response to the filtered data as a two-stage process. The first stage is modeling of the physical world state, and the second stage is reasoning upon the resulting state

information. The reasoning outputs are then used to control the actor, to feedback into the data filtering process, and to generate outputs for the DVE.

In this version of the architecture, the sensor interface still serves as a data warehouse and data router between the container and the physical component models. The physical component models the physical operation of the sensors and incorporates phenomenology and weapon information into its filtered output DVE state information. The output of the physical component is passed to the decision-making component via the physical state information interface (PSII). The PSII stage of DVE state processing routes the information from a physical component to the actor-specific knowledge bases that require the particular type of information produced by a sensor. The operation of the PSII in many ways parallels that of the CODB: the PSII functions as a run-time data repository for the physical model computations of the sensor outputs and passes the information to the decision-making components of the actor that requires the information. The incoming DVE state data is used in conjunction with the information contained in the actor-type, specific-threat knowledge bases by the SDE, TDE, and CDE to perform their long-range, mid-range and immediate decision-making functions. The SDE, TDE, and CDE place the outputs of their computations into the AE, which selects the actions that are fed back to the physical components to be acted upon, the dynamics and sensor components, and can also be sent out to the DVE as well.

7 DMTITE System Design

The design for DMTITE is object based and strongly reflects the structure suggested by the

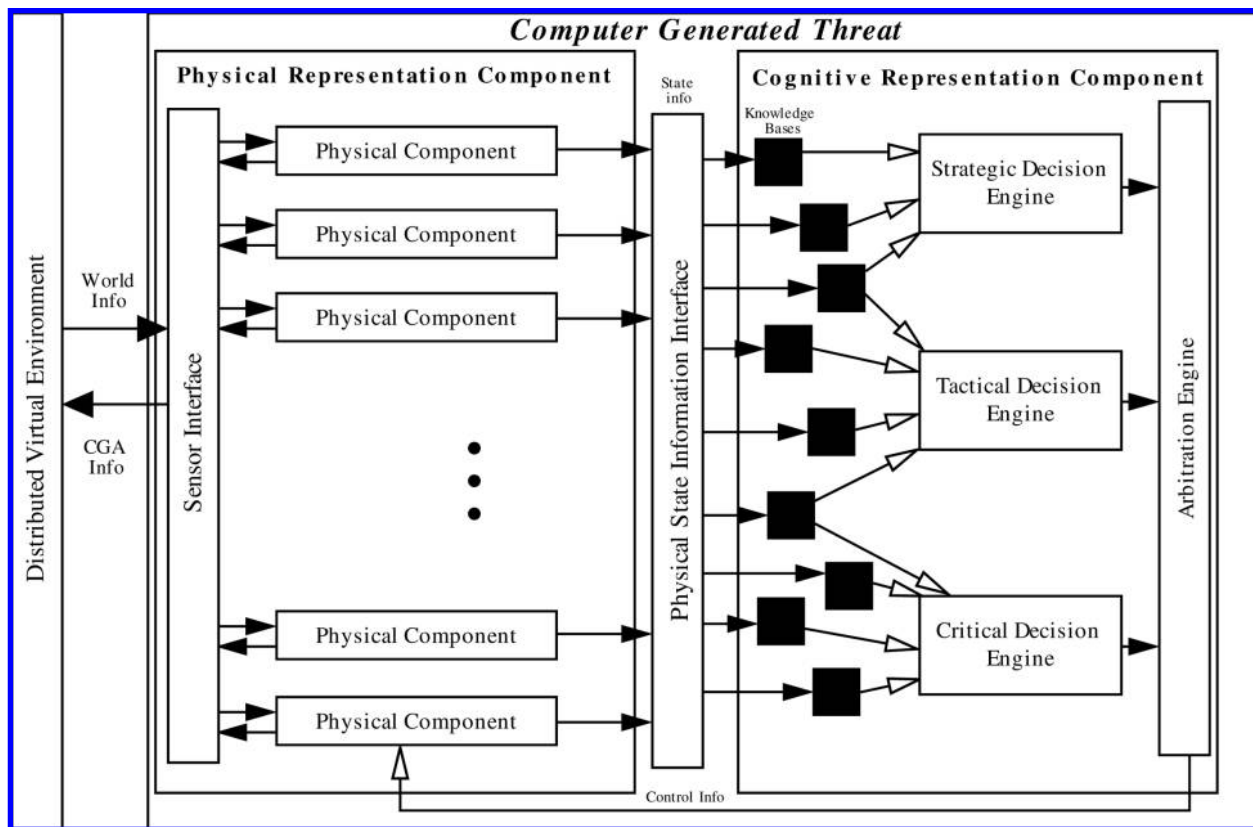


Figure 8. Data flow through the DMTITE architecture.

DMTITE architecture. We endeavored to maintain a straightforward mapping between the elements of the architecture and the design for several reasons. We believed that a close mapping would allow us to better trace the effects of the architecture on the eventual implementation, would provide us with a system that was readily explainable and easily modified, and would also allow us to readily identify elements of the architecture and design that required changes if problems were identified. The design description that we present in the next few pages is current at the time of this writing and reflects the effects of several iterations on both the architecture and design. We will first discuss the design of the CODB component in this section and then conclude with a discussion of the design for the DMTITE actor software. All of the software is written in C++ for Silicon Graphics computers running IRIX 6.2.

The common object database design: The CODB is crucial to the effective operation of DMTITE

because all incoming and outgoing DVE information is maintained by and routed through it. In the design of the CODB, the most important issues were ensuring uncorrupted access to the data in the containers and structuring the CODB and its containers to hold all of the information. To simplify the data management process, we decided to maintain a strict separation between the containers that bring in data to an actor from the CODB and those that send data from the actor to the CODB. When information comes into an actor, the data comes in a container that is read-only for the actor and write-only for the CODB. When state information departs an actor, it is placed in a container that is write-only to the actor and read-only for the CODB.

In the design of the container class, we also differentiate between persistent and nonpersistent data. Persistent data is data tied to an entity that has a relatively long lifespan in the DVE, but some components of the persistent data may change over time. For example, the existence of an entity in the DVE is a relatively persistent

piece of data even though its velocity and location, and hence its state, may be continuously changing. Nonpersistent data, on the other hand, is usually a singular event, has a relatively brief lifespan, and is needed by each DMTITE actor only once. For example, a weapon detonation is a nonpersistent event, and, typically, once an actor has processed the fact that a weapon detonated, the actor no longer needs the information. For persistent and nonpersistent data containers, the readers are monitored. When all readers of a persistent data container have accessed the container, the container is updated with new information. In the case of a nonpersistent data container, once all the readers have accessed the container, the container's contents are discarded, and the container becomes available to hold information about the next transient DVE event to occur.

A persistent container holds DVE entity state data, data that specifies the type of container (entity, phenomenology, etc.), and four additional fields that specify where in the container the information for the destination object is stored. The four additional fields provide the actor with a map of the data in the container, so that the location in the container of the palettes of information and even individual actor state data are specified. The CODB obtains the information concerning the types of information each actor is interested in when the actor is initialized. When an actor is initialized, it connects to the CODB and specifies the information that it requires and the types of information that it will supply (which must be identical to the information specified in the SOM). When the actor has information that is ready to be transmitted, the actor loads its container and then sends the information to the CODB by connecting to the CODB and informing the CODB that its outbound container is ready. As part of the data passed at this time, the actor also informs the CODB which portions of the actor's container have changed since the last container was sent by that actor. Conversely, when the CODB has information ready for an actor, the CODB connects to the actor and informs the actor that the CODB outbound container for the actor is ready and of those portions of the container that have changed since the last container was dispatched to the actor. The information passed by the CODB includes data on the number of

entries in the container (which is fairly constant as this value only changes when an entity that the actor has requested information about enters or leaves the DVE), the entity ID for each container entry, and the location in the container of the entity's data. Because the writer controls the dispatch of the container to the reader, we do not need to address mutual-exclusion issues in the design. However, one issue addressed is of multiple readers of the same container or sub-CODB. To ensure that all entities that require data from the container or sub-CODB have a chance to read, we attach a counter to the container. The counter is incremented when an entity finishes its read, and, when the counter value matches the number of readers for the container or sub-CODB, then the last entity to increment the counter connects and signals that new data is required. Of course, a semaphore protects the counter value.

The DMTITE Actor Software Design: Our approach to the design of the software to implement each DMTITE actor was to develop a set of objects that mirror the architecture in the major system components for each actor, as shown in Figure 9. For each actor in the DMTITE application, we have four main objects, the physical representation component, the physical state information interface, the dynamics component, and the cognitive representation component. Each of these four components is in turn composed of a variety of additional objects.

The physical representation component (PRC) has two major subobjects, the physical model and the sensor interface. The physical model component implements the functionality of the physical component called for in the architecture. The implementation of this object allows us to encapsulate one or more physical models for the operation of a sensor within a single package for the actor, and each actor can have one or more physical models. Because the information that is provided to the physical model is encapsulated within a standardized state message, we can interchange different physical models to change sensor functionality in an actor. Also, because we hide the physical model's functionality, we have been able to incorporate existing sensor models into DMTITE with relative ease. The other component

of the PRC is the sensor interface. The sensor interface is responsible for extracting information from the incoming container and providing each sensor model with the information that it requires to function. The information is transferred from the sensor model to the physical state information interface using a state message. Once the sensor-filtered information is in the physical state information interface, the information is incorporated into the knowledge base repository for use by the decision engines.

Each DMTITE actor contains the four DEs shown in Figure 9. Each of the four DEs contain a set of knowledge base IDs and variables. The knowledge base IDs identify the knowledge bases that the actor will use for decision making. Each of the DEs can use a different type of reasoning mechanism for its operation, the choice of decision mechanism for an engine is completely independent of the type of decision-making mechanisms used in the other three engines. The only requirement that we impose upon the DEs is that their decision-making systems be completely self-contained. The knowledge bases are also constructed independently of the DEs, one of the requirements that we impose on the knowledge bases is that the information they provide must be able to be presented to each of the DEs in a format that is useful to the reasoning mechanism used by the particular DE being served. The AE has one unique aspect missing from the other three engines, it contains a combat skills (CS) model. An example of the use of a CS model is contained in Figure 10.

In Figure 10, we present a rule for specifying how anxiety and anger, the skill variables, combine to affect reaction time. The rule is that if anxiety has a value that exceeds 0.70, and if the value for anger exceeds 0.50, then the CGA's reaction time is increased. This rule can be used for all levels of skill for a type of CGA because the outcome of the evaluation of this rule for a CGA is dependent upon two types of variables, trait variables and effects variables, and their combined effect upon a skill variable and not upon any data contained in the rule itself. A trait variable value is constant for each modeled skill used by the CGA throughout the entire time it operates in the DVE. For example, the CGA in Figure 10 has little experience in combat but has an aggressive atti-

tude. Because of the lack of experience coupled with a positive attitude toward being in combat, the anxiety trait is set to a value of 0.15, which indicates that the CGA will be more anxious than a veteran, but calmer than most inexperienced pilots. The value for the anger trait is set to 0.45 to reflect the CGA's inexperience; we would expect an inexperienced human in similar circumstances to be somewhat angry at the enemy. From time to time during the course of its operation, the CGA will encounter situations that can cause anxiety, anger, or both to change. These dynamic changes to the CGA combat skills are reflected in the effects variables. In the situation reflected in Figure 10, the loss of the lead aircraft while under attack by overwhelming force affects both the anger and anxiety skill variables. Because of the odds against the CGA surviving the encounter are increased, anxiety should also increase. The value of 0.30 for the anxiety effects variable reflects this change. However, the situation also affects the CGA's anger effects value, which we set to 0.10 to reflect the loss of the lead aircraft and pilot. To determine if the rule should fire, the values for trait and effect are summed to yield an instantaneous value for their associated skill variable. In this circumstance, the anxiety skill variable has a value of 0.45 and the anger skill variable has a value of 0.55, so the rule does not fire.

The combat skills model is used by the AE to refine the outputs from the DEs to match the skill level desired for the particular actor. The CS model provides us with a means to depict aggressiveness, ability, capability to anticipate enemy maneuvers, and ability to apply knowledge as expressed by the doctrine and tactics contained in the knowledge bases. In addition, the CS model allows us to portray a particular actor's promptness in obeying orders, the actor's morale and anxiety about its mission, and the simulated eyesight acuity for the actor. When required, we can establish a new skill level by specifying the appropriate parameters for the CS model. The skill model is used in conjunction with the knowledge bases needed by the AE decision-making mechanisms to select outputs from those provided by the other DEs so that the actions performed by the actor match its desired skill level. Within the AE, the CS model can be rule-based, case-based or fuzzy-set based; in this regard

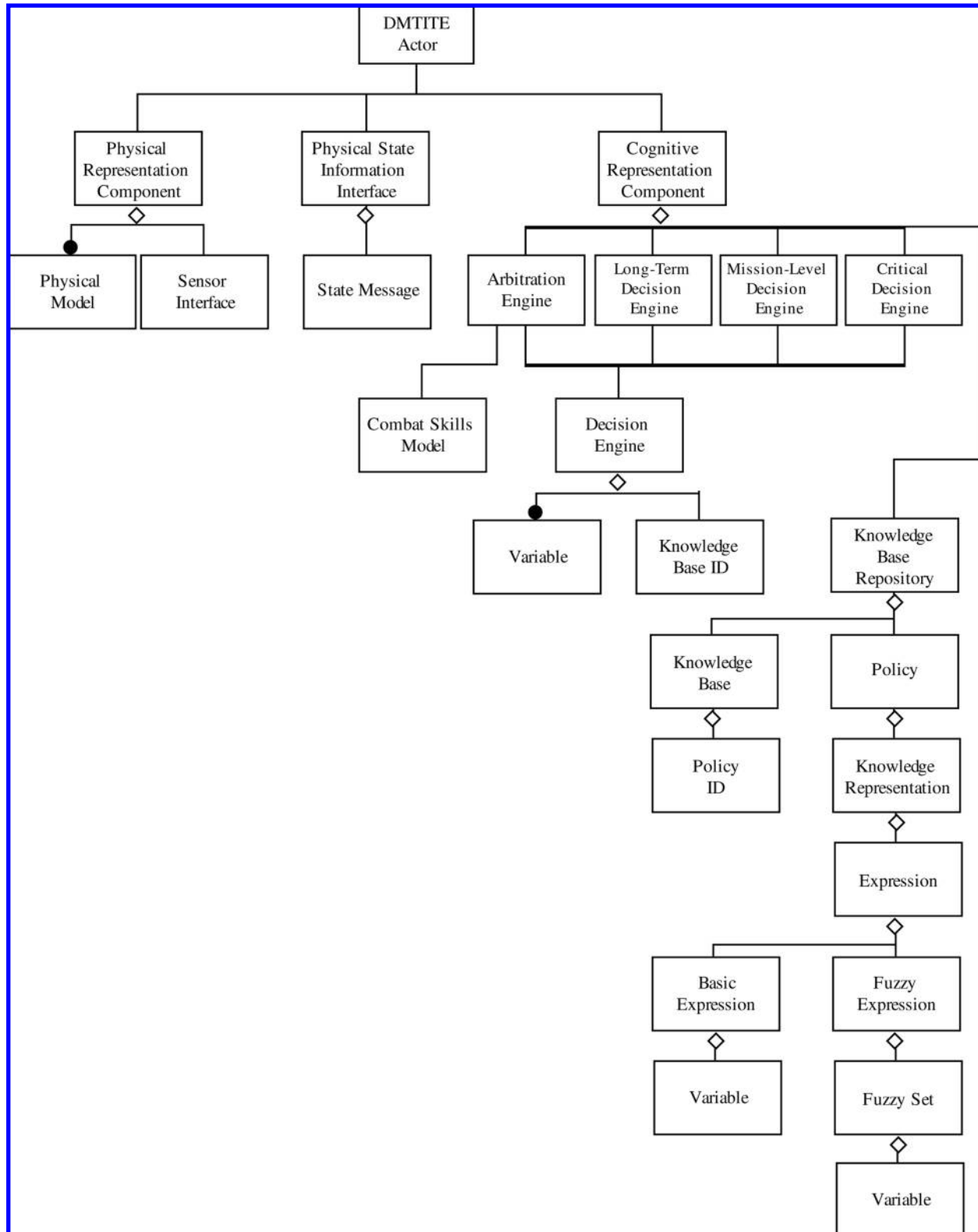


Figure 9. Software design for a DMTITE computer-generated actor.

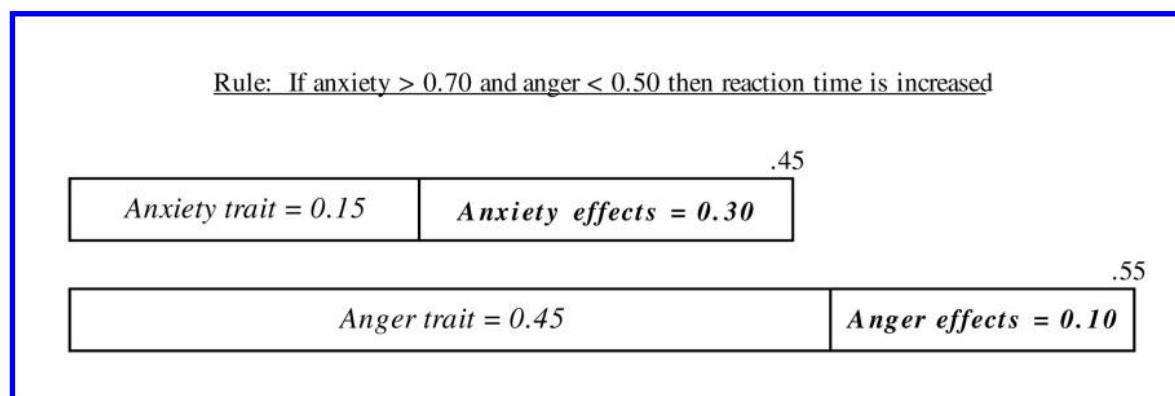


Figure 10. *Combat psychology model example for an aggressive CGA that is inexperienced in combat reacting to the shutdown of its lead while being attacked by overwhelming force.*

the AE operates like all the other DEs in that it does not know how the information is represented in its knowledge bases. By isolating the skill-level properties for the actor within a single component of the CRC, we can more easily specify a skill level than if we distributed this aspect of the actor's decision making capability among all four of the decision engines.

The knowledge base repository, portrayed in Figure 11, is the final component of the DEs design. The knowledge base repository holds all of the knowledge required by an actor. The components of an actor's knowledge base are specified when it is configured, and each of the decision engines is thereby able to access the repository when required by using these IDs.

In Figure 11, the expression class establishes the access methods to an expression. The access methods are independent of the reasoning strategy used by a decision engine or the type of knowledge expression contained, such as case-based, rule-based, or fuzzy logic. As shown in the figure, within the knowledge base repository are a number of knowledge bases. A DMTITE knowledge base is composed of a set of policies. A policy is a self-contained unit of knowledge, for example, a checklist for dealing with in-flight emergencies. Policies are atomic and nonoverlapping; they do not rely upon knowledge contained in other knowledge bases within DMTITE. This approach to knowledge partitioning allows us to expand the contents of a knowledge base by adding poli-

cies to a knowledge base or by extending the contents of an individual policy. Policies and knowledge bases can be shared among actors in DMTITE. A policy, as shown in the diagram, can be either a basic policy (currently supports case-based or rule-based reasoning) or a fuzzy policy (one that supports fuzzy logic).

In DMTITE, because the knowledge implementation is separated from the reasoning engine that acts upon that knowledge, the DEs need to know only which knowledge bases to access for information, which is provided in knowledge base IDs, and the manner for extracting knowledge from the particular knowledge base, which is provided in the policy. This separation of knowledge and reasoning engine allows us to experiment with different types of knowledge representations and decision mechanisms without changing the software design.

The fourth and final component of every DMTITE actor is its dynamics component. The dynamics component holds the dynamics model for the actor and is used to move the actor throughout the DVE in a realistic and accurate manner. Each actor has at least one model and may have several. If the actor has several dynamics models, we generally attempt to construct them so that we have several different levels of dynamics fidelity to choose from. In this way, when an actor must move in a highly accurate manner for a particular scenario, we can employ a high-fidelity model, but, in other circum-

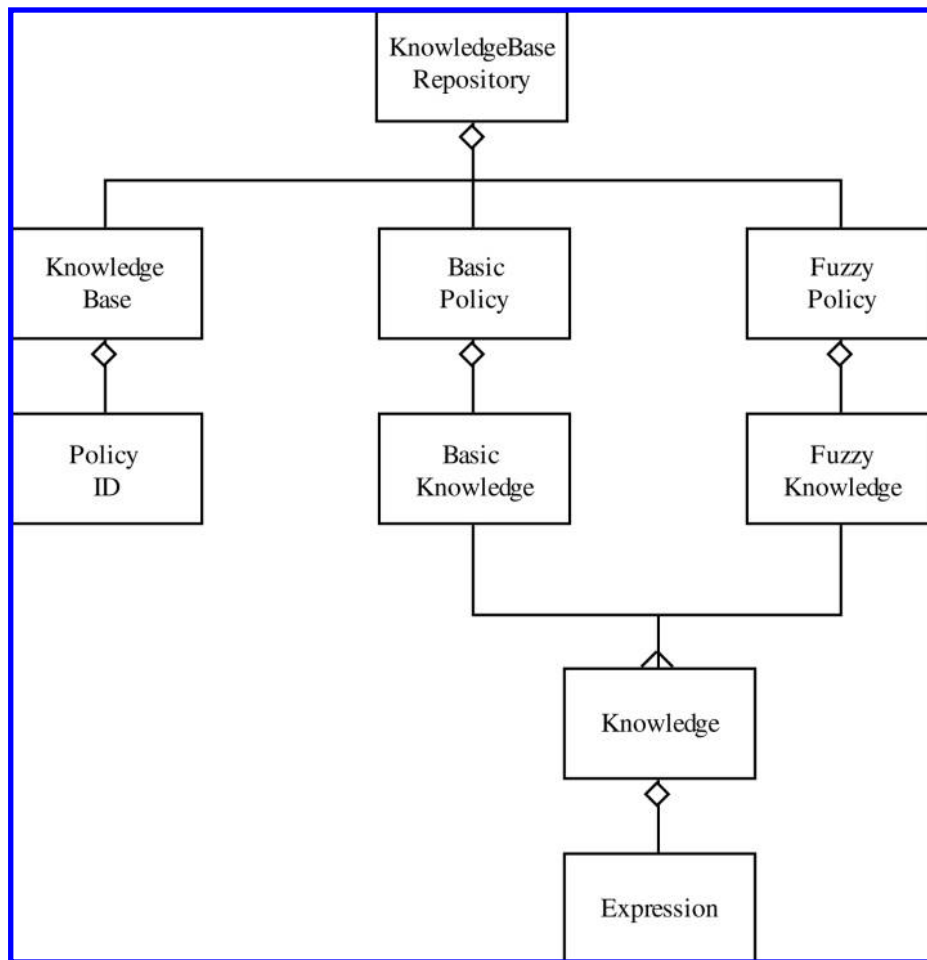


Figure 11. The design of the DMTITE actor knowledge base repository.

stances when dynamics is not as important and we need to conserve computational power, we can employ a lower-fidelity model that consumes fewer CPU resources.

8 Summary and Future Work

In this paper we presented an architectural solution to the requirements for a distributed mission training threat system. The architectural solution is based on the CODB architecture and permits the use of REEP, ensures the isolation of reasoning components from knowledge-base components, permits multiple threats

to be instantiated within a single DMTITE host, and restricts available information to a model of the information available in the real world.

Several issues remain to be addressed in DMTITE and in CGAs as a field. One of the most pressing issues is the cost of assembling a knowledge base for an actor. As might be expected, the cost of the knowledge base increases with the complexity of an actor's desired behavior, so that when an actor must faithfully emulate an aircraft the amount of knowledge required is quite overwhelming. One step we took in DMTITE to ameliorate this problem was incorporating the capacity to reuse knowledge bases across actor types. Our goal is that we will have to assemble and maintain a particular

knowledge base only once. However, this solution is not a final one, and we do not foresee a solution until better models of human behavior are developed. At this stage of CGA development, we have no model to guide us in the selection of information to be included in a knowledge base, and we are generally forced to rely upon subject-matter experts to furnish guidance. However, this is a notoriously slow procedure.

Another pressing issue is the need to be able to validate the behaviors of a CGA without examining all of the possible DVE states to ensure that the response of the CGA is correct. We do not see a quick solution to this problem, but it may lie in being able to be addressed by reusing knowledge bases, validating behaviors against standardized scenarios, and by developing better behavioral models. In any regard, the development of better behavioral models is an important problem because it is the only tool we have to guide us in the development of CGA capabilities and in populating knowledge bases.

Another question that remains unanswered is the number of actors that should be serviced by a CODB or sub-CODB. As the number of actors per CODB or sub-CODB increases, the system saves time in moving data to the actor, but the amount of time required for all the actors to retrieve their data increases. As the number of actors per sub-CODB decreases, the servicing time decreases, but the proportional amount of time required to move data to a sub-CODB or servicing container increases. We suspect that the number of actors serviced will depend upon the acceptable delay in data arrival that the actors can tolerate, the number of actors per CPU, and the amount of data that must be loaded into the container.

The final issue we believe should be addressed is the challenge of skill levels. As we currently conceive of the problem, skill level and fidelity are separable in a system, but we have no evidence to support this conjecture. Another component of this issue is that of properly defining the skill level for an actor in a way that a computer can use to modify the output from a decision process. We have taken a first step in this direction with the use of a combat skill model, but this model is by no means a complete picture of the skills that an operator brings to

bear when using a system. The combat skill model addresses the psychological aspects of the operator, which are those issues that relate to aggressiveness and willingness to engage in combat and a few physical attributes, such as eyesight acuity. However, the model needs to be expanded to account for physical factors, such as tolerance of G forces, exhaustion, and physical coordination, as well as additional psychological factors such as mental alertness, creativity, and the ability to improvise.

References

- Blau, B., Hughes, C. E., Moshell, J. M., & Lisle, C. (1992). Networked Virtual Environments. *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 157–160.
- Blau, B., Moshell, J. M., & McDonald, B. (1993). The DIS (Distributed Interactive Simulation) Protocols and Their Application to Virtual Environments. *Proceedings of the Meckler Virtual Reality '93 Conference*, 19–21.
- Calder, R. B., Smith, J. E., Courtemanche, A. J., Mar, J. M. F., & Ceranowicz, A. Z. (1993). ModSAF Behavior Simulation and Control. *Proceedings of the Third Conference on Computer-Generated Forces and Behavioral Representation*, 347–356.
- Calvin, James O., & Weatherly, Richard. An Introduction to the High-Level Architecture (HLA) Run-time Infrastructure (RTI). *15th Workshop on Standards for the Interoperability of Distributed Simulations*. 705–715.
- Dahman, Judith, Ponikvar, Donald R., and Lutz, Robert. HLA Federation Development and Execution Process. *15th Workshop on Standards for the Interoperability of Distributed Simulations*, 327–335.
- Edwards, M., & Stytz, M. R. (1996). The Fuzzy Wingman: An Intelligent Companion for DIS-Compatible Flight Simulators. *The SPIE/SCS Joint 1996 SMC Simulation Multiconference: 1996 Military, Government, & Aerospace Simulation Conference*, 28(3), 77–82.
- Fujimoto, Richard M., & Weatherly, Richard M. HLA Time Management and DIS. *15th Workshop on Standards for the Interoperability of Distributed Simulations*, pp 615–628.
- Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, 33, 1–64.

- Laird, J. E., et al. (1995). Simulated Intelligent Forces for Air: The SOAR/IFOR Project 1995. *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, 27–36.
- Miller, Duncan C. The DOD High-Level Architecture and the Next Generation of DIS. *15th Workshop on Standards for the Interoperability of Distributed Simulations*, 799–806.
- Stark, Thomas S., Weatherly, Richard, & Wilson, Annette. The High-Level Architecture (HLA) Interface Specification and Applications Programmer's Interface. *15th Workshop on Standards for the Interoperability of Distributed Simulations*, 851–860.
- Stytz, M. R. (1996). Distributed Virtual Environments. *IEEE Computer Graphics and Applications*, 16(3), 19–31.
- Stytz, M. R., Banks, S. B., & Santos, E. (1996). Requirements for Intelligent Aircraft Entities in Distributed Environments. *18th Interservice/Industry Training Systems and Education Conference* (publication on CD-ROM).
- Stytz, Martin R., Adams, T., Garcia, B., Sheasby, S. M., & Zurita, B. (1997). Rapid Prototyping for Distributed Virtual Environments. *IEEE Software*, 14(5), 83–92.
- Tambe, M., Johnson, W. L., Jones, R. M., Koss, F., Laird, J. E., Rosenbloom, P. S., & Schwamb, K. (1995). Intelligent Agents for Interactive Simulation Environments. *AI Magazine*, 16(1), 15–40.