Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

ENHANCEMENT OF MEMORY POOLS TOWARD A MULTI-THREADED IMPLEMENTATION OF THE JOINT INTEGRATED MISSION MODEL (JIMM)

David W. Mutschler

Naval Air Systems Command (NAVAIR)
Naval Air Warfare Center – Aircraft Division (NAWC-AD)
Patuxent River, MD 20670, U.S.A.

ABSTRACT

The Joint Integrated Mission Model (JIMM) is a legacy real-time discrete-event simulator. Its initial single-threaded implementation employed a memory pool to speed up run-time performance and easily checkpoint simulation state. Unfortunately, when JIMM started migrating to a multi-threaded implementation, this legacy memory pool was quickly identified as a bottleneck. This problem is addressed by dividing the memory into large chunks managed by a global controller but where thread-specific memory managers handled lower level memory allocation. This paper will focus on the legacy memory pool in JIMM and enhancements necessary for an efficient multi-threaded implementation.

1 THE JOINT INTEGRATED MISSION MODEL (JIMM)

The Joint Integrated Mission Model (JIMM) is a legacy real-time discrete-event simulator employed by the NAVAIR Air Combat Environment Test and Evaluation Facility (ACETEF), the Joint Strike Fighter Program Office (JSFPO), and other agencies for constructive analyses, training, and installed system test (Lattimore et al. 2005). Specific uses of JIMM include analysis of swarms of Unmanned Aerial Vehicles (Niland and Skolnik et al. 2005), radar simulation (Worsham 2002), Goal-Oriented Human Performance (Hoagland, Martin, Anesgart et al. 2001), and Weather Effects in Combat (Kelly, Vick, Schloman, and Zawada 2004). JIMM was initially created in 1998 as a merger of the Simulated Warfare Environment Generator (SWEG) (Lattimore et al. 1996) and the Suppressor models and thus, is derived from a line of models dating back to 1968.

To generate scenarios, JIMM uses its own simulation language known as the JIMM Conflict Language (JCL) as input. JCL uses generic systems and basic tactical criteria to build complex players with extensive tactics and doctrines. Coupled with its extensive and programmable data capture, JIMM is highly useful in standalone constructive analyses (Duquette, Nalepka, and Luczak 2004; Mutschler 2005; Nalepka 2000).

In addition, JIMM permits integrated operation through a shared memory protocol known as Simulated Warfare Environment Data Transfer (SWEDAT). With this protocol, any number of external systems can be interfaced into a JIMM scenario and thus, act and react as if operating in the simulated environment (Mutschler 2005). When operating at real-time, JIMM thus provides a threat environment highly useful for installed system test. Furthermore, in addition to hardware, the interfaced system could be a virtual cockpit, a stealth viewer, an engineering level simulation, or another threat environment provided by another protocol such as the Distributed Interoperability Simulation (DIS) or the High Level Architecture.

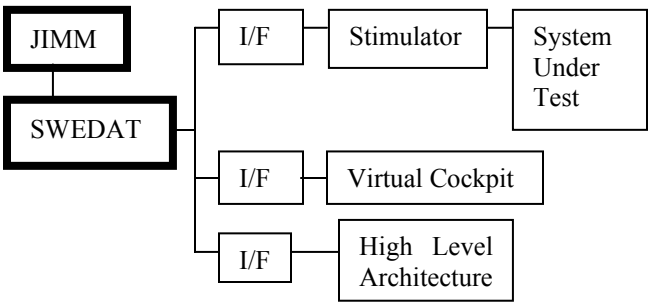

Figure 1: JIMM and the SWEDAT Architecture

Internally, SWEG (and hence JIMM) was initially implemented with a single-threaded architecture using the C++ programming language. However, as entities became more complex and as more entities were employed in scenarios, JIMM had difficulty meeting its real-time deadlines. Extensive work by ACETEF and others to improve

performance was highly successful. However, the limits imposed by the single-threaded architecture became apparent (Mutschler 2005).

Work to migrate JIMM to a multi-threaded implementation was started in the year 2000 when the High Performance Computing Modernization Program Office (HPCMPO) selected the effort as Project 7 of the Forces Modeling and Simulation (FMS-7) Computation Technology Area (CTA) (Michelletti 2003). The overall approach employed POSIX threads or "pthreads". First to be implemented were separate threads for output and then later, execution of events in parallel (Mutschler 2005). Early in the effort, the memory pool was identified as a significant potential bottleneck.

2 THE LEGACY MEMORY POOL

Both the SWEG and the JIMM simulators employed a legacy memory known as "general purpose memory" (Bulka and Mayhew 2000) or "gpMemJnr" (Lattimore et al. 2005). Though later translated to the C++ programming language, SWEG was initially written in FORTRAN and this framed the construction of the memory pool (Lattimore et al. 2005). The memory pool is essentially one large 32-bit integer array. A "free" index is maintained in the array to show what memory is allocated for use within the simulation. Initially, memory of a needed size is obtained by moving the free index further in the array.

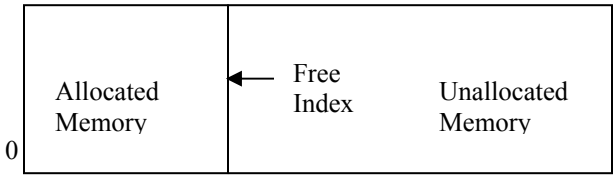

Figure 2: The Memory Pool and the Free Index

When memory was no longer needed and if it was located next to the free index, then the free index could be moved back. More often however, the memory was stored in an array of lists of the same-sized memory chunks. The size of the chunks was always aligned on a 64-bit boundary for the use of double-precision variables. Since integers were 32 bits, size was always specified as an even number where an additional integer was added to requests of an odd size. Moreover, if the memory chunk was larger than the maximum size in the array, a linked list of the larger chunks was maintained. These lists (known as 'buckets') of both fixed-sized and large variably sized chunks would be referenced first before moving the free index. In this manner, overall memory usage was reduced.

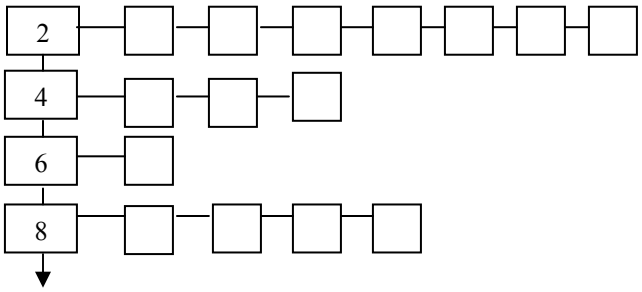

Figure 3: JIMM Memory Buckets

The gpMemJnr memory pool has several advantages.

1. The allocation and subsequent return of memory avoids the system overhead within procedures provided by the operating system. Hence, it operates more quickly.
2. The index of a data structure within the array serves as a unique identifier in cases where the data structure location is the same throughout the simulation.
3. The index also serves as an address. Conversion from the index to a pointer is achieved by adding the value of the index to the address of the first element of the array.
4. Lastly, the state of the simulation is easily saved (or "checkpointed") to a single file (a.k.a. a "checkpoint") by a single write operation from its beginning of the array to the free index. The use of indices remains the same between successive checkpoints. If the old value of the array addressed is retained, then pointers in the array can be "fixed" or adjusted by the difference between the old value and the new value.

The main disadvantage is that the scenario developer must explicitly state the size of the memory pool. Should additional memory be required, then the simulation would terminate. Hence, the developer had to be sure that sufficient memory was specified and that a contiguous array of that size could be provided by the underlying operating system.

2.1 Checkpointing

Checkpointing is a critical component of JIMM operation (Lattimore et al. 2005). Though it can occur during an execution, it more commonly occurs at the end of an execution. A checkpoint could be used in case of error recovery. However, it is more often used to allow a simulation to proceed past its initial stage. Once the checkpoint (also known as a "big bang" in Lattimore et al. 2005) is taken, an analyst could execute multiple runs from that point or

modify the simulation as required using the checkpointed state of the simulation as a baseline.

More importantly, since checkpointing is also done at the end of a JIMM execution, it also allows simulation construction to proceed in steps where each step builds on the results of the previous step. Normally, JIMM scenarios are constructed and executed using nine distinct steps. The transition from one step to the next is provided via the checkpoint.

The nine steps of JIMM are provided in Table 1 in the order they are normally executed.

Table 1: The Nine Steps of JIMM

Step Name	Acronym	Purpose and Comments
Language Data Base	LDB	Sets up the JIMM Conflict Language (JCL) for the following steps
Icon Data Base	IDB	Sets up the icons and colors for the JIMM graphics display. This is skipped when no graphics are used.
Ground Data Base	GDB	Translates Digital Terrain Elevation Data (DTED) data for use by the EDB. This step is normally not checkpointed.
Environment Data Base	EDB	Takes terrain output from the GDB step and provides a "terrain skin". This skin is kept in a separate file and this step is also not commonly checkpointed.
Type Data Base	TDB	Develops specific player (simulation object) types. Includes characteristics and the tactics the player types employ.
Scenario Data Base	SDB	Specifies specific instances of player types and their laydown.
Run Data Base	RDB	Actual execution of the simulation
Configuration Data Base	CDB	Actual execution of the simulation with the addition of instructions for integrated operation with SWEDAT.
Analysis Data Base	ADB	Post-processing of simulation runs such as filtering of captured data, counts of events for analysis etc.

The use of checkpointing within the nine steps is diagrammed in Figure 4 with solid arrows indicating transitions via checkpoints.

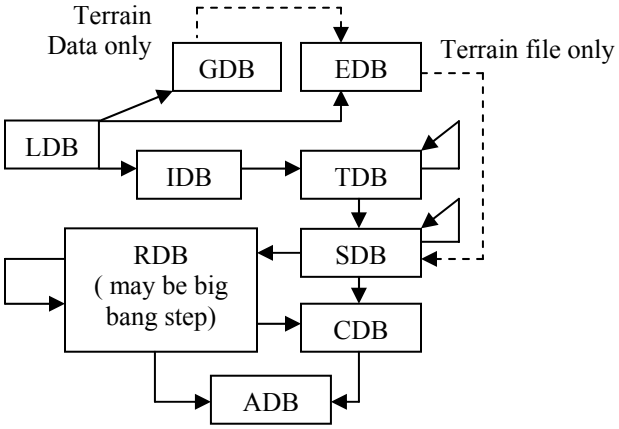

Figure 4: Checkpointing in the Nine Steps of JIMM

2.2 Temporary Memory

Initially, JIMM also extended the memory pool to allow "temporary" memory (Lattimore et al. 2005). Temporary memory had its own free index and was allocated from the end of the array as opposed to the beginning. In this manner it could be referenced and used in the same manner as "permanent" memory. However, it would not be saved whenever a checkpoint was taken.

Temporary Memory was commonly used for simulator graphics. It was also used is cases when permanence between checkpoints was not anticipated.

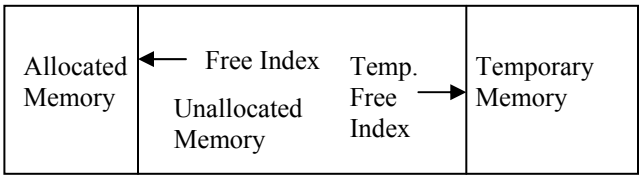

Figure 5: Memory Pool with Temporary Memory

Temporary memory also had its drawbacks.

1. Programmers sometimes used temporary memory for data that could be required to persist between successive events. Since a checkpoint could occur between these events, the data would be lost. This was a common source of error.
2. Since the memory operation was common, the small amount of overhead associated with differ-

entiating permanent and temporary memory was still significant and reduced simulation execution speed.

In the initial phases of the multi-threading effort, it was determined that the temporary memory mechanism should be removed and "permanent" memory used instead. In cases where temporary memory was useful, memory recollection procedures were employed. The capabilities that temporary memory possessed were also later provided by use of the multi-threaded memory pool.

3 PROTECTING THE POOL WITH MUTUAL EXCLUSION

When the effort for multi-threading in JIMM was started, internal operation of the memory pool had to be protected. The first approach was to employ a pthreads "mutex" variable. This variable ensured mutual exclusion in that one and only one thread could operate within a critical region of code. Unfortunately, given the large number of memory operations, the overhead associated with the mutex was very high. Initial timing studies (later confirmed by the author) showed an increase of nearly 16% in execution time. Given the desire to improve performance through parallelism, this drop was deemed to be too great.

4 THE MULTI-THREADED MEMORY POOL

After it was determined that the single memory pool could not effectively protected, the use of separate memories was explored. Each of the memory managers would have a separate set of buckets and would also have a separate store of memory from which to allocate.

After some analysis, the following requirements for these memories were determined.

1. Indices and Pointers from one memory should be usable by other memories.
2. Overhead should not be high.
3. Memory use should be reasonably efficient.
4. Checkpointing should still occur in a single write operation.
5. Memories should be able to combine with other memories.
6. If a memory does not contain references to data in other memories, then it can be deleted easily.

The mechanism developed was based on a two-tiered approach. First, the array was divided into large fixed-size chunks controlled by a single manager known as 'TJNRmemory'. The control of memory within the single TJNRmemory is protected by a mutex variable.

In turn, each thread has its own memory manager (known as a 'TJMemory'). Whenever a new TJMemory manager is created, it obtains a chunk from TJNRmemory. It then allocates and returns memory from this chunk as needed. Since this access is only from a single thread, there is no need for mutex protection. In a similar manner, if additional memory is needed, it obtains an additional chunk from TJNRmemory. Fortunately, most memory operations do not require acquisition of additional chunks. Thereby, the need to access a mutex variable is significantly reduced.

In effect, the need for mutex protection is removed from the immediate thread memory manager and moved to the TJNRmemory. Hence, the protection is needed significantly less often and overhead is reduced. Initial performance studies showed that the additional overhead was negligible.

The construction also satisfies the other requirements. The use of a single array in TJNRmemory means that array indices employed by the different memories would be interpreted in the same way since the offset is from the beginning of the overall array. Hence, an index of memory created from one thread memory manager could be properly referenced by another thread. In addition, memory allocated from one manager could be added to the buckets of another without difficulty.

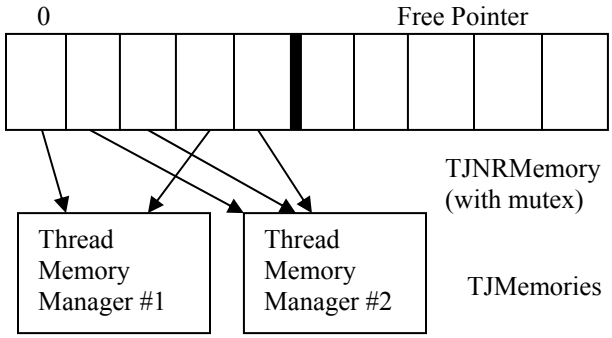

Figure 6: Multiple Thread Memories

Moreover, if the chunks were reasonably sized, then the loss of usable memory through internal fragmentation within the larger chunks would be small. Thus, the use of memory is still reasonably efficient.

Rapid checkpointing is still achieved via the TJNRmemory since it allocates the chunks in order of ascending index. Thereby, the checkpoint still consists of a single file to the chunk last allocated.

Thread memories can merge by combining their buckets and their chunks. The large fragment at the end of the

chunk being processed and merged is added to the buckets. If the chunk is large, it is added to the variable list. Moreover, allocation of memory is modified to look at larger chunks in the buckets before requested memory from the TJNRmemory manager.

Lastly, if thread is isolated such that the memory from its memory manager cannot be placed into another manager's buckets, then the memory can be restored at thread termination by simply returning the chunks back to TJNRmemory control. This mimics the capability previously provided by "temporary" memory.

4.1 Other Memory Types

In some cases such as terrain (constructed in the EDB step) and contour graphics, it was found that memory operation via memory pools was desirable but that there was also no need to intermix the required memory with memory used for the general simulation. Furthermore, the specific amount of memory required could be determined in advance. For these cases, specific instances of memory without a TJNRmemory manager were created. This was implemented using a base class for a memory (TBMemory) and derived classes for the thread memory mangers (TJMemory) and these other more simple managers (TMemory).

5 EXPERIMENTAL WORK

The performance of the initial solution of protecting the memory pool directly and the solution using fixed-sized chunks of memory was confirmed by the author. The timing test result is the average of one hundred (100) runs of JIMM ACE 2.4.1 A29 using the default JIMM "Final Battle Obruty" Scenario (Lattimore et al. 2005). The size of the chuck was set to 16K 32-bit integers. A count of the calls to pthreads "mutex" operations was also taken.

Table 2: Timing Test of Proposed Solutions

Solution	Mutex Call Count	Average Time (100 runs)
No Solution Implemented	0	60.7 sec.
Protecting the memory pool directly (call for every allocation from the pool and return of memory to the pool)	88,989,102	70.4 sec.
Memory chunks (with a call every time a chunk is allocated or returned to its pool)	242	60.7 sec.

The scenario itself is used to provide users with examples of JIMM operation and thus contains a wide variety of simulated activities. The scenario runs for an extended period of simulated time (4.2 hours) and requires more than two million events. The experiment was executed on an 866 MHz IBM PC running the Red Hat 9 Linux Operating System. The code was compiled with the default GNU compiler with optimization (-02) turned on.

The effect of the size of the chunks was also tested. Over the range of the test, the difference in timing was not significant when compared to the solution of protecting the memory pool directly. Figure 7 shows the indirect relationship between the number of mutex calls given the page chunk size.

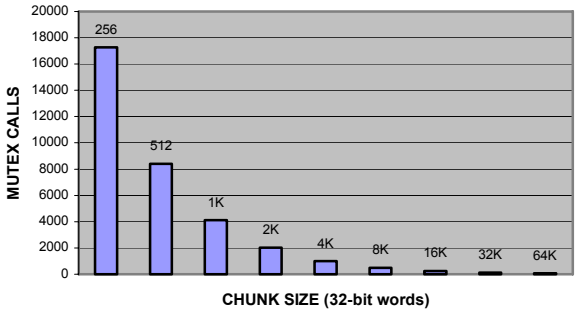

Figure 7: Relation of Chunk Size to Mutex Calls

In JIMM ACE 5.0 (Lattimore et al. 2006), the size of the memory chunk is set to 16K 32-bit bytes.

6 EXPANDING THE MEMORY POOL

One of the main shortcomings of memory pools is the requirement for a single contiguous allocation of memory. However, once the implementation of multiple thread managers was tested and proven, it was noted that the fixed-size chunks in TJNRmemory are similar to pages (or frames) as commonly used by operating systems. Therefore, if a mechanism akin to a page table was added to the memory managers, then the chunks would not need to be part of a contiguous array and additional chunks could be obtained from the operating system should additional memory be required (Kitchen, 2005).

In work done by Mr. Blair Kitchen, each chunk of allocated memory was referenced in a thread memories page table. The operations where indices and pointers were converted back and forth were modified to use these page tables. Conversion from an index to a pointer was handled by a single table lookup. However, the reverse conversion required a search of the table. This significantly increased overhead.

Checkpointing was achieved by writing each of the pages (chunks) to the file in order of initial indices. This would restore the contiguous nature of the memory manager. Furthermore, the old page table was retained for conversion of pointers should the checkpoint be utilized.

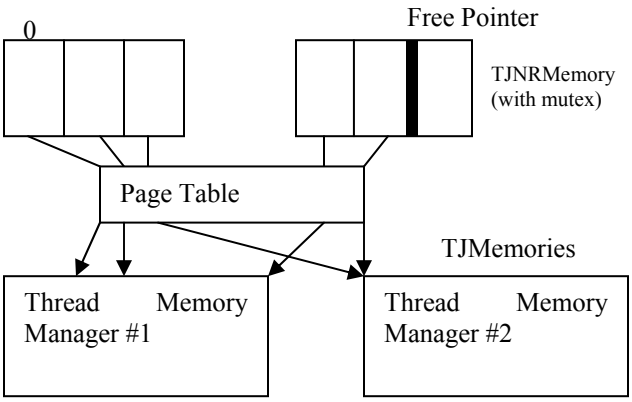

Figure 8: Thread Memories without a Contiguous Underlying Array

This work with page tables was done as a prototype. It showed that a paging scheme could be implemented and still maintain correct operation of the model. However, the overhead in use of a single page table as well as the overhead of the page lookup was deemed to be too great to implement in its current form (Kitchen 2005).

Anecdotal evidence suggests that as modification of JIMM progresses, there will be less explicit use of indices for memory addresses. Hence, the use of tables can be explored at a later case since greater source of overhead will be reduced. Moreover, the use of multiple page tables to relieve contention can also be explored to further improve efficiency.

7 CONCLUSION

This paper has described the successful implementation of multiple thread-specific memory managers as a solution to multi-threading given a common memory pool. The additional cost was significantly lower than the cost of the initial implementation using a single mutex variable.

The use of smaller memory chunks leads to the employment of a mechanism similar to page tables. This eliminates the constraint given the limits of contiguous in a similar and the need for a scenario programmer to specify the size of memory pool. The implementation was shown to be correct. However, further efficiencies are currently needed for integration into JIMM operation.

ACKNOWLEDGMENTS

The acknowledged creator of the SWEG and JIMM family of models is Peter Lattimore. The initial implementation of the memory pool was conducted under his guidance and leadership.

The work to initially create the thread memory managers was done as part of project FMS-7 from the High Performance Computing Modernization Program Office. The CTA leader was Dr. Larry Peterson. Dr. Michael Chapman and Ralph Gibson were instrumental in testing the results and showing correctness. William E. Brooks provided the initial implementation with mutual exclusion and performed that initial timing study. Other team members include Jon Anderson, Stuart Baldwin, Ronald Chesley, Doug Pickeral and Jonathan Smith. The work for tables of non-contiguous memory was done in 2004 by Blair Kitchen after completion of the parallelization project.

This work has been cleared for open publication by the Naval Air Systems Command (NAVAIR) Public Affairs Office (PAO) as NAVAIR Public Release 06-0079, Distribution Statement A – "Approved for public release; distribution is unlimited".

The multi-threaded version of JIMM is known as JIMM ACE. All versions of JIMM are currently managed by the JIMM Model Management Office (JMMO). Current members of the JMMO include Natasha Bailey, Summer Brandt, David Cassidy, Michael Chapman, Ronald Chesley, Jeffrey Fischer, Ralph Gibson, and Maritza Miller. The JIMM Model Manager and head of the JMMO is Gordon Long. The JMMO can be contacted via e-mail at <jmmo@navy.mil>.

REFERENCES

Bulka, D. and D. Mayhew. 2000. *Efficient C++ -- Performance Programming Techniques*. Addison Wesley, Boston Mass.

Duquette, M., J. Nalepka, and R. Luczak. 2004. The enhanced generic air defense system. *AIAA Modeling and Simulation Technologies Conference and Exhibit* AIAA-2004-4799. Providence RI, Aug 16-19.

Hoagland, D., E. Martin, and M. Anesgart. 2001. Representing goal-oriented human performance in constructive simulations: validation of a model performing complex time-critical-target missions. *Proceedings from the Spring 2001 Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. San Diego CA. Paper Number 01S-SIW-137.

Kelly, M., S. Vick, J. Schloman, and F. Zawada. 2004. A weather service for introducing dynamic attenuation

factors in the joint integrated mission model (JIMM). *Proceedings from the Simulation Interoperability Workshop*. Simulation Interoperability Standards Organization. 04F-SIW-107, Fall.

Kitchen, B. 2005. Eliminating memory constraints in JIMM. *JIMM Users Group, May 2005*. JIMM Model Management Office, Patuxent River MD 2005. Available via the JMMO at <jmmo@navy.mil>.

Lattimore, P. et al. 2005. SWEG 6.5.5 source code and user guides. JIMM Model Management Office. Patuxent River MD 2005. Available via the JMMO at <jmmo@navy.mil>.

Lattimore, P. et al. 2005. JIMM 2.4.1 volume I users guide. JIMM Model Management Office. Patuxent River MD 2005. Available via the JMMO at <jmmo@navy.mil>.

Lattimore, P. et al. 2005. JIMM ACE 5.0 source code. JIMM Model Management Office. Patuxent River MD 2005. Available via the JMMO at <jmmo@navy.mil>.

Michelletti, M.L. 2003. "FMS-7 JIMM ACE beta test review JIMM ACE 2.4.1_A529". DoD High Performance Computing Modernization Program Office (HPCMPO). 31 July. Available via the JMMO at <jmmo@navy.mil>.

Mutschler, D.W. 2005. Parallelization of the joint integrated mission model (JIMM) using cautious optimistic control. *Proceedings of the 2005 Summer Computer Simulation Conference*. Society for Modeling and Simulation International, July, pg. 145-152.

Mutschler, D.W. 2005. Language-based simulation, flexibility and development speed in the joint integrated mission model. *Proceedings of the 2005 Winter Simulation Conference*. Orlando FL, December 2005

Mutschler, D.W. 2005. Improved integrated operation in the joint integrated mission model (JIMM) and the simulated warfare environment data transfer (SWEDAT) protocol". *ITEA Modeling and Simulation Conference*, Las Cruces NM, December.

Nalepka, J.P. 2000. JIMM: the next step for mission level Simulation models. *AIAA Modeling and Simulation Technologies Conference*. AIAA 2000-4491, AIAA, Washington D.C.

Niland, W., B. Skolnik, S. Rasmussen, K. Finle, and K. Allen. 2005. Enhancing a collaborative UAV mission simulation using JIMM and the HLA. *Proceedings of the Spring 2005 Simulation Interoperability Workshop*, Simulation Interoperability Standards Organization, San Diego CA, Spring.

Worsham, R. 2002. Northrop Grumman radar simulation (AVSIM). *Proceedings of the 2002 IEEE Radar Conference*. April. pg 176-186.

AUTHOR BIOGRAPHY

DAVID MUTSCHLER obtained his Ph.D. in Computer and Information Sciences from Temple University in 1998. He has been employed by the Naval Air Systems Command (NAVAIR) since 1985 working for ten years at Warminster PA and the remainder at Patuxent River, MD. He has served as the principal investigator of the project "Parallelization of the Joint Integrated Mission Model (JIMM) Using Cautious Optimistic Control (COC)" and as the JIMM Model Manager and head of the JIMM Model Management Office (JMMO). He is also an Associate Professor at the Florida Institute of Technology School University College. He is a member of Association for Computing Machinery (ACM) and its Special Interest Group in Simulation (ACM/SIGSIM) and the Institute of Electrical and Electronics Engineers Computer Society (IEEE, IEEE/CS). His research interests include modeling and simulation, parallel discrete event simulation, and software engineering. His e-mail address is <david.mutschler@navy.mil>.