ADST-II-CDRL-MODARC-9800135A
1 September 1999

ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY II (ADST II)

MODSAF ARCHITECTURE

(DO #0066)

CDRL AB01

SCIENTIFIC AND TECHNICAL REPORTS

FINAL

MODSAF RE-ARCHITECTURE REPORT II

For:

United States Army
Simulation, Training, and Instrumentation Command
12350 Research Parkway
Orlando, Florida 32826-3224

By:

Science Applications International Corporation
12479 Research Parkway
Orlando, FL 32826-3248

Lockheed Martin
Information Systems Company
12506 Lake Underhill Road
Orlando, FL 32825

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1 SEP 1999	final

4. TITLE AND SUBTITLE
Advanced Distributed Simulation Technology II (ADST-II)
Final MODSAF Re-Architecture Report II

5. FUNDING NUMBERS
N61339-96-D-0002

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Lockheed Martin Information Systems
ADST-II
P.O. Box 780217
Orlando Fl 32878-0217

8. PERFORMING ORGANIZATION REPORT NUMBER
ADST-II-CDRL-MODARCH-9800135

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

NAWCTSD/STRICOM
12350 Research Parkway
Orlando, FL 32328-3224

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The purpose of this report is to document the ADST II ModSAF Re-Architecture modifications and lessons-learned from the analysis conducted to facilitate the use and development of the ModSAF system. The tasking under this Delivery Order (DO) is a follow-on effort to the ModSAF Program DO #0009, specifically in the area of architecture. The efforts in this DO continue to expand on the findings of the Re-Architecture Report from DO #0009. These efforts focused on reducing maintenance costs and improving usability, and increasing general capabilities by architectural enhancements.

14. SUBJECT TERMS	15. NUMBER OF PAGES
STRICOM, ADST-II, MODSAF, simulation, , Re-Architecture	18
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Document Control Information

Revision	Revision History	Date
	Preliminary	05/10/98
A	Final	09/01/99

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this report is to document the ADST II ModSAF Re-Architecture modifications and lessons-learned from the analysis conducted to facilitate the use and development of the ModSAF system. The tasking under this Delivery Order (DO) is a follow-on effort to the ModSAF Program DO #0009, specifically in the area of architecture. The efforts in this DO continue to expand on the findings of the Re-Architecture Report from DO #0009. These efforts focused on reducing maintenance costs and improving usability, and increasing general capabilities by architectural enhancements.

1.2 Technical Overview

Architectural enhancements to be addressed under this DO include:

- Single Processor Threading Execution
- Memory Analysis Capability
- Dynamically Linking ModSAF Modules
- FSM Compiler Enhancements
- Time Management
- Digital Message Interface

These enhancements and their efficacy within ModSAF system are described in the following paragraphs.

2. Applicable Documents

2.1 Government

ADST II Statement of Work (SOW) for ModSAF Architecture Delivery Order, AMSTI-97-W087, 9 September 1997.

2.2 Non-Government

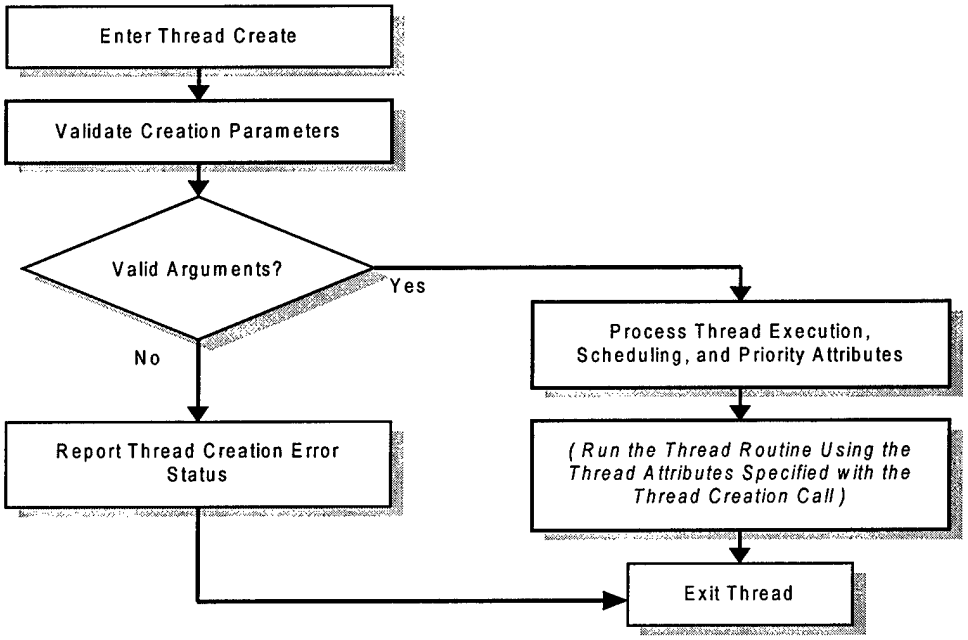
ModSAF Re-Architecture Report, ADST-II-CDRL-011R-9600207, 28 June 1996.

3. ModSAF Re-Architecture Study Results

3.1 Single Processor Threading Execution

In general, it is believed using a multithreaded execution model would significantly increase overall performance of the ModSAF runtime, even on a single processor based system. Separate parts of the program could be divided into individual threads, each performing its clearly defined task in concert with the others. Processor time then could be put to better use by dispatching these threads when processor time is available. The context of a thread is much less than that of a process, and data is automatically shared across sibling threads. The kernel's scheduling algorithms are clearly a better choice to support multithreaded execution because of their efficiency and portability. Useful work could be performed by the CPU, instead of waiting during expensive delay loops, or for alarm signals to perform task switching functions. The complexity of design and code for new tasks would not be as dependent on overall program execution flow, providing less development impact for new DOs.

Threads in the ModSAF program were created indirectly through the development of the libsafthrd API interface. When a thread object is created, it is spawned by a call to the **safthrd_create** library operation. This service creates the thread by configuring the thread's execution attributes and using the underlying system's capabilities to begin the new thread's execution. The following figure shows the flow of this operation. Allowing the **safthrd_create** call to establish thread attributes and initiate the new thread abstracts thread creation away from the calling environment.



Thread execution attributes are made available to the initiator through enumerated types defined in the libsafthrd library. Other values, such as thread priority, are passed to the creation operation and tested for compliance with the capabilities of the underlying implementation. Should these values be invalid, an error is reported. If the thread attributes are valid, they are used to modify the execution characteristics of the thread prior to calling the target routine. The thread function pointer that is provided in the input to the creation operation is used to run the target routine in the context of the new thread. Parameters for this target function are also passed to the routine using a parameter list pointer. Any resource allocation needed to create the thread will be made by the libsafthrd library through the underlying architecture. If the target function is joinable and provides return status, this status will be propagated back to a joining thread when this thread exits. If the thread is created as a detached or server thread, it never has to exit and returns no status.

3.1.1 Conclusions Concerning the Threaded Socket Interface

The benchmark test results demonstrate the implementation of the threaded socket interface is functionally operational for packets sent to, and received from the same workstation. While the throughput increase was not as dramatic as expected, it was at least as good as the non-threaded version. With a single processor platform, this lack of extensive improvement is not unreasonable. The normal operation on network packets in the benchmark is to send a packet onto the network, then read a packet back. In either the non-threaded version, or the threaded version this sequence requires the nearly same amount of functional overhead to be expended.

The basic socket interface is now using a blocked socket, instead of an unblocked socket connection. This allows the socket connection to synchronize data transfers through it. The non-threaded version needed to make adjustments in the application code to achieve the same result. When packet transactions are performed to the network, most of the conditions bypassing the application overhead will be satisfied, resulting in more efficient execution.

The socket thread is performing much of the low-level functions and absorbing the system interface overhead associated with the socket read calls. When this threaded version is ported to a multiprocessor system, the socket thread overhead can be further segregated from the main application by separating the executions of the two threads onto separate CPUs. Because the main thread has only the overhead of buffer reads, and is not severely impacted by the high overhead socket read calls, the network PDU packet reads will less impact it. Because the packet interface buffer between the threads can be adjusted in size, the efficiency of the buffer interface may be tuned for the best performance.

The tests that capture the throughput of packets between the workstations represent closer to the normal network packet activity in ModSAF during general use than the activity in the benchmark tests. These results again show a minimal gain in packet throughput for a low numbers of packets read, but they also show an increase percentage gain when the number of PDU packets transacted increases. Again, many of the observations and conclusions stated

for the benchmark tests can be applied to the explanation of gains when packets are read from external stations.

Two main conclusions may be stated from the results of the network packet tests using the Blaster program:

A. The threaded version of ModSAF functionally operates as expected from the intent of the design.

B. The threaded version of ModSAF functionally performs equally as well as the non-threaded version, even when high numbers of packets-per-second are being processed.

3.1.2 Combination Testing Results

The test runs executed as a combination of the benchmark and network packet tests provides an even greater opportunity to test the packet throughput capability for the threaded version of ModSAF. These tests show a significant improvement in packet throughput over the comparable non-threaded version of the program. These performance gains most likely are a result of the operation of the main thread in this type of test execution mode. The combination of packet transactions from within the same station and also from remote stations provides a greater availability of packets to process, and therefore a greater number of packets in the buffer, for the main thread to process. The socket thread can fill the packet buffer when it gets time to run and spends less time waiting for packets from the network as these packets come from both the local station and other stations. While the performance of throughput of 91.7% for the first test may be exceptional, the improvement is still significant as verified by the 36.6% increase from the second test. This type of improvement gives credibility to the assumption that multiple processor platforms will provide significant performance improvement with the threaded version of ModSAF.

3.1.3 General Conclusions Regarding Threading of ModSAF Capabilities

In the process of creating the thread API library libsafthrd and converting the socket read capability in libpktvalve several issues were addressed to accomplish these goals:

A. First, libsafthrd is ultimately intended to provide a generic threads API for the ModSAF program. Other parts of ModSAF will be easily able to create and use threads on any platform where ModSAF runs. The support on various platforms is not so generic and is not supported as consistently as would be desired. The current implementation of libsafthrd uses a minimal POSIX pthreads interface and consequently will function properly on any platform that supports a pthreads API. While POSIX pthreads is defined as a standard, it is not implemented in a similar way across all platforms supporting it. Libsafthrd uses a minimal subset of pthread functionality and may not realize any detrimental effects from this, as long as the minimal requirements for libsafthrd operations are met.

B. Second, the implementation of the read socket interface was more problematic at the level of implementation than first thought during design of the interface. Several issues about how much functionality in libpktvalve should be put into the thread, or left in the main thread were complicated by several sub-functions being performed in libpktvalve by the read routines. These sub-functions include the unbundling of packets, the filtering of PDU packets, and the dispatching of PDU packets. The reduced risk of the lower layer of implementation allowed the socket thread to divorce itself of these issues by passing the packets to the main thread were these operations could be performed as needed. The downside of this implementation was the inability to take advantage of threading the socket interface at a higher implementation level and separating more functionality into the socket interface thread. The risk at this level is greater, but the benefits could also be greater if successful.

C. The third issue is in regards to the general threading of ModSAF functionality. Conceptually, threading of an application requires several characteristics be present to accomplish performance improvements or even the basic ability to thread the application. These characteristics include the following:

    1. A task must be identifiable as an independent functional module. Does it use common resources of the other parts of the program? Does it depend extensively on other parts of the program? Can it execute separately and minimally effect other parts of the program?

    2. The task must be able to run when necessary to accomplish its duty. Does the task become blocked extensively waiting for a resource? Does the task need to interface often with other parts of the program? Can the task tolerate either of these conditions and still perform its job effectively?

    3. The task must be able to interface asynchronously with the other parts of the program. Are the objects affected by the task used by many other operations within the program? Are the objects affected by the task universally available to all tasks using them? Do the objects affected by the task change often? Are the objects affected by the task critical to the operation of the program? Will synchronization of these objects adversely affect program performance?

    4. The relative importance of the task must be controlled. Can the priority or scheduling of a task be set relative to the other work occurring in the program? Will simultaneous execution severely impact overall program performance?

ModSAF has many operations that do not conform to some or all of these basic requirements allowing them to be easily converted into a threaded implementation. The liberal use of dynamically created objects complicates the ability to thread and synchronize many tasks. The multifunctional nature of many operations makes the separation of these tasks into threads more problematic. Although the results of the threading of the socket read task were completed successfully, the implementation was not straightforward due to the complexities

of the code and may be more complicated in other areas of the program. Before the threading of a particular operation is attempted, the conformance to the requirements listed above needs to be carefully evaluated.

3.2 Memory Analysis Capability

The memory management task resulted in the development and testing of a general purpose library that provides benefits to applications like ModSAF.

This library started out as MemBoy and is now called HeapMan. HeapMan provides specialized allocation pools on top of the standard memory allocation mechanisms available in ANSI-C (i.e. malloc, realloc, free). These specialized pools can be used to take advantage of specific patterns of memory allocation and deallocation that take place in ModSAF.

HeapMan provides three types of pools:

- Fixed-size block: Useful for code that needs to manage many objects that are the same size.
- Constant: Useful for code that allocates lots of memory without the intent of ever deallocating it.
- List: A pool of objects that have built in doubly-linked list functionality.

These pools can be stacked. For example, a list pool can be built from fixed size objects that are in turn permanent objects.

HeapMan also provides hook functions that can be used to insert debugging code at any point in the allocation process. This is useful for tracking bugs.

This library was linked into an experimental version of the ModSAF source code. A few libraries were picked to upgrade to it based on their heavy use of memory allocation and their use of memory in specific ways that is covered by HeapMan.

Testing was done to determine the impact of these changes. The testing showed that just by changing a few libraries, approximately 6% less memory was consumed by the SAF for a benchmark run. This represented 2.4 megabytes of the approximately 40 megabytes consumed total. Performance was improved by just over 1%.

This code is clearly worthy of integration because it improves both performance and memory consumption. It has not been integrated so far because it will have an impact on many libraries in the system.

3.3 Dynamically Linking ModSAF Modules

The purpose of the dynamically loadable modules (DLM) task is to make the SAF more modular at run-time. This means making some functionality loadable at run-time upon demand instead of requiring it to be always available at link time.

Examples of the kind of functionality that can be loaded at run-time include the behavior and physical models. Ideally, these models would be loaded at run-time upon the creation of a unit that uses those models. This way, changes to these models have less impact on the rest of the system at build time.

This task implemented the necessary infrastructure to dynamically load behavior modules. This includes the code necessary to remove the compile-time dependencies that behavior libraries have on each other in the form of a resource database that can be used to share data. Also, many changes were made to the build process to make it possible to specify which behaviors were statically linked into the monolithic executable and which ones are dynamically loaded.

For this task, one behavior library, urendevous, was updated to take advantage of the new architecture. This library was chosen because of its small number of dependencies on other behaviors. The rest of the behaviors will take more work to flush out the legacy dependencies.

3.4 FSM Compiler Enhancements

ModSAF behaviors are implemented as a variant of traditional Finite State Machines (FSMs). ModSAF maintains a special C extension language for developers to define behaviors. The FSM compiler, fsm2ch, converts this language into C code. The C extension language describes the structure of the behavior and provides the syntax on the mechanics of the behavior (e.g., how to transition between states, how to handle events). This language is augmented by C code to fill in the details of the behavior.

This behavior language benefits developers by automatically generating interfaces to the task manager and the rest of the system (thus ensuring consistent interfaces), automatically filling in the details of the task transitions, and establishing event handlers parameter changes and the behavior getting its normal time to run (tick). This auto-generation of code frees developers from writing largely duplicate code to handle the mechanics of the SAF's FSMs and reduces maintenance costs since the code is generated.

In the existing (platoon and above) behaviors, the unit executing a behavior often checks to see if their subordinates have changed (e.g., been deleted, destroyed, mobility impaired) and if so, react accordingly (e.g., shuffle their formation, call off the attack). Unfortunately there was no support in the FSM language to help with this, and so behavior writers have all

established their own methodology and code to handle these cases. This has made the behaviors unnecessarily difficult to read for SAF developers, and created a bevy of duplicated code.

To remove the need for developers of future behaviors to create their own checks to determine if their subordinates have changed, an optional new event handler was added to the FSM language called subord_changed. As part of this effort, modifications were also made to the task manager, which controls behavior execution on entity. Before these modifications, the task manager would invoke a behavior's parameter change handler if its parameters had been altered since it last ran, and if its time to tick had arrived. The task manager was modified to now check if an entity's subordinates had changed since the last time the task manager ran this a given behavior, and if so, it would invoke the behavior's subord_changed event handler, if one was defined.

The advantages of these modifications were then demonstrated in an existing behavior, urwaassemble. A snippet of code from a state in the original behavior used to look like:

```
`
assembling
`                tick
{
    int32 i, all_done;
    UUTIL_NUM_SUBS   num_subs;
    /*
     * If the number of subordinates has changed
     * then reassign the tasks.
     */
    num_subs =
      unitutil_subordinates_changed(priv->n_subordinates,
      priv->subord_id,
      unit_entry,
      SAFCapabilityMobility);
    if (num_subs != UUTIL_NUM_SUBS_SAME)
      ^do_stop; kids have changed

    /* Check to see if all the vehicles have arrived
     * at their assigned positions:
     */
    for (all_done = TRUE, i = 0; i < priv->n_subordinates;
      i++)
    {
        if (!vrmove_done(priv->po_db,
          &state->vrmove_tasks[i]))
        all_done = FALSE;
    }
    if (all_done)
```

```
^in_assembly;
}
`                        params
{
urass_update(priv, state, unit_entry,
parameters, vehicle_id);
^do_stop;
}
```

Became:

```
assembling
`               tick
                {
                int32 i, all_done;

                /* Check to see if all the vehicles have arrived
                 * at their assigned positions:
                 */
                for (all_done = TRUE, i = 0; i < num_subordinates;
                   i++)
                {
                  if (!vrmove_done(priv->po_db,
                                &state->vrmove_tasks[i]))
                    all_done = FALSE;
                }
                if (all_done)
                  ^in_assembly;
            }
`                       params
                        {
                      urass_update(priv, subordinates, num_subordinates,
                                         state, unit_entry,
                                parameters, vehicle_id);
                      ^do_stop;
                        }
`   subord
{
    ^do_stop; kids have changed
}
```

Note that the highlighted areas in the original code have been changed the highlighted areas in the modified code. These behaviors perform functionally equivalent.

These modifications will allow more consistent behavior development without adding duplicate, difficult to maintain code, and eases visibility of the logic of the behavior for maintainers.

The FSM compiler enhancements have reduced the amount of code development necessary for future behavior programmers. The subordinate changed event handler now provides a consistent and maintainable mechanism for dealing with subordinate changes. Any changes to the mechanics of how a behavior determines that its subordinates have changed can be made in the task manager, rather than in the individual behaviors that use this approach. This functionality was incorporated into ModSAF 5.0, and developers of future platoon and above behaviors should take advantage of it

3.5 Time Management

This effort was not implemented, as per customer request, as a duplicate effort is being performed by another organization sponsored by STRICOM.

3.6 Digital Message Interface

A Digital Message Interface (DMI) was implemented in order to support Command, Control, Communication, and Computer Interface (C^4I) and digital messaging. This implementation is incorporated into the OneSAF Testbed Baseline, Build B software tree.

The DMI provides the mechanism to send and receive FBCB2 Variable Message Format (VMF) messages contained within Signal PDUs on the Simulation Local Area Network. The DMI utilizes the network communication libraries currently implemented in the SAF. The two main components of the DMI are the Digital Message Manager (DMM) and the Digital Data Manager (DDM).

The DMM interfaces directly with the SAF network communications library in order to process incoming and outgoing digital messages. The DMM utilizes configuration files that describe the message type formats (e.g., VMF), therefore if a message format changes it will not affect the DMM implementation. The DMM maps the message format described in the configuration file with the message data provided by the DDM.

The DDM interfaces with the behavior libraries by providing a mechanism for the libraries to register their pertinent data for outgoing messages, and register for callbacks for significant event information in incoming messages.

The VMF messages implemented within the SAF behaviors utilize the new C4I interface to send and receive messages. The messages implemented are described in the following table.

Message	Predicate Condition	Send
	Receive	Response

Message	Predicate Condition Receive	Send Response
1. Situation Report K05.14	A subordinate unit detected low fuel or ammo, has been resupplied, or has just engaged enemy.	The situation report message will be sent from a subordinate unit to its immediate superior.
	N/A	N/A
2. Spot/Salute Report K04.1	A subordinate unit receives an OTB SAF operator's command to send a spot report, the unit detects enemy contact, engagement, or/and salute events	Subordinate units transmit spot, contact, engagement, and salute events in the Spot/Salute report.
	N/A	N/A
3. Message To Observer K02.14		Transmission of fire mission data to the originator of the Call for Fire.
	N/A	N/A
6. Call for Fire (CFF) / On-Call Fire K02.4 K02.12	N/A	N/A
	The designated artillery unit receives the order.	The artillery unit fires at the proper target locations. Ack req'd.
7. Subsequent Adjust Command K02.22	N/A	N/A
	The designated artillery unit will receive the order.	The artillery unit changes the target locations. Ack req'd

Message	Predicate Condition	Send
	Receive	Response
8. Check Fire K02.	N/A	N/A
	The artillery unit receives the Check Fire message.	The artillery unit halts engagement. After receiving this message the artillery unit may receive an End of Mission message from the FBCB2 unit to halt the mission. Ack req'd

In the DMI implementation, there is Message Managers and Data Managers at the vehicle, platoon, company, and battalion behavior levels to process the SA/C^2 messaging for and between the respective levels, and a utility library to provide dynamically bound interfaces.

A new library, Libdmiutil, provides a set of functions to allow behaviors, the Data Manager, and the Message Manager to communication in a common way. The design uses routines similar to libpduapi to load data files and to allow another library to define which attributes in the data file's fields will be used in the interface.

New libraries, Lib*dmimm, represents the DMI Message Manager library at the vehicle level, libvdmimm; at the unit level, libudmimm; at the company level, libucdmimm; and at the battalion level, libubndmimm.. The basic functional requirements for each of the Message Managers is to provide a generic interface with the respective Data Manager and pass information to and from SAF's network libraries.

New libraries, Lib*dmidm, represents the DMI Data Manager library at the vehicle level, libvdmidm; at the unit level, libudmidm; at the company level, libucdmidm; and at the battalion level, libbndmidm. Each of these libraries contain data structures for communication at the appropriate echelon level.

The Digital Message Interface architectural changes were found to be successful in the transmission and reception of digital messages and will be incorporated into the OneSAF Testbed Baseline Build B which will be released in November 1999.

4. Conclusion

Under this DO, several architectural enhancements were analyzed, implemented, and evaluated for their impact upon reducing maintenance costs, improving usability, and increasing general capabilities of the ModSAF baseline. Some enhancements, such as the with the FSM compiler and Digital Message Interface, provided immediate improvement to the system and were incorporated into a release of the software. Other enhancements, such as

Single Processor Threading and Memory Management require further analysis and development so that their benefits to the system can be fully realized. Dynamically linking ModSAF modules proved to be a considerable benefit and an initial implementation was incorporated into the ModSAF 5.0 baseline, and further implementation is anticipated to be released in the near future.

5. Points of Contact

ADST II ModSAF Architecture Team
Bryan Cole
Project Director
407-306-4062
Bryan.A.Cole@saic.com

Derrick Franceschini
Project Engineer
407-306-4113
Derrick.J.Franceschini@saic.com

Deanna L. Nocera
OTB C^{4}I Lead
407-306-3706
Deanna.L.Nocera@saic.com

James (Gene) McCulley
407-306-2557
James.E.McCulley@saic.com

Barry Savchuk
407-306-5589
savchukb@adstii.com

6. Acronym List

ADST	Advanced Distributed Simulation Technology
DO	Delivery Order
FSM	Finite State Machine
ModSAF	Modular Semi-Automated Forces
SAF	Semi-Automated Forces
SOW	Statement of Work
STRICOM	(US Army) Simulation Training and Instrumentation Command