NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPROVING LIFE CYCLE MANAGEMENT THROUGH SIMULATION AND EFFICIENT DESIGN

by

Alberto A. Garcia

September 2008

Thesis Advisor:	Thomas W. Lucas
Second Reader:	Paul Sanchez

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, 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, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2008	Master's Thesis

4. TITLE AND SUBTITLE Improving Life Cycle Management Through Simulation and Efficient Design	5. FUNDING NUMBERS
6. AUTHOR(S) Alberto A. Garcia	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Headquarters, USMC Installations and Logistics (I&L) 2 Navy Annex, Washington, D.C. 20380-1775	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	A

13. ABSTRACT (maximum 200 words)

Life Cycle Management (LCM) is defined as a decision-making process that takes into consideration the benefits, costs, and risks associated with each action over the full life cycle of a system. Effective LCM requires good forecasting to help determine future requirements for design and development, acquisition, in-service support and sustainment, modernization, and final disposal of a fleet of systems. It is in forecasting that simulation tools play a key role in LCM by helping program managers to gain insights into their supported systems.

The Total Life Cycle Management Assessment Tool (TLCM-AT) is a probabilistic modeling and simulation analysis tool developed to support and improve the USMC's LCM. This powerful tool is capable of performing "what-if" scenario analysis to compare the merits of multiple courses of action (COAs) or policies. Unfortunately, such analytical results are predicated on a set of conditions developed in the model that have little chance of occurring in real life.

This thesis introduces a Java-based application that combines the capabilities of TLCM-AT with the benefits of a sophisticated design of experiments (DOE) to perform in-depth sensitivity analysis of alternatives. A well-developed DOE can simulate real life by modeling a wide range of conditions under which the performance of each COA is measured. Data from this kind of experiment can be used to help in the development and selection of robust COAs and policies.

14. SUBJECT TERMS Life Cycle Management, Life-Cycle Costs, Availability, Data Farming, Design of Experiments, Nearly Orthogonal Latin Hypercube, TLCM-AT, Light Armored Vehicle, Clockwork Solutions, SEED Center, Modeling and Simulation, Partition Analysis	15. NUMBER OF PAGES 116
	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	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

IMPROVING LIFE CYCLE MANAGEMENT THROUGH SIMULATION AND EFFICIENT DESIGN

Alberto A. Garcia
Lieutenant Commander, United States Navy
B.S., Embry-Riddle Aeronautical University, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 2008

Author: Alberto A. Garcia

Approved by: Thomas W. Lucas
Thesis Advisor

Paul Sanchez
Second Reader

James N. Eagle
Chairman, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Life Cycle Management (LCM) is defined as a decision-making process that takes into consideration the benefits, costs, and risks associated with each action over the full life cycle of a system. Effective LCM requires good forecasting to help determine future requirements for design and development, acquisition, in-service support and sustainment, modernization, and final disposal of a fleet of systems. It is in forecasting that simulation tools play a key role in LCM by helping program managers to gain insights into their supported systems.

The Total Life Cycle Management Assessment Tool (TLCM-AT) is a probabilistic modeling and simulation analysis tool developed to support and improve the USMC's LCM. This powerful tool is capable of performing "what-if" scenario analysis to compare the merits of multiple courses of action (COAs) or policies. Unfortunately, such analytical results are predicated on a set of conditions developed in the model that have little chance of occurring in real life.

This thesis introduces a Java-based application that combines the capabilities of TLCM-AT with the benefits of a sophisticated design of experiments (DOE) to perform in-depth sensitivity analysis of alternatives. A well-developed DOE can simulate real life by modeling a wide range of conditions under which the performance of each COA is measured. Data from this kind of experiment can be used to help in the development and selection of robust COAs and policies.

THIS PAGE INTENTIONALLY LEFT BLANK

THESIS DISCLAIMER

The reader is cautioned that the computer programs presented in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logical errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF SYMBOLS, ACRONYMS, AND/OR ABBREVIATIONS

AAO	Approve Acquisition Objective
Ao	Availability
AoH	Achieved Operating Hours
COA	Course of Action
CSV	Comma Separated Value
Deg	Degradation Rate
DoD	Department of Defense
DOE	Design of Experiments
DOF	Depot Overhaul Factor
DSN	Data Source Name
En	Number of Platform Events Due to Failures
GUI	Graphical User Interface
ICAP	I-Level Capacity
IQ	Induction Quantity
JDBC	Java Database Connectivity
JMP	Statistical Software used for analysis
LAV	Light Armored Vehicle
LCM	Life Cycle Management
LRU	Line Replacement Unit
M&S	Modeling and Simulation
MEF	Marine Expeditionary Force
MOE	Measure of Effectiveness
MTBF	Mean Time Between Failures
NOLH	Nearly Orthogonal Latin Hypercube
NPS	Naval Postgraduate School
ODBC	Open Database Connectivity
PM	Program Manager
Pt	Task Performed
R&D	Research and Development
RSquare	Percent of Variability Explained by Regression Model
SBn	New Spare Buys
SecRep	Secondary Repairable
SEED	Simulation Experiments and Efficient Designs
Sh	Number of Shipments between Bases
SMR	Source Maintenance and Recoverability
SQL	Structured Query Language
SRAN	Stock Record Account Number
SRU	Shop Replaceable Unit
ST	Service Time
TLCM	Total Life Cycle Management
TLCM-AT	Total Life Cycle Management Assessment Tool
URL	Uniform Resource Locator
USMC	United States Marine Corps

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I want to thank my wife, Elizabeth, for her patience and support. Your continual understanding made it possible for me to successfully complete my work at NPS. Elizabeth, your support throughout my career has made me the person I am and has allowed me to pursue goals beyond my wildest dreams. I love you.

I want to send an enthusiastic thank you to my advisor, Dr. Tom Lucas, who provided me with the necessary leadership and technical expertise to enable me to complete this work. Also, Colonel Ed Lesnowicz, USMC (Ret.), who kept me focused on the goal at hand. Equally important is Captain David Vaughan, USMC, from USMC I&L, who envisioned this project and supported it so well—thanks!

I extend a special thanks to the Clockwork Solutions' TLCM-AT team: Hugh Saint, Patrick Connally, Peter Figliozzi, and Sean Breed. You provided an impressive level of support for this research.

Lastly, I want to thank Major Brad Young, USMC, for his friendship and assistance during our tour in Monterey.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis seeks to improve the United States Marine Corps' (USMC) Life Cycle Management (LCM) through the development of a computer program that enables the service to exploit the benefits of the Total Life Cycle Management Assessment Tool (TLCM-AT) by combining its functionality with sophisticated design of experiments (DOEs). This summary gives an overview of LCM and TLCM-AT, introduces a Java application created during this effort, describes the methodology used during employment of the application, and provides the resulting conclusions and recommendations. The primary goal of this research is to demonstrate how using the Java application with a well-designed DOE can significantly enhance the value of TLCM-AT to the USMC. The analysis focuses on finding analytical insights that could be useful to decision makers. A secondary goal is to provide a methodology capable of executing closed-loop DOEs for TLCM-AT based analyses.

LCM is defined as a decision-making process that takes into consideration the benefits, costs, and risks associated with each action over the full life cycle of a weapon system. It requires program managers (PMs) to possess a superior holistic understanding of the system being supported. Effective LCM requires good forecasting to help determine future requirements for logistics and system support functions.

TLCM-AT is a probabilistic modeling and simulation (M&S) tool designed to improve LCM throughout the service. It allows PMs to quickly gain insights into the system being modeled. The tool helps PMs to better understand how decisions involving system configuration, operations, logistics, and support could affect a weapon system's effectiveness. It generates outputs representing many system parameters, including availability, mean time between failures, spare stock levels, cost per operating hour, and many others.

TLCM-AT uses Microsoft Access databases to manipulate model inputs and outputs. Users can modify the model by directly accessing its database or they can use the Graphical Users Interface (GUI) to develop what-if scenarios. The complexity of the

database files, coupled with the burdensome nature of the GUI, makes the use of TLCM-AT for more sophisticated DOEs very difficult. A crucial motivation for this work is to overcome that difficulty in order for the Marine Corps to better realize the benefits of TLCM-AT.

To overcome the difficulty of performing TLCM-AT based analyses using DOE techniques, the author introduces a Java-based application capable of combining TLCM-AT functionalities with sophisticated DOE. The application runs in closed-loop form from beginning to end. It is capable of automatically modifying the Access database models used with TLCM-AT, with inputs from a predetermined experimental design, which includes the full specification of input settings and runs to be used during a set of simulation experiments. This allows users to examine a broad range of possibilities with TLCM-AT. Figure ES-1 shows the data generation process employed by the application. Users can take advantage of experimental designs, and combine them with the baseline model to employ the application. The application produces a file containing all output and input data necessary for subsequent analysis.

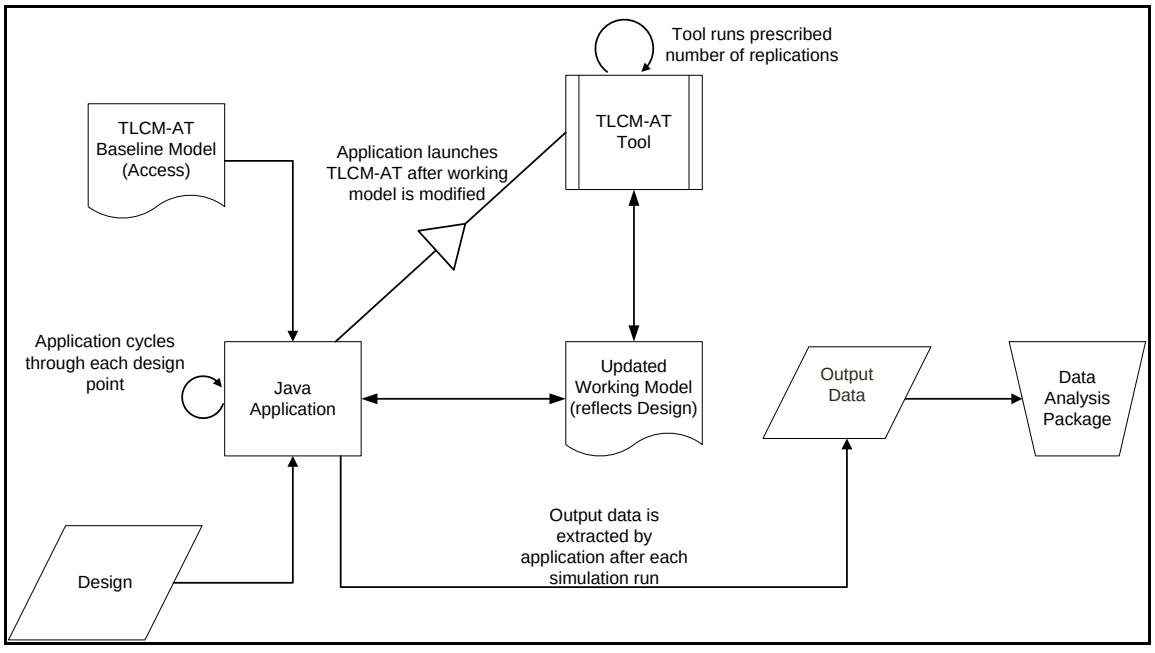

Figure ES-1. Data generation process using Java application

To demonstrate the benefits of using the application, the author performs a comparative analysis of four courses of action (COAs). These COAs are based on a notional scenario in which Marines are deploying two battalions of Light Armored Vehicles (LAVs) to a tropical region. Environmental conditions on the ground significantly affect the performance of two LAV secondary repairables (SecReps). According to the scenario, one LAV-25 battalion will deploy first, followed by the second battalion a few weeks later. Four potential COAs are suggested to help mitigate the anticipated impact of the two faulty SecReps. Table ES-1 describes the COAs analyzed for this study. The author used achieved operating hours (AoH) as the primary measure of effectiveness (MOE) for this thesis.

COA 1	o	Send a large number of spares with the follow-on battalion
COA 2	o	Acquire improved components
	o	Spend $1M on a one month Research and Development (R&D) program
	o	New components cost 2.5 times the cost of legacy components
	o	Deploy fewer improved spares and install them whenever legacy parts are removed
	o	Acquire new parts when legacy parts are condemned—one for one
COA 3	o	A variant of COA 2
	o	Legacy parts are purchased to replace condemned parts
	o	No money is invested in the new components
	o	Goal is to save some money, while maintaining similar level of performance and maintenance usage
Baseline	o	No action taken

Table ES-1. Description of COAs included in comparative analysis

Using TLCM-AT by itself, an analyst could perform four "what-if" scenario simulation experiments, each running for a predetermined number of histories, which corresponds to replications in statistics. The output would consist of the average AoH over every history and its standard deviation. These results are predicated on the accuracy of the data used to develop each model, and say nothing about how sensitive the results are to changing conditions.

Alternatively, using more sophisticated DOEs can significantly alleviate these shortcomings. DOEs allow analysts to complete the same analysis done above, while exploring a larger set of possibilities. For this type of study, analysts vary several parameters simultaneously to simulate a space of possibilities, and to help identify factor

interactions. A space of possibilities could symbolize changes in field conditions and inaccuracies in the data used to build each model. The output from this experiment enables analysts to perform a more in-depth analysis of each COA, help in the identification of factor interactions, and would enable them to determine how sensitive each model is to changing conditions.

The results of the analysis indicate that COA2 is the most robust of the four COAs. After running 129 design points, COA2 outperforms all other alternatives 99 times; even in the designs where COA2 does not have the best performance, the AoH differences among them have no practical significance. Further analysis of the output reveals that the time it takes to repair a failed component has the greatest impact on the AoH result. The insights gained by using DOEs are far superior when compared to simple "what-if" analyses. With this information, maintenance managers can implement reductions in repair times in different ways, including increases in capacity or personnel, improved training, better tools, implementation of lean work habits, etc. Performance thresholds, factor interactions, and significant factors are a few of the insights that can be gained from DOE analysis.

The wealth of information collected from combining TLCM-AT and DOE techniques can significantly improve the forecasting ability of decision makers. It can be used during the development of USMC Approve Acquisition Objectives (AAOs), it can provide insights into a system's interactions, it can assist in the development of robust policies or COAs, etc. Armed with this knowledge, logisticians, maintainers, and program managers can significantly improve a system's LCM, resulting in enhanced reliability, availability, and maintainability. Analysts can compare proposed COAs and could perform sensitivity analysis on each in order to ensure that a robust policy is implemented. Other benefits include cost avoidance by making better logistical decisions and/or rearranging planned maintenance programs, while maintaining—or even improving—system safety, reliability, and readiness.

The author recommends that the tools presented in this thesis be implemented so that the full benefits of TLCM-AT can be realized.

I. INTRODUCTION

A. BACKGROUND AND MOTIVATION

Life Cycle Management (LCM) can be defined as a decision-making process that takes into consideration the benefits, costs, and risks associated with each action over the full life cycle of a system. The LCM approach is applied throughout the life of a system; it bases all programmatic decisions on the anticipated mission-related economic benefits derived over the life of the system. Programmatic decisions include aspects of design and development, acquisition, in-service support and sustainment, modernization, and final disposal.

The Secretary of the Navy defines specific responsibilities for those tasked with supporting LCM efforts. The SECNAVINST 5400.15 series describes the research and development, acquisition, and associated life-cycle management and logistics responsibilities of the Assistant Secretary of the Navy (Research, Development and Acquisition), Program Executive Officers, Direct Reporting Program Managers, Chief of Naval Operations, Commandant of the Marine Corps, and Commanders of the Systems Commands. Having specific guidance like this is very important given that every action taken by a contractor, program manager, or operator could have a significant impact on the reliability and effectiveness of a system, and could greatly affect the ability of our armed forces to perform their missions.

In response to the Secretary of the Navy's directive (SECNAVINST 5400.15), the United States Marine Corps (USMC) acquired a new tool designed to improve their LCM. Designed and developed by Clockwork Solutions, the Total Life Cycle Management Assessment Tool (TLCM-AT) is a probabilistic modeling and simulation analysis tool that uses computer models to represent a fleet of systems. TLCM-AT helps Program Managers to better understand how decisions involving system configuration, operations, logistics, and support could affect a system's effectiveness. The software generates many outputs representing system status, including availability, mean time between failures, spare stock levels, cost per operating hour, and many others. A

major disadvantage of TLCM-AT is that it is only useful for "what-if" scenario analysis. A simulation tool of this kind would be of greater value if it contained the ability to perform more sophisticated experiments, where a greater range of factors could be simultaneously varied. Varying multiple factors at once would allow decision makers to discover relatively quickly any critical interactions between two or more factors that might otherwise take a long time to ascertain. Major Brad Young (2008) performed an exploratory analysis on TLCM-AT where he investigated the interactions between support and maintenance parameters, and their results on availability (Ao). This research is an extension of Young's (2008) work, as it focuses on automating the TLCM-AT data farming[1] process and expands on his exploratory analysis by including multiple measures of effectiveness (MOEs).

TLCM-AT uses Microsoft Access databases to manipulate model inputs and outputs. Users can modify the model by directly accessing its database or they can use the Graphical Users Interface (GUI) to develop what-if scenarios. The complexity of the database files, coupled with the burdensome nature of the GUI, makes the use of TLCM-AT for more sophisticated designs of experiments (DOEs) very difficult. The main motivation for this work is to overcome that difficulty in order for the Marine Corps to better realize the benefits of TLCM-AT.

B. OBJECTIVE

The author's efforts are focused on developing a computer-based application capable of automating data farming functions using TLCM-AT. Having this tool will allow analysts to perform sensitivity analysis of proposed policies, or it can be used to compare different courses of action (COAs); such comparisons can be used during the development and selection of robust policies. Additionally, the author's goal is to create an application that can be used in closed-loop form, capable of executing a well-designed experiment from start to finish, without any human intervention. Furthermore, we

[1] Data Farming is the process of using a high-performance computer or computing grid to run a simulation thousands or millions of times across a large parameter and value space. The result of Data Farming is a "landscape" of output that can be analyzed for trends, anomalies, and insights in multiple parameter dimensions (Wikipedia, 2007).

demonstrate how to use the newly created application by performing a comparative analysis of the four COAs in Young's (2008) analysis; however, this analysis includes a greater number of MOEs.

C. EXPECTED BENEFIT

The first benefit of this research is the creation of an automatic process to manipulate the input model used in TLCM-AT. This automatic process enables the employment of sophisticated DOE functions in order to efficiently data farm the modeling tool. The wealth of information that can be collected from this process can be made available to LCM leaders. They can then use the insights provided by the DOE to make the best possible decisions pertaining to LCM policies. Armed with this knowledge, logisticians, maintainers, and program managers can significantly improve a system's LCM, resulting in enhanced reliability, availability, and maintainability. Analysts can compare proposed courses of action and could perform sensitivity analysis on each in order to ensure that a robust policy is implemented. Other benefits include cost avoidance by making better logistical decisions and/or rearranging planned maintenance programs, while maintaining—or even improving—system safety, reliability, and readiness.

The newly created application is capable of automatically manipulating Access databases. This is particularly important in the current environment, where simulation tools are often produced by private organizations, many of which choose to use databases to handle inputs and/or outputs. The problem is that databases are not well suited for easy integration with many DOE environments. Consequently, this research seeks to make the application easy enough to modify so that anyone with more than a basic knowledge of Java and Structured Query Language (SQL) can use it to manipulate other databases.

D. APPLICATION FUNCTIONALITY

Chapter IV demonstrates the capabilities of the application created for this research. During the demonstration, the application uses TLCM-AT to analyze four

COAs included in Young's (2008) analysis. The application's current design is capable of collecting output data required to analyze seven MOEs. The seven MOEs collected during the demonstration are:

- Availability (Ao) – Systems availability percentage for a period of time
- Achieved Operating Hours (AoH) – Achieved operating hours
- Events (En) – Number of platform events due to failures
- New Spare Buys (SBn) – Number of new spare buys
- Shipments (Sh) – Number of shipments between bases
- Mean Time Between Failures (MTBF) – Ratio between total achieved operating hours and platform events due to failures
- Task Performed (Pt) – Number of tasks performed by all levels

This thesis includes six chapters. Chapter II introduces readers to the concepts of LCM. It discusses the principles of discrete-event stochastic models and how they compare to deterministic models. The chapter also includes a broad description of the TLCM-AT tool, coupled with a simple overview of the merits of DOE. Chapter II closes by presenting the work performed by Young (2008) and describes how this research relates to it. Chapter III summarizes the process involved in creating the prototype application. This chapter introduces the intricacies of the TLCM-AT database and how it played a key role in the need to develop the prototype application. Discussions include a presentation of how data should be generated, processed, and handled within the Java application, along with an introduction to SQL and the Java implementation. Chapter IV describes the scenarios used and how DOE was applied to the scenarios. It also covers the simulation runs and the format of the output data. Chapter V presents all the data analysis, and Chapter VI summarizes the conclusions.

II. BACKGROUND AND RELATED WORK

A. LIFE CYCLE MANAGEMENT (LCM)

Secretary of the Navy Instruction 5400.15 series defines LCM as "A management process, applied throughout the life of a system that bases all programmatic decisions on the anticipated mission-related economic benefits derived over the life of the system. This encompasses the acquisition program, in-service support and sustainment, modernization, and final disposal." According to David Sykes, General Manager of IXL Metal Castings, "Life cycle management means best practice. It doesn't cost to do it; it costs not to do it" (EPA Victoria, 2006). Effective LCM requires good forecasting to help determine future requirements for logistics and system support functions.

It is in forecasting that simulation tools play a key role in LCM by helping program managers to gain insights into their supported systems. In fact, Department of Defense (DoD) leadership believes so strongly in the value that modeling and simulation (M&S) brings to LCM that DoD instruction 5000.2 series requires program managers (PMs) to include M&S throughout the acquisition life cycle. The instruction directs reporting PMs to identify and fund required M&S resources early in the life cycle in order to gain insights and a better understanding of the system being supported.

B. DISCRETE-EVENT STOCHASTIC MODELING

One common M&S technique is the employment of discrete-event simulations. Such simulations take advantage of mathematical models created to represent a complex system, with the goal of promoting a better understanding of the system. The power of simulation is that it can shorten the time it takes to learn basic features of the system being modeled. By doing so, it allows analysts to discover critical interactions and a system's behaviors in a matter of minutes—a process which would otherwise take a considerably longer time to discover. When properly executed, a simulation could be thought of as a window into a possible future.

Mathematical models used to represent complex systems can be categorized in one of two ways—probabilistic or deterministic. A deterministic model uses its initial conditions to determine the outcome, or final state, of the system. It only needs to be run once, since the outcome does not change; it provides a single point estimate to describe system state. A probabilistic model has embedded within random elements representing one or more uncertainties or events within the system. This kind of model is very helpful in describing real-life systems, specifically in cases where data are estimated, e.g., reliability data or arrival processes. The key for a successful probabilistic model is to have the right distributions on the inputs and events. Provided that the inputs are modeled appropriately, the random variations of the simulation should provide users with a system state that closely represents the possibilities of the real system under study. For further information on M&S processes, read Law and Kelton (1999).

C. TLCM-AT FUNCTIONAL OVERVIEW

TLCM-AT is a probabilistic modeling and simulation analysis tool developed by Clockwork Solutions for the USMC (Clockwork Solutions, 2007). The simulation tool uses a model which is a simplified representation of a system at some point in time, intended to enhance the understanding of real systems. It is a holistic, continuous-loop representation of the life cycle of any system. Model functions include operations, maintenance, and logistics, as shown in Figure 1. Items in blue represent inputs to the model, while items in green represent outputs. The simulation manipulates the model to compress system operation over time, enabling users to discern interactions and system characteristics that would otherwise take a long time to detect. One main goal of TLCM-AT is to develop a holistic understanding by providing users with a fleet-level view of the system being modeled.

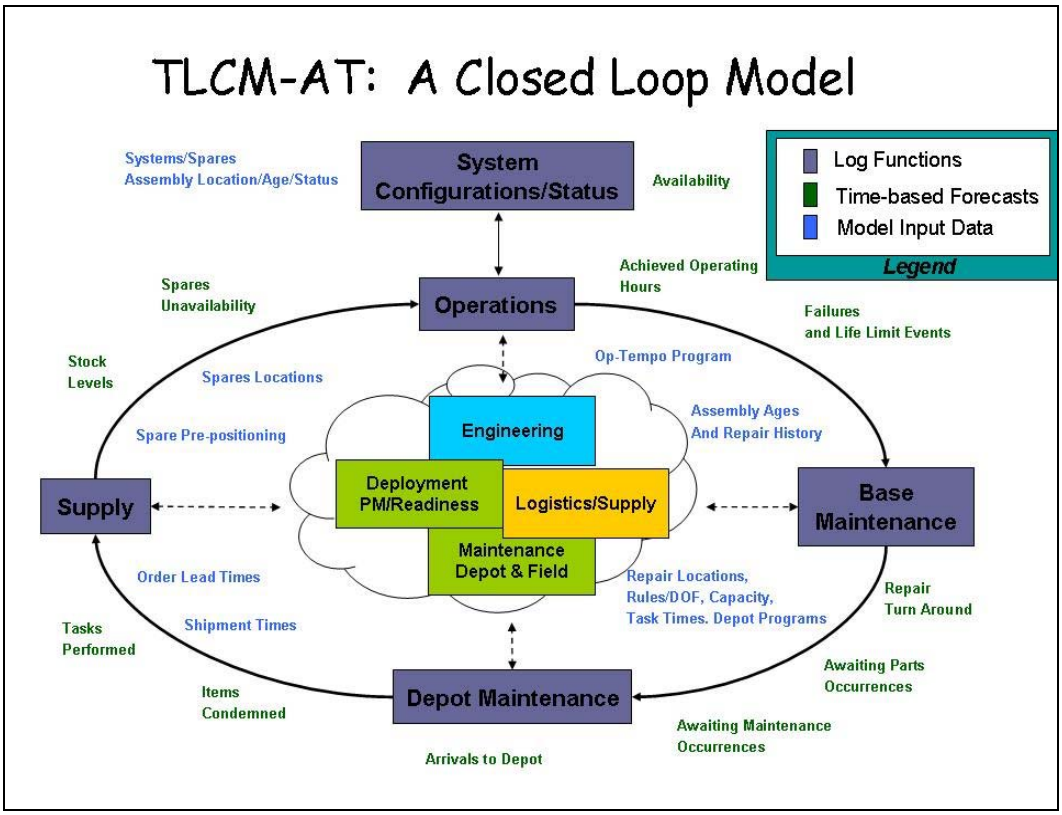

Figure 1. TLCM-AT continuous-loop model [Best viewed in color]
(From Clockwork Solutions, August 2007)

TLCM-AT models represent a fleet of systems, like the Light Armored Vehicle (LAV), or any other supported system. Each platform is created in the model using a hierarchical structure. Figure 2 shows the hierarchical structure concept, where platforms are composed of Line Replaceable Units (LRUs), while LRUs are composed of modules, assemblies, or Shop Replaceable Units (SRUs).

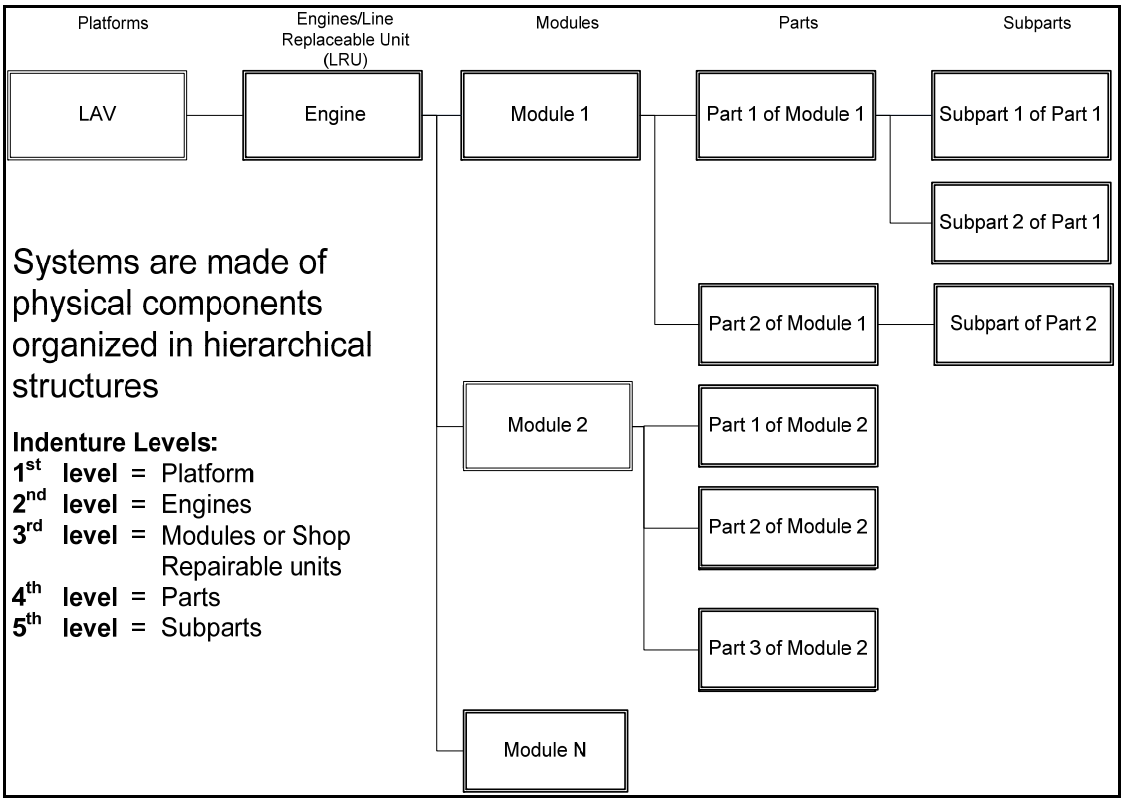

Figure 2. Platform hierarchical structure

The system is further broken down into submodules and other parts or consumables. The tool models the logistical infrastructure of the supported fleet. Bases are classified as Organizational, Intermediate, and Depot level; Depot represents the highest level of maintenance activity and also acts as supply during acquisition of new spares. For a further description of TLCM-AT, read Clockwork Solutions (2007).

D. DESIGN OF EXPERIMENTS (DOE)

For a simulation to be useful, it needs to be exercised using a well-designed set of experiments. A DOE is a complete specification of all input variables over all simulation runs. Input variables are simultaneously varied between their predefined low and high values. With the proper design, a simulation can help an analyst develop a basic understanding of the system modeled. It can also assist in the development of robust policies and can be used to asses the merits of various COAs. A poorly designed experiment, one that does not actively change multiple factors simultaneously, can waste

a lot of computing time, while yielding limited insights. In order to successfully investigate interactions, multiple factors need to be varied simultaneously.

A typical DoD model has a large number of factors, and produces outputs that represent many MOEs. The analysis identifies many significant effects, and interactions between two or more factors are common. Executing simulation models that simultaneously vary a large number of factors requires a great deal of computer power and time due to the large number of variable combinations necessary to complete the analysis (Kleijnen, Sanchez, Lucas, & Cioppa, 2005). To maximize the usefulness of the data collected from a simulation, while reducing the number of experiments to a manageable level, we use a technique included in Young's (2008) work called the Nearly Orthogonal Latin Hypercube (NOLH) (Cioppa & Lucas, 2006). An NOLH design allows analysts to efficiently explore much of the sample space, while reducing the number of designs required to obtain an accurate picture of the system being modeled. Insights gained from a well-designed experiment can be used to develop policies or to compare the merits of various COAs.

E. RELATED WORK

As part of his thesis research, Young (2008) performed an exploratory analysis of TLCM-AT. He analyzed four scenarios, each representing a different COA, relating to potential decisions about the USMC LAV-25. For each scenario, five factors were adjusted:

- Spare Levels
  - Total number of spares at each repair location
- Induction Quantity
  - A limit on the number of inductions that can occur at the Depot level in a given quarter, at a given repair facility
- Capacity
  - Number of parts that can be processed concurrently at a repair facility
- Service Times
  - Time it takes to repair a part
- Unscheduled Removal Rates
  - Part failure rate

The MOE used in Young's (2008) study is Ao, which is defined as the percent of systems available to operate for a specific period of time.

The three scenarios being analyzed represent possible COAs for a notional contingency deployment of the LAV. Each scenario is modeled separately, and they represent a plan of action to replace two faulty components on the LAV-25.

- OT 702275001, Sensor Unit, Laser Designator
- OT 702261001, Control Display Unit

The main scenario has the Marines deploying for combat operations in a tropical region and they need to include at least a battalion of LAVs in the force mix. Their deployment consists of lead and follow-on LAV battalions. The COAs represent alternatives on how to handle the faulty components during the deployment. Table 1 describes each of the nonbaseline COAs.

COA 1	o	Send a large number of spares with the follow-on battalion
COA 2	o	Acquire improved components
	o	Spend $1M on a one month Research and Development (R&D) program
	o	New LRUs cost 2.5 times the cost of legacy components
	o	Deploy fewer improved spares and install them whenever legacy parts are removed
	o	Acquire new parts when legacy parts are condemned—one for one
COA 3	o	A variant of COA 2
	o	Legacy parts are purchased to replace condemned parts
	o	No money is invested in the new LRUs
	o	Goal is to save some money, while maintaining similar level of performance and maintenance usage

Table 1. Description of COAs

The five varying factors during this work affect 25 LRUs, which are determined to be the top 25 degraders using a Clockwork Solutions-provided formula. These parts cause the most problems during the LAV-25 life cycle, using the baseline model. For more details on the top degrader selection, see Young (2008). Each of the three scenarios, and the baseline, is simulated in TLCM-AT using the DOE concept and NOLH. Each scenario runs for 129 design points, i.e., input combinations, using a NOLH design and each design point completes 30 replications. The result of the

simulation indicates that the practical change in Ao is very minimal and is too small to effect a decision. For a more detailed look at the previous results, see Young (2008). The small variation in Ao observed in the previous work is one of the motivations for the author to expand upon this research.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DESIGN OF PROTOTYPE APPLICATION

A. TLCM-AT MODEL DATABASE

When Clockwork Solutions designed TLCM-AT, they decided to use Access databases to manipulate input and output. This choice of data structure seems fitting given the complexity of a model designed to represent a holistic view of a system. For any given model, there are over 120 input and output tables making up the database file. Many of those tables contain over 240,000 line entries. One reason for this complexity is that TLCM-AT tracks each component by serial number, i.e., each component that makes up a platform, including LRUs, modules, submodules, and consumables, is tracked individually. Considering that an LAV includes over 175 components, coupled with a platform inventory of a few hundred, it is easy to see how manipulating a database of this size can get complicated.

Running a DOE scheme with TLCM-AT requires a method of modifying the database for each design point. The author gained knowledge by manipulating Access databases in the hope of being able to do it manually, either by directly modifying the database or by using TLCM-AT's GUI. It quickly became apparent, however, that direct database modification is too cumbersome, and an efficient method needed to be created if anyone hoped to perform any well-designed DOE using TLCM-AT. The solution is to create a computer application that automates the process of database manipulation to modify the model in accordance with a design, and to extract the required outputs needed for subsequent analysis.

B. DATA GENERATION PROCESS AND HANDLING

The data generation process is depicted in Figure 3. The newly created application takes as inputs a file containing the NOLH design and a baseline model in an Access database format. The baseline model is copied into a working model, which is modified using the parameter data from the NOLH design. Using a working model enables us to keep the baseline intact for use again during subsequent design points.

After the baseline model is copied and modified, the application launches TLCM-AT, which uses as input the modified working model. The number of replications executed by TLCM-AT is a direct input prescribed by the application's user. Upon completion of the simulation run, TLCM-AT saves the output files in the same database as the input. At that point, the application collects the output data pertaining to the MOEs of interest and it saves the information for later analysis.

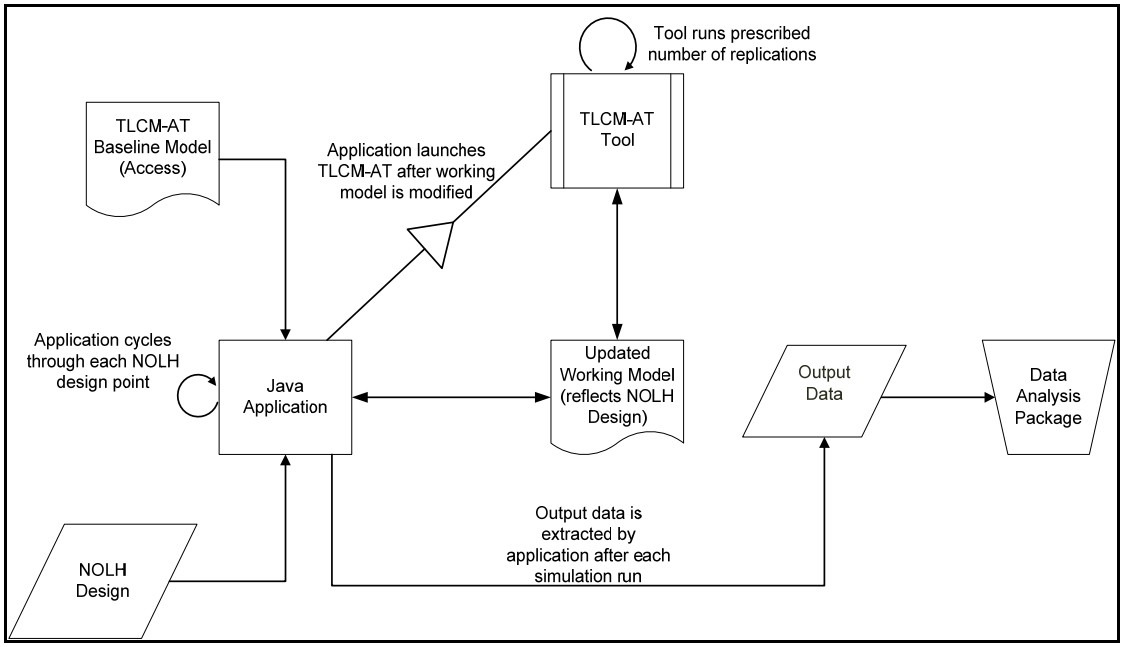

Figure 3. Data generation process

The above described process repeats itself, under the control of the application, and it continues to run until every design point of the NOLH is completed. Once the whole DOE is executed, the application saves a comma separated value (.csv) file containing all MOE data extracted from the simulations. The user can use any data analysis package to process the output data. The .csv file classifies the output data by design point and it matches those with the values of the input parameters that were modified prior to executing the simulation. Having the input data included in the output file is critical for proper data analysis techniques.

C. STRUCTURED QUERY LANGUAGE (SQL) INTRODUCTION

The new application's main strength is its ability to automatically modify Access databases. In order to accomplish this, the application uses a programming language known as SQL. A very unusual programming language, SQL is the official standard for relational database access (Horton, 2005). Any operation that needs to be carried out on a relational database has to be expressed in SQL. SQL is declarative in nature, which means that it tells the database engine what to do, not how to do it. The database engine is a separate piece of software that carries out any user-defined SQL command. The language is very readable in comparison to other programming languages; in fact, every SQL statement reads like a sentence, so it takes little time to grasp the concepts. Here is an example:

SELECT [Object type], [SRAN ID], [SERVER TYPE], [Tsf P1]

*FROM [*Server times] WHERE [Object type] = "value"*

In this case, the SQL statement is a query of the **Server times* table included in the database. The statement asks for values on four columns: *Object type*, *SRAN ID*, *SERVER TYPE*, and *Tsf P1* with criteria of *Object type* equal to "*value*." *SELECT* determines which data will be retrieved, *FROM* identifies the table to query, and *WHERE* sets the criteria. For more information on SQL and its application, see Horton (2005).

D. JAVA IMPLEMENTATION

The most current Java Development Kit includes a class created to maintain a connection between a Java program and a database. Java Database Connectivity (JDBC), part of the java.sql package (Horton, 2005), is a library that provides a connectivity interface and the means for a Java program to execute SQL statements in a relational database. More information on JDBC can be found in Horton (2005).

The first step before using Java for our simulation is to set up a suitable JDBC driver. Users can set up an Access database driver in Windows XP using the Open Database Connectivity (ODBC) Data Source Administrator dialog box as follows:

1. Select Start
2. Select Control Panel
3. Select Performance and Maintenance
4. Select Administrative Tools
5. double click on the Data Sources (ODBC) icon
6. Select the System Data Source Name (DSN) tab and click Add
7. Select Microsoft Access Driver (*.mdb)
8. Click finish
    a. Another box will come up with the title ODBC Microsoft Access Setup
9. Type the name of your database in the Data Source Name text box
    a. A suitable description in the Description field is optional
10. Click on the Select button in the Database section
11. Browse to your database, select it, and click OK
12. Exit out of all dialog boxes

The system should now be able to work, barring any unforeseen problems. As stated earlier, for more information on this topic, see Horton (2005).

Appendix A includes the source code for the application. The Java class *UpdateDataBase* contains the Main Method. The application uses the following input:

- List of secondary repairables (SecReps)
    - Intermediate and Depot Level Repairables according to applicable Source Maintenance and Recoverability (SMR) codes
- List of SRANs[2]
    - This list includes Organizational Level activities only
- NOLH design
    - List of parameter values for each design
- Access database baseline model

[2] SRAN is an internal TLCM-AT code for base location (Clockwork Solutions, 2007).

Using the above input, the application first makes a copy of the baseline model and creates a working file named "smalldb1.mdb." This choice of name cannot be altered, it is the only file name used by TLCM-AT whenever the tool is launched from the command line. The number of histories is the only argument used when operating TLCM-AT from the command line.

After the application creates the working model, it modifies it using the parameter information included in the NOLH design. When the application makes a change to the baseline model, it only affects the LRUs included on the SecRep list. In the case of Spare levels, the application creates the number of new SecReps given by the NOLH design for each location listed on the SRAN list. If the NOLH design number is zero, no new spares are added; if it is three, three new spares of each SecRep are added to each location. For Induction Quantity, the application sets a quarterly limit on SecRep inductions into each depot repair facility equal to the value in the NOLH. In the case of Capacity, the application sets a limit on the number of SecReps that each Intermediate-level facility can process at any given time, equal to the value in the NOLH. It is important to note that there are no limits on the total number of SecReps that can be processed; these limits apply to the number of SecReps of the same type. The application takes the Server Times and Repair Degradation design levels and multiplies them by their current values to set the new levels.

Upon successful completion of all the updates, the Java application uses the command line to launch TLCM-AT. Afterwards, the program collects the necessary data from the working model to process the MOEs of interest. This process repeats itself for the number of experiments in the DOE.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SCENARIO DEVELOPMENT AND EMPLOYMENT OF APPLICATION FOR DATA GENERATION

A. INTRODUCTION

This chapter explains the four fictional LAV scenarios used as the basis for this analysis and how the Java application is employed to generate the data. The application uses DOE techniques to perform simulation runs on each scenario, with the purpose of collecting enough data to be able to decide on a robust COA. After each scenario, the application automatically collects and saves the data for analysis afterwards.

B. SCENARIO DEVELOPMENT

A notional scenario involving the deployment of LAVs forms the foundation for this analysis. The details of the scenario are as follows: The Marine Corps receives a warning order to deploy troops to a tropical region. Due to mission and geographical considerations, the Marines decide to include a lead LAV battalion with the deploying force, followed by a second battalion a few weeks later. Historical data shows that combat operations in hot and humid environments adversely affect the unscheduled removal rates of two computerized LRUs in the LAV system:

- OT 702275001, Sensor Unit, Laser Designator
- OT 702261001, Control Display Unit

A few weeks after the lead battalion's arrival to theater, problems with the above-mentioned LRUs begin to affect combat operations. At this point, decision makers are considering the following three COAs to mitigate the effect of increased failure rates due to environmental conditions:

- COA 1
  - Send a large number of spares with the follow-on battalion
- COA 2
  - Acquire improved components
  - Spend $1M on a one-month R&D program

    - New LRUs cost 2.5 times the cost of legacy components
    - Deploy fewer improved spares and install them whenever legacy parts are removed
    - Acquire new parts when legacy parts are condemned—one for one
- COA 3
    - A variant of COA 2
    - Legacy parts are purchased to replace condemned parts
    - No money is invested in the new LRUs
    - Goal is to save some money, while maintaining similar level of performance and maintenance usage
- Baseline
    - Take no action

The simulation runs using four Access database models and each model represents a distinctive COA. Notional scenarios used in this thesis were modeled by Dr. Peter Figliozzi, modeler and analyst for Clockwork Solutions.

The application requires a SecRep list to be included in the inputs. Using the SMR codes as a reference, the SecRep list includes every LRU repairable at the Intermediate or Depot level. To build the list, users can query the **Object type* table, filtering the results for *Object type* greater than 700000000 and less than 900000000. The result is a list of 175 SecReps for models with no upgraded LRUs, or 177 otherwise. After the SecRep list is developed, users need to save the file using a .csv format, entering each SecRep on a separate row. Care needs to taken to ensure that there are no extra characters after the last entry in the list or the application can fail. Users can locate the cursor at the end of the last entry in the list and press "Delete" several times to ensure that nothing else is included in the .csv file.

The **Base Names* table provides the information required to create an SRAN list. For this simulation, the list includes only the Organizational-level bases, Marine Expeditionary Forces (MEFs) 1 through 4 and the jungle base.

C. DESIGN OF EXPERIMENTS (DOE)

In order to maximize the efficiency and space-filling effect of this design, the author uses the orthogonal and nearly orthogonal LH worksheet (Sanchez, 2005) to develop the design points. The worksheet is an Excel-based tool developed to ease the design of large-scale simulation experiments. The author describes five varying factors in Table 2.

Factor	Label	Low Range	High Range	Decimal Places	Description
Spares	Spares	0	5	0	Determines number of spares added to system
Induction Quantity	IQ	0	30	0	Maximum number of each SecRep that could be inducted into each Depot for repairs
I-Level Capacity	I Cap	0	30	0	Maximum number of each SecRep that could be processed at each I-Level facility in any given time
Degradation Rate	Deg	0.5	1.5	4	Current value multiplied by the design value
Service Time	ST	0	10	4	Current value multiplied by the design value

Table 2. Range of factors for DOE

To use the worksheet, users have to select the appropriate sheet based on the number of factors to be varied. For this experiment, the author purposely uses the sheet for 17-22 factors so he can develop 129 design points. Next, users can fill in the labels, upper and lower values and number of decimal places desired. Figure 4 shows a partial view of the NOLH Worksheet Design with the high and low values chosen for this design, along with the number of decimal values.

low level	0	0	0	0.5	0
high level	5	30	30	1.5	10
decimals	0	0	0	4	4
factor name	Spares	IQ	I Cap	Deg	ST
	1	13	12	0.9531	3.3594
	4	9	13	0.9609	0.9375
	2	23	0	0.7734	4.1406
	3	27	9	0.8672	4.375
	0	12	17	0.7344	1.0156
	4	13	21	0.5078	3.9844
	2	30	23	0.7891	1.5625
	3	21	27	0.5625	3.4375
	0	1	8	0.7031	1.9531
	5	2	7	0.7656	1.1719

Figure 4. Portion of NOLH worksheet design

Each column represents a varying factor, while each row represents a design point. Appendix B has a copy of the full NOLH design used for this experiment.

A goal of this research is to develop a metamodel that can easily explain the relationships between input factors and model outcomes. A metamodel is defined by Cioppa (2002) as a relatively simple function that is estimated given an experimental design and the corresponding responses. The metamodel uses factor coefficients to describe what effect factors have on MOEs of interest. One important characteristic of a design of experiment is that the columns representing the inputs are not strongly correlated, i.e., they do not have a strong linear relationship since strong correlation among input variables can adversely affect the precision of metamodel coefficient estimates (Cioppa, 2002). Table 3 summarizes the strength of the linear relationships between each pair of input parameters. The number 1.000s across the diagonal show the perfect linear relationship of each input with itself. The greatest value shown in the correlation matrix, at 0.0274, does not represent a strong linear relationship between the input variables of Spares and IQ.

Correlations

	Spares	IQ	I Cap	Deg	ST
Spares	1.0000	0.0274	0.0074	0.0138	0.0111
IQ	0.0274	1.0000	-0.0000	0.0030	0.0030
I Cap	0.0074	-0.0000	1.0000	-0.0084	-0.0051
Deg	0.0138	0.0030	-0.0084	1.0000	-0.0000
ST	0.0111	0.0030	-0.0051	-0.0000	1.0000

Table 3. Input parameter correlation matrix

Another way to look at linear relationships between pairs of input variables is to create a scatterplot matrix displaying all two-way input combinations, like the one displayed in Figure 5.

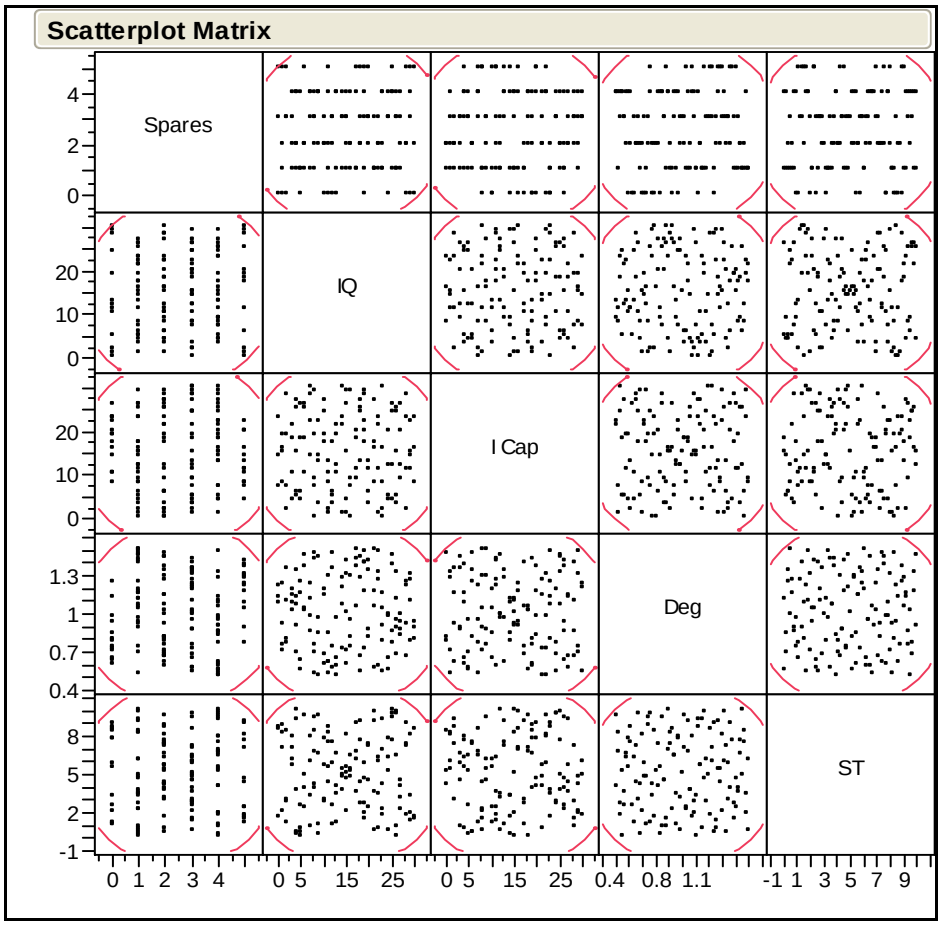

Figure 5. Two-way input combinations

The scatterplot gives a visual indication of linear relationships and, in this case, none are present. Linear relationships are discernable in a scatterplot matrix when a pattern exists between two inputs, e.g., large values of one input are paired with large values of another input for a positive relationship, or large values of one input are paired with small values of another input for a negative relationship. The ease to distinguish a line pattern increases as the strength of the relationship increases.

In the case of Spares, the design has a low value of zero and a high value of five, with zero decimal places; all other high and low values can be seen in Figure 4 or in Table 2. The Java application adds the number of spares displayed in a given design point as new objects into the **Object attributes initial* table. The application has to insert each object into the table with its own object type, serial number, and location where the spare will be stored. These new spares are stored in the Organizational-level bases included in the SRAN list.

The design value for IQ goes directly into the **Depot Spares Program* table. The application looks for every table entry pertaining to a SecRep and changes the existing *Quantity* value, which represents the number of a specific SecRep that can be inducted at that Depot facility on a given quarter, to the new design value. For I Cap, the process is the same. In this case, the application updates the **Capacity* table by updating the maximum number of SecReps that an I-level facility can process at a given time. This update applies to all I-level facilities and to all SecReps, but the limitation is specific to a particular component, i.e., the limitation only applies to the number of SecReps of the same type being processed in the same facility.

The application treats the Deg and ST values differently than those explained above. For Deg, it updates the **Unscheduled Removal rates* table by multiplying the Deg design value by the existing *Rate* and updating it accordingly. In a similar fashion, the application updates the **Server times* table by multiplying the existing service time value—defined in the table as *Tsf P1*—by the ST design value and updating it accordingly.

D. SIMULATION RUNS

At this point, we have all the input necessary to run our 129 designs for 30 replications each. Using the **Analysis Range* table, the author limits the length of each replication to 12 quarters. This limitation is designed to expedite the runs and it fits perfectly with our scenario-based models, which end the LAV deployment after 12 quarters. Additionally, every MOE collected during this experiment describes the system at the end of the 12th quarter. To run the tool for 30 replications, users need to include the number 30 at the end of the command line argument used to launch the tool inside the application's main method.

By adjusting the simulation runs to 12 quarters each and modifying the Java application, the author is able to shorten the length of the experiment dramatically. During Young's (2008) experiments, the process completed 2 design points per hour; the revised experiment completes 3.6 design points per hour. This reduction in time is significant given the fact that the revised application is able to modify parameters over every SecRep, compared to the initial application that only applied changes to a limited number of SecReps, which were selected by determining the worst degraders in the system. The experiment includes four scenarios, 129 design points, and 30 replications for each scenario, for a total of 15,480 runs and 143.3 hours of computing time, using a typical Pentium® 4 computer with Windows XP Professional.

The output produced by the application is a .csv file. The file includes a header row with the remainder rows representing a design point each. Output results include MOEs of interest and input values lined up in the same row. It is important to note that TLCM-AT outputs only include the sample mean and standard deviation resulting from the replications performed for each design point, thus limiting the analysis that can be performed by not having access to the raw data.

THIS PAGE INTENTIONALLY LEFT BLANK

V. DATA ANALYSIS

The combination of the Java application, the capabilities of TLCM-AT, and the benefits of the DOE described in the previous chapter, form an excellent construct, which allows for a remarkable collection of information to be extracted from the simulation experiments. In this chapter, the data collected and its post processing are described and analyzed. The purpose is to demonstrate the kind of analysis that can be completed, and insights that can be gained, by using the technology presented in this research. Using the scenarios described in Chapter IV, the analysis centers on data collected at the end of the 12^{th} quarter of operations, which is programmed in the scenarios as the end of the contingency.

Throughout this analysis, it is important to keep in mind that the results presented here are only applicable to the four scenarios included in the simulation, and should not be generalized to other applications. The primary goal is to demonstrate how using the Java application with a well-designed DOE can significantly enhance the value of TLCM-AT to the USMC. The analysis focuses on finding analytical insights that could be useful to decision makers.

A. TLCM-AT RESULTS

1. Availability

Ao differences among COAs are minimal, and by themselves do not justify any decision; this behavior closely matches what Young (2008) found in his study. This result is very surprising, given the fact that for this study, a larger number of SecReps were affected by the DOE modifications. During Young's (2008) analysis, only 25 SecReps were affected by the modifications, compared to a minimum of 175 in this study. The unexpected behavior of Ao within TLCM-AT can be attributed to the way the tool measures Ao. According to Clockwork Solutions (2005), the tool measures Ao as the ratio between the average number of platforms that are operating and the number of platforms that should be operating. An operating object, including platforms, is defined

further as an object that is populated—i.e., every required subassembly is installed. This definition of populated does not cover the material condition of the components populating the object; as a result, an object appears available in terms of Ao, but it might be filled with faulty components.

Dr. Naaman Gurvitz, chief scientist for Clockwork Solutions, hypothesizes that the surprising Ao behavior is because in these models there are a total of 703 LAVs, 211 of which are not driven during the simulation and, therefore, they achieve 100% availability. Thus, when the tool averages the availabilities in quarter 12 over all the platforms, the changes are being suppressed, resulting in smaller Ao variations (Gurvitz, 2008).

Figure 6 shows Ao data from each COA independently sorted in increasing order. The numbers across the x axis have no meaning other than to show that each series represents 129 design points, but no connection can be made between design points and the index value. One way to read this graph is to see how all COAs achieve an Ao of 78 percent or better in 124 of the 129 design points. More than 96 percent of all design points have an Ao between 78 and 84 percent. In general, COA2 is better than COA3, followed by COA1 and baseline, respectively. With the exception of some small differences, Ao data for each COA follows a very similar pattern; the range of data is nearly the same for all COAs, from 72 to 84 percent. The author uses the same approach to build graphs from the other MOEs collected during the simulation.

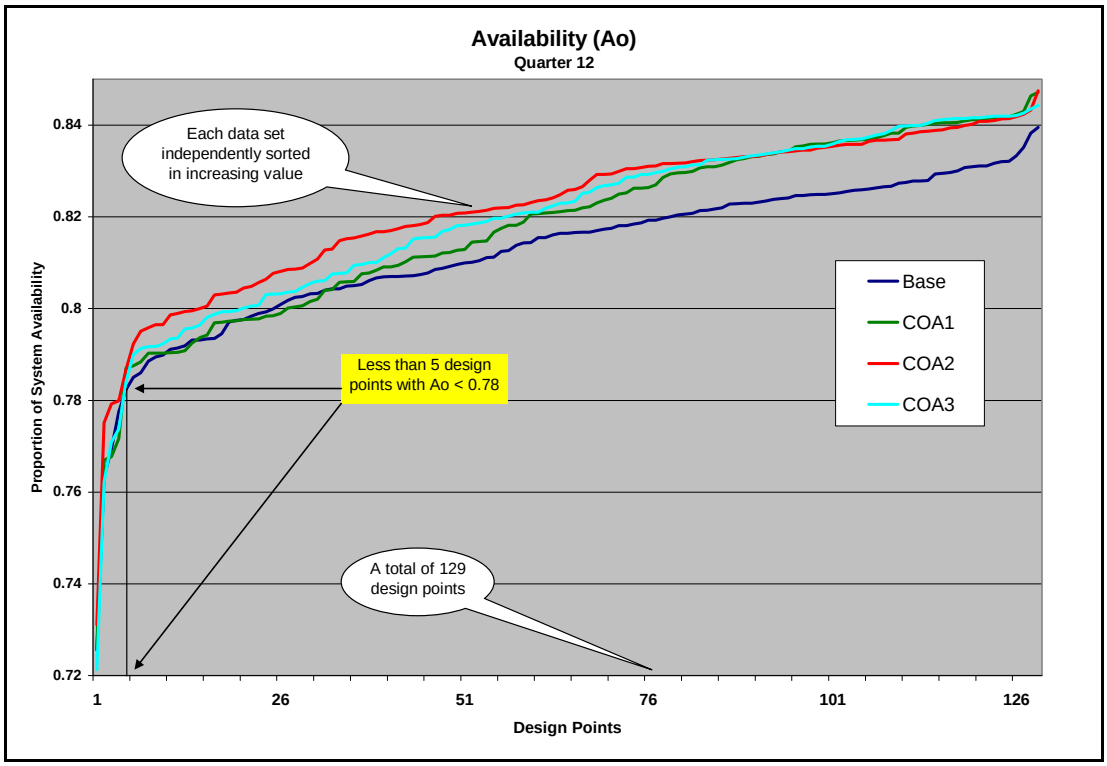

Figure 6. Percent of systems available to operate during the 12th quarter [Best viewed in color]

2. Achieved Operating Hours (AoH)

Figure 7 represents AoH data independently sorted in increasing order. The data shows that there is a significant difference in the number of AoH among COAs, with COA2 consistently producing the best results, followed by COA3 and COA1, respectively. AoH data for COAs 1-3 converge at 123,916, because that is the programmed number of operating hours required in the scenarios. COA2 achieves fewer than 110,000 operating hours in 5 of 129 design points, while it exceeds 120,000 operating hours in 86 design points. At the same time, COA3 achieves fewer than 110,000 operating hours in 33 of 129 design points. In contrast, the baseline model achieves fewer than 110,000 in 115 of 129 design points.

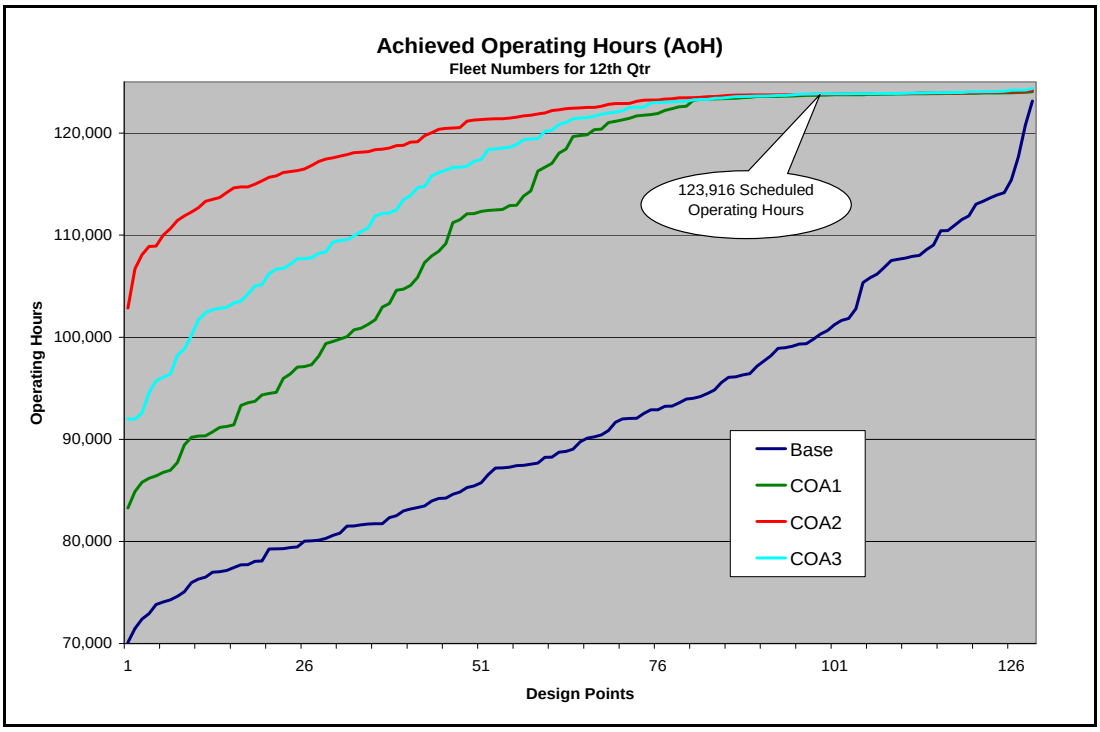

Figure 7. Achieved operating hours [Best viewed in color]

The 129 design points represent a space of possibilities, i.e., a range of possible conditions that might materialize during real-world circumstances. Table 4 shows a breakdown of how each COA performed against the others throughout the space of possibilities. The numbers represent how many times a COA in the row heading outperforms a COA in the column heading. For example, COA2 outperforms COA3 a total of 99 times; the baseline does not outperform any other COA. Since COA2 produced such positive results over a much greater number of design points, and has the smallest observed variance, it can be concluded that it is the most robust COA of the four analyzed.

	base	1	2	3
base		0	0	0
1	129		16	25
2	129	113		99
3	129	104	30	

COA2 achieved better than COA1 113 times

Table 4. Breakdown of COA performance

A robust COA can yield fewer AoH surprises when compared to other alternatives, due to their superior performance over the whole space of possibilities. Figure 7 shows how well COA2 behaved over a wide range of parameter values. Such a performance provides evidence that this alternative is the least sensitive to changes in field conditions, thus producing the fewest surprises if implemented. Range is another measure of robustness that can be applied to this data. COA2 has a range of just over 21,000 operating hours compared to the baseline, COA1 and COA3, with a range of 53,000, 40,000, and 32,000 operating hours, respectively. The small range achieved by COA2 provides additional evidence that, if implemented, COA2 would produce the fewest surprises. For the operator, these results suggest that this COA should provide a minimum of 102,852 operating hours, which is the lowest AoH in the data set, regardless of drastic changes in field conditions. Given the dynamic nature of combat operations, having access to this kind of information is critical to selecting a robust COA.

3. Failure-Induced Platform Events (En)

Figure 8 represents failure-induced platform events (En) data during the 12^{th} quarter. The most interesting part in this graph is how the data set representing COA2 has the smallest range among all alternatives. Table 5 represents a data range comparison. COA2 has the lowest difference between lowest and highest levels, providing evidence of its level of stability throughout the sample space. This seems to fit perfectly with COA2's robust nature and the expectation of few surprises. The baseline model consistently achieves a lower En than any other alternative, but these lower values are a result of the system's exceedingly low AoH, i.e., a platform has to be operating before it can have a failure. It is noteworthy that COA2 achieves lower En than COAs 1 and 3, in spite of having achieved a greater number of AoHs, which means that the platforms operated longer, but failed fewer times. COA2 is able to operate longer while experiencing fewer failures because it uses two improved LRUs to replace the two previously identified faulty LRUs, and these new and improved components are modeled with far superior reliability than the legacy ones.

	Base	COA1	COA2	COA3
Max	1,763.8	2,229.6	1,638.4	1,931.8
Min	927.1	1,270.7	1,247.1	1,216.0
Difference	836.7	958.9	391.4	715.8

Table 5. Range of Events (En) data

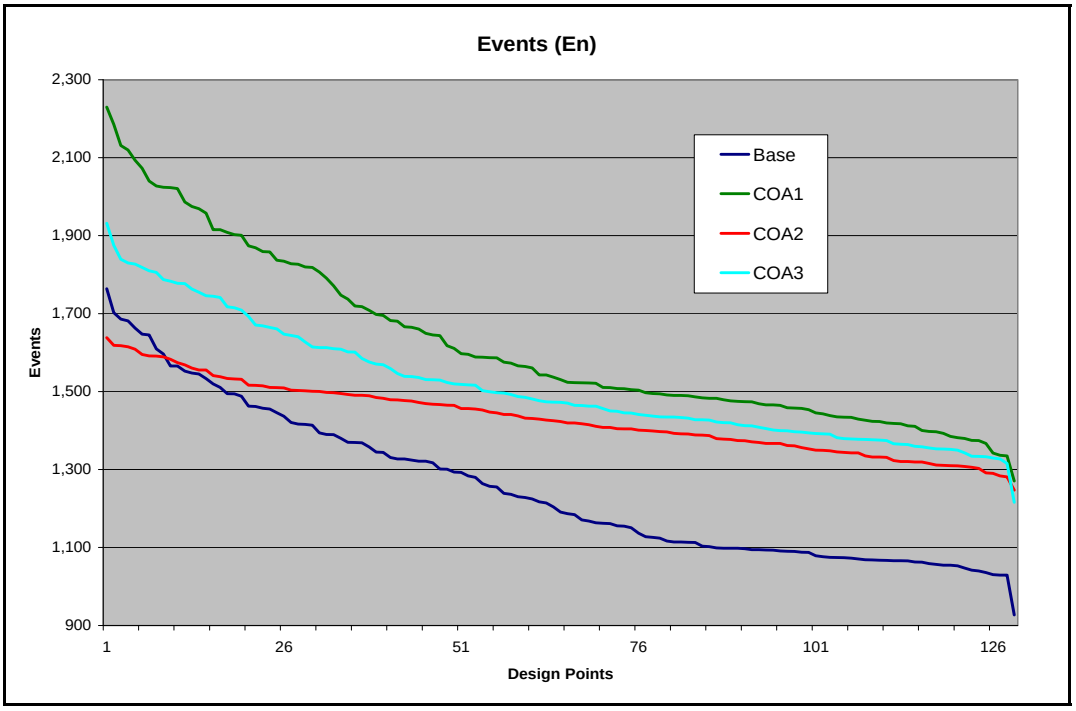

Figure 8. Number of platform events due to failures [Best viewed in color]

4. New Spare Buys

SBn data, shown in Figure 9, shows a similar trend in terms of data range. Again, COA2 is the most consistent of the four alternatives and it can provide the fewest surprises to the user. In many designs, COA2 has a higher SBn number than the other alternatives; still that is a result of the scenario, which programmed the system to replace failed SecReps with new spare parts. The baseline achieves low SBn levels for similar reasons, as the scenario is programmed to buy spare parts only when SecReps are

condemned by the system. In the baseline scenario, new spare buys are triggered by the condemnation rates, while in COA2, new spare buys are triggered by SecRep failures, resulting in increased SBn levels.

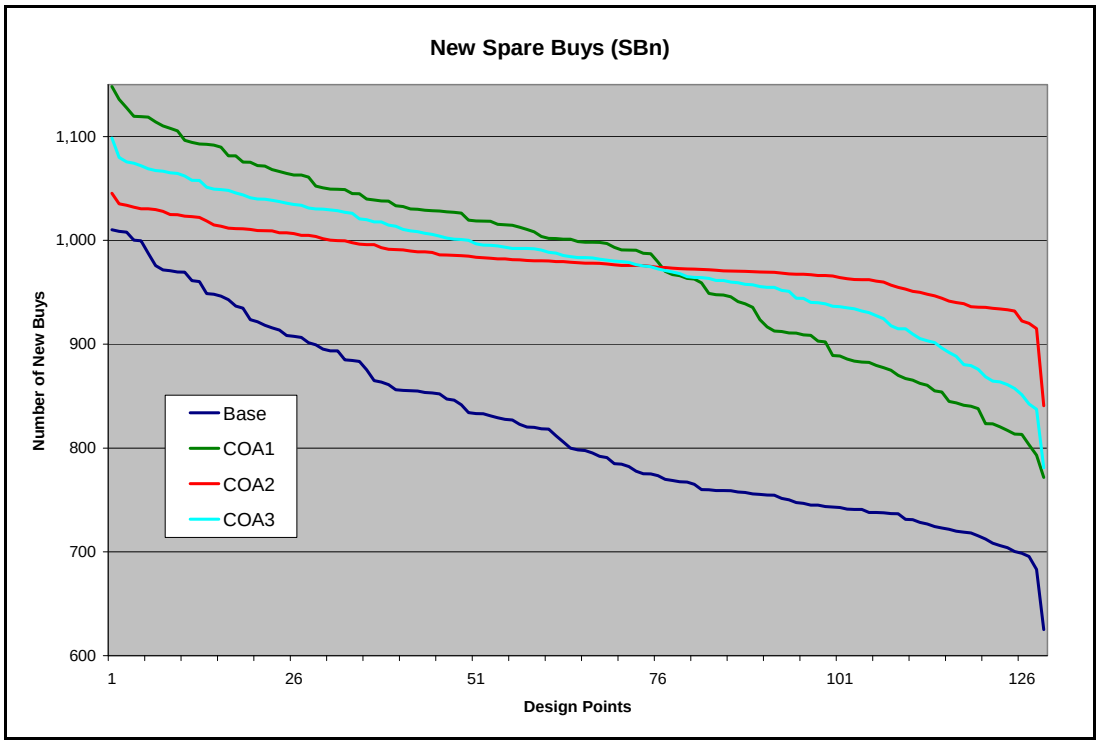

Figure 9. Number of new spare buys [Best viewed in color]

The baseline SBn numbers are driven by the condemnation rates implemented in the model. When any previously identified faulty LRU requires a Depot level repair, the part is condemned or repaired, based on a condemnation rate included in the model. TLCM-AT uses a random number and the aforementioned rate to determine the fate of the failed part. If the LRU is condemned, the model triggers a new spare buy. Condemnation rates are small enough to result in few spare purchases, since most LRUs are repaired several times before they are condemned, and replaced by a new spare. Alternatively, COA2 is implemented to handle LRU failures differently; the model triggers a new spare part purchase as soon as one of the identified LRUs fails, resulting in greater SBn number. The condemnation rate used in the baseline model plays no role in COA2.

5. Output Correlation

Figure 10 shows a COA2 scatter plot of every MOE collected in the simulation. This plot provides users with insight into the behavior of TLCM-AT. Experience dictates that there should be a correlation between many pairs of MOEs. Many of the existing ones are obvious, e.g., Sh and En, En and Pt, Sh, and Pt, etc. More interesting is the exploration of the not-so-obvious correlations such as AoH, which has an interesting relationship with En, Sh, and Pt. AoH achieves consistently high values in the low and high ends of En, Sh, and Pt, alternatively; when these three outputs are in their middle ranges, AoH is less predictable. Ao has the same behavior as AoH in relationship to En, Sh, and Pt. In Figure 10, the COA2 AoH versus En plot shows how, in the middle ranges of En, AoH is less predictable, but as En data moves out to the low and high ends, AoH consistently achieves high values. From a maintenance professional perspective, these relationships are the most interesting. In addition, the scatter plot shows a very clear extreme point, which will be discussed later in the analysis.

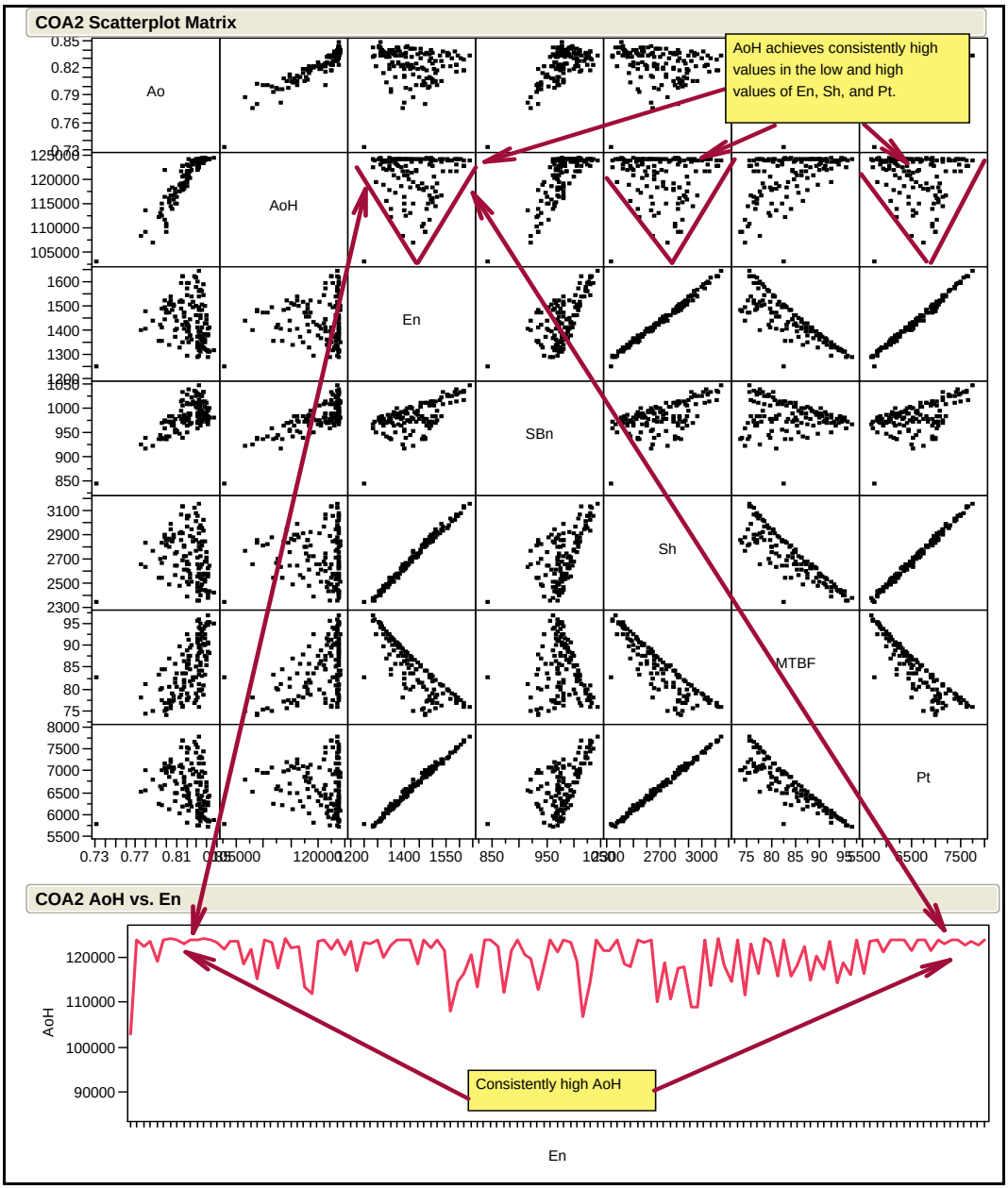

Figure 10. COA2 MOE scatter plot matrix [Best viewed in color]

Table 6 displays a correlation matrix of every COA2 MOE combination. The correlation matrix describes how strong the correlation is between two MOEs—a correlation equal to one means there is a perfect linear relationship between two data sets. Correlation between Sh and Pt is greater than .99, showing that their correlation is very strong, i.e., either is a very strong predictor for the other.

COA2 MOEs Correlations							
	Ao	AoH	En	SBn	Sh	MTBF	Pt
Ao	1.0000	0.9234	-0.0944	0.6670	-0.1543	0.5744	-0.1954
AoH	0.9234	1.0000	0.0152	0.7633	-0.0557	0.5212	-0.0931
En	-0.0944	0.0152	1.0000	0.6305	0.9949	-0.8438	0.9939
SBn	0.6670	0.7633	0.6305	1.0000	0.5726	-0.1290	0.5420
Sh	-0.1543	-0.0557	0.9949	0.5726	1.0000	-0.8775	0.9969
MTBF	0.5744	0.5212	-0.8438	-0.1290	-0.8775	1.0000	-0.8965
Pt	-0.1954	-0.0931	0.9939	0.5420	0.9969	-0.8965	1.0000

Table 6. COA2 MOE correlation matrix

B. ANALYSIS

The author decided to use AoH as the main MOE of interest for this analysis. As a result, this section seeks to identify the critical factors affecting AoH by demonstrating a comprehensive analysis of the output data. The section starts with a data summary, and continues with the search for critical factors and the creation of a metamodel.

1. Data Summary

Before moving to the identification of critical factors, it helps to take a good look at the whole data set. Figure 11 shows a summary of AoH data separated by COA. The figure shows the distribution for each alternative, along with some statistics, including sample mean. This result clearly shows, once again, the strength of COA2. Still, the nature of these data sets makes the use of sample mean less attractive due to its sensitivity to outliers and its sensitivity to behaviors like those of COAs 1 through 3, where they achieve a required number of operating hours and then stop operating. A better statistical measure for this analysis is the sample median, which is not affected by either outliers or tightly grouped data subsets. Table 7 compares the mean and median for each data set. The median in this case provides a better summary measure of the data. Baseline data shows a lower median than the mean, which indicates that the true performance of the baseline is worse than previously expected. The opposite can be said about COAs 1 thorough 3; in these cases, the median is higher than the mean, providing evidence that these alternatives are, indeed, significantly strong.

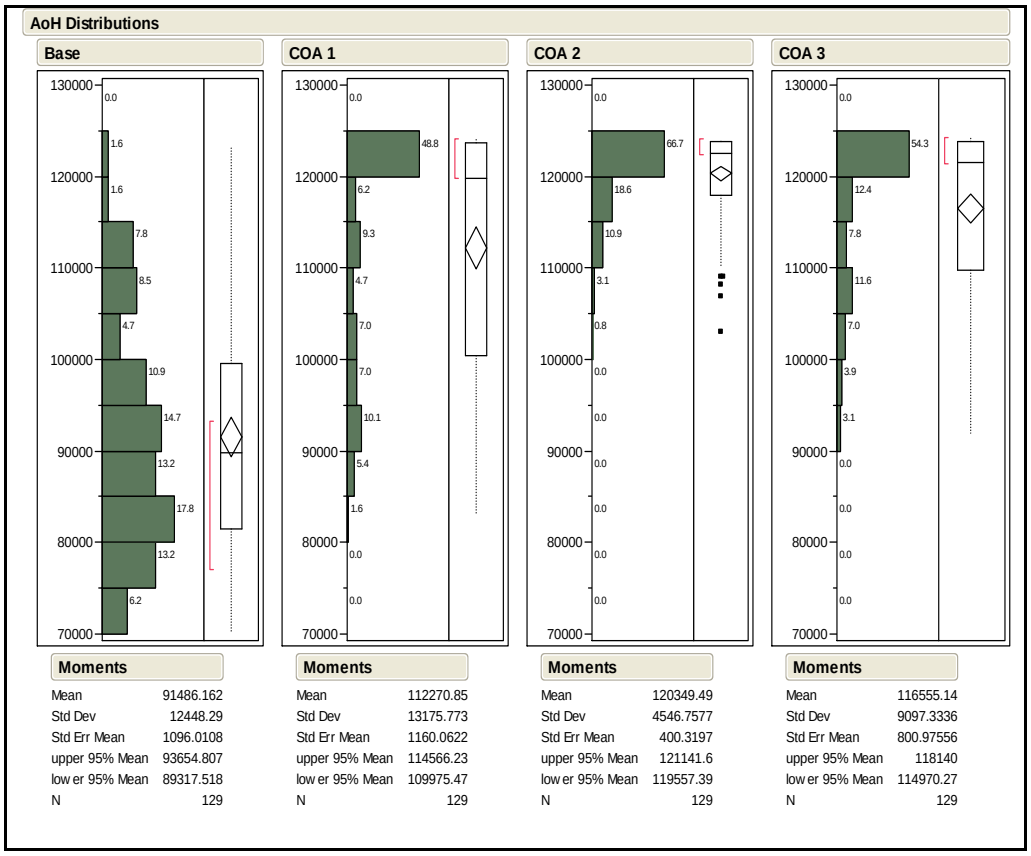

Figure 11. Summarized AoH data

	Base	COA1	COA2	COA3
Median	89766.52	119762.9	122474.9	121484.1
Mean	91486.16	112270.8	120349.5	116555.1

Table 7. Summary of Achieved Operating Hours (AoH) data

Next, the author takes a closer look at COA2, since it appears to be the most robust alternative. Figure 12 expands COA2's outlier plot to allow identification of the extreme values included in the data. The extreme values are identified as designs 77, 81, 75, 71, and 59.

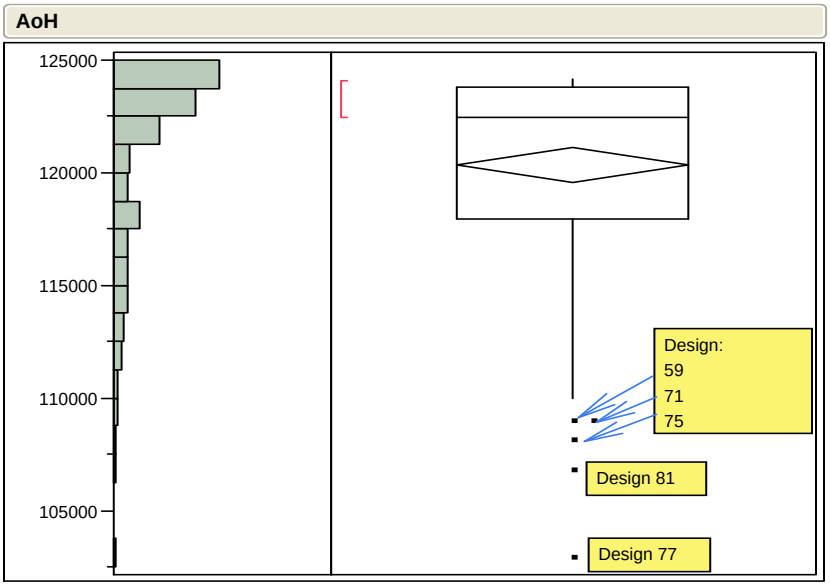

Figure 12. Expanded COA2 outlier box plot

To explain the behavior of these points, we need to look at the input of each design as shown in Table 8. The worst-performing design point has no spares or induction quantity; it has a relatively high capacity, a higher than normal degradation, and very high service time. Any experienced maintenance professional would have a difficult time arguing with these results. This set of parameters would have been disastrous in a real-life scenario; in fact, the results should have been worse than what they are, but they still show that TLCM-AT does behave in a rational manner.

Design	Spares	IQ	Capacity	Degradation	ServTime	AoH
77	0	0	23	1.1172	8.5938	102852.4
81	1	8	0	1.4688	8.2813	106706.3
75	0	28	23	1.2344	8.8281	108084.1
59	1	25	12	1.4453	10.0000	108883.5
71	1	17	9	1.4922	6.0156	108926.1

Table 8. Extreme values design data

Looking at the rest of the data points, the common characteristics among the extreme values are the high service times, coupled with low spare levels. Another interesting point is that when spares levels are 1, degradation increases to near its

maximum level. Thus, one spare is not enough when the failure rate gets too high. Capacity and induction quantity do not seem to make a difference at this point, but further analysis will determine a final list of critical factors.

2. Simple Linear Regression Model

The regression analysis in this section starts with a look at the individual factors by way of simple linear regression models, followed by a study of all factors, in order to support the later development of a metamodel. Figure 13 shows the percent of the variability of the data explained by each of the quantitative factors alone. RSquare is the measurement used to determine percent of variability explained by a model. It represents the ratio of variation in AoH explained by regression, divided by the total observed variation in AoH. A value of RSquare equal to 100 percent means that the selected model fits perfectly; conversely, a value of zero indicates that the fit is no better than using the mean of the data as a model.

The first insight into our system is that service time has the most impact on AoH and, equally important, the above assessment that capacity and induction quantity do not seem to make a difference is proven by this simple analysis. These values were calculated by fitting a simple linear regression model for each regressor against AoH. Spares and degradation are the only other factors to show any influence. From the values in Figure 13, it is clear that service time and degradation are critical predictors of how many operating hours a fleet of LAVs can run successfully, while a full regression model will decide the effect of spares.

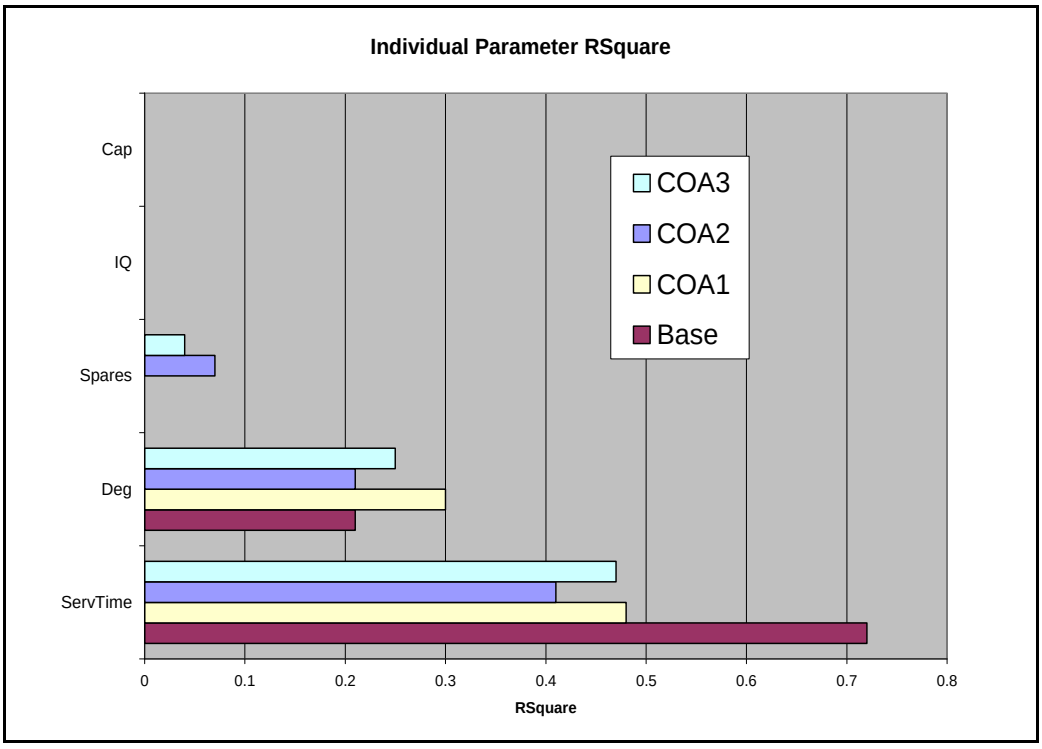

Figure 13. Variability explained by each individual factor without interactions [Best viewed in color]

3. Multiple Linear Regression Model

The first attempt to develop a metamodel is to use a multiple linear regression model, without polynomial terms, with every factor in the simulation. It is always a good idea to do this in order to determine how well this simple model behaves. This process is not the same as the one displayed in Figure 13. In this instance, every factor is included in one model. In contrast, the models described in Figure 13 were built using one factor at the time. Figures 14 and 15 display the summary results for each COA presented.

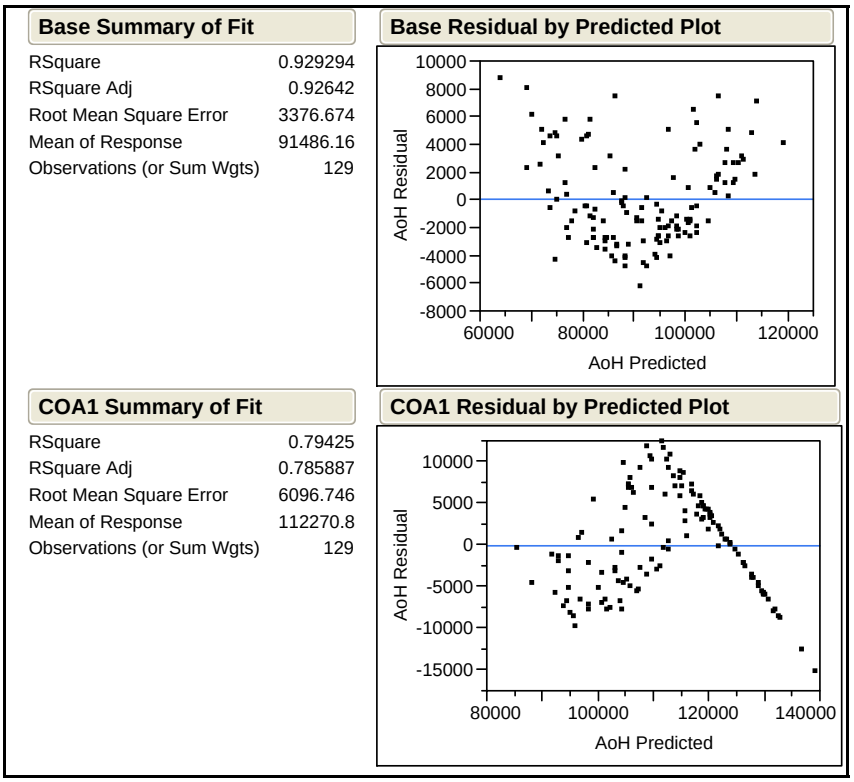

Figure 14. Base and COA1 main parameter model

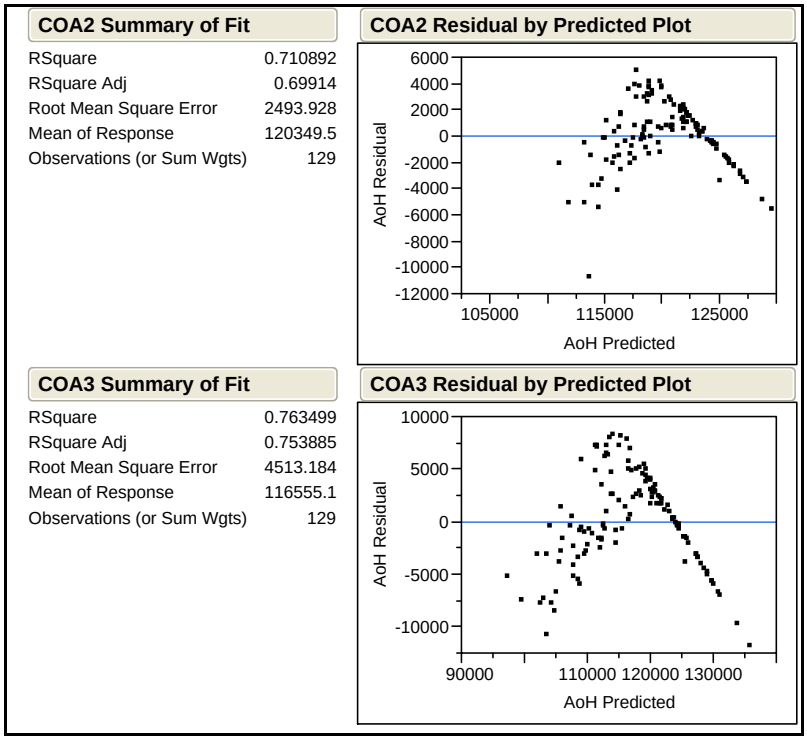

Figure 15. COA2 and COA3 main parameter model

Figures 14 and 15 show how challenging it can be to analyze a scenario in TLCM-AT, where the platforms have an easily achievable target for required operating hours and do not operate past that level. The clear diagonal lines included in the residual plot of COAs 1-3 represent the design points where the LAVs reached the required operating hours and stopped operating. This is a normal situation in real life, but complicates the data analysis. The remainder of the analysis will focus on the Baseline scenario, due to its more conforming data set.

4. Polynomial Regression Model

An analysis of variance (ANOVA) on the Baseline multiple linear regression model described in Figure 14 results in an F-test p-value3 of less than .0001, providing evidence that the regression model is highly significant. The RSquare is a very strong 93 percent. Model results prove that, in the presence of service time, degradation, and spares, the parameters' induction quantity and capacity are not statistically significant at any reasonable level. Figure 16 shows the result of the F test and parameter estimates.

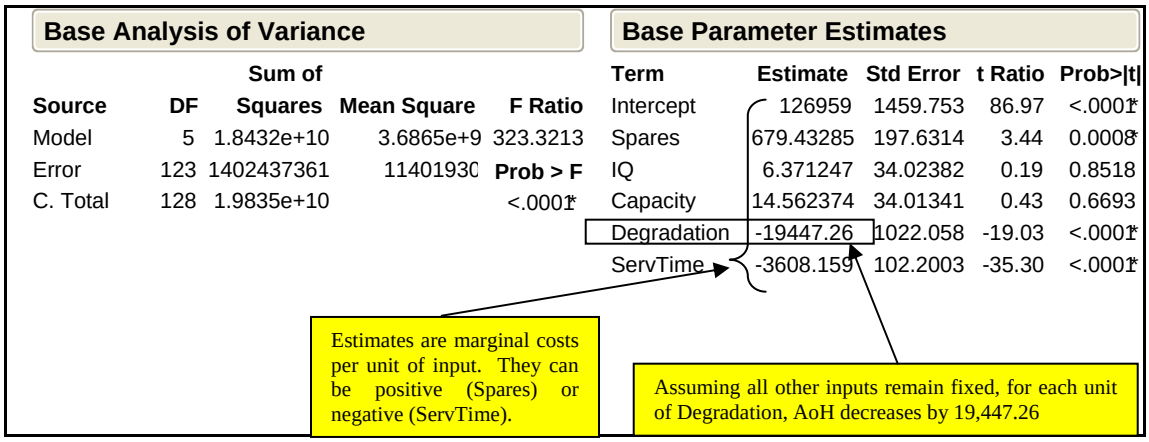

Figure 16. Baseline main parameter ANOVA and estimates

[3] The P-value is the probability, calculated assuming H_o is true, of obtaining a test statistic value at least as contradictory to H_o as the value that actually resulted. The smaller the P-value, the more contradictory is the data to H_o (Devore, 2004).

Armed with this information, a decision maker can determine that an increase in service time would have the greatest potential impact in AoH, followed by degradation and spares. The main parameter model has every attribute needed to be chosen as an acceptable metamodel for our system. Nevertheless, the residual plot shows a trend that does not fit the regression assumptions. To solve this problem, the author develops a different model, one that includes two-way interactions between parameters and quadratic terms.

In order to identify the significance of parameter two-way interactions, the author fits a polynomial regression model to the data using the JMP statistical software. The software allows the user to execute a stepwise regression, which is a method of selecting a subset of parameters to develop a good linear mathematical model that fits a data set. Analysts must be careful to balance simplicity with explanatory power, consequently, the author limits the new model to a second-degree polynomial regression model. Stepwise regression helps to select a set of factor; while a least squares linear regression is used to create the fitted metamodel.

After an iterative process of examining various models with their strengths and weaknesses, the model shown in Figure 17 is chosen. Figure 17 shows the parameters sorted in order of significance, where the first two parameters have a negative impact on AoH, and the rest have a positive impact.

Base Full Regression Summary

RSquare	0.991182
RSquare Adj	0.990749
Root Mean Square Error	1197.321
Mean of Response	91486.16
Observations (or Sum Wgts)	129

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	123023.1	484.1437	254.10	<.0001*
Spares	688.31995	70.0498	9.83	<.0001*
Degradation	-19450.96	362.3932	-53.67	<.0001*
(Degradation-1.00001)*(Degradation-1.00001)	10549.254	1399.358	7.54	<.0001*
ServTime	-3607.947	36.23828	-99.56	<.0001*
(ServTime-5.00001)*(ServTime-5.00001)	393.72058	13.99817	28.13	<.0001*
(Spares-2.50388)*(ServTime-5.00001)	110.17993	22.77668	4.84	<.0001*

Sorted Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
ServTime	-3607.947	36.23828	-99.56	<.0001*
Degradation	-19450.96	362.3932	-53.67	<.0001*
(ServTime-5.00001)*(ServTime-5.00001)	393.72058	13.99817	28.13	<.0001*
Spares	688.31995	70.0498	9.83	<.0001*
(Degradation-1.00001)*(Degradation-1.00001)	10549.254	1399.358	7.54	<.0001*
(Spares-2.50388)*(ServTime-5.00001)	110.17993	22.77668	4.84	<.0001*

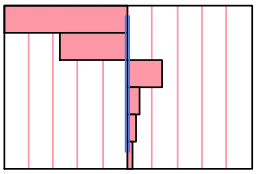

Figure 17. Selected predictive model

The analysis of variance for this model also resulted in an F-test p-value of less than .0001, once again, providing evidence that the regression model is significant. The metamodel resulted in an RSquare value of 99 percent, and an equation containing seven terms. Each factor in the model is statistically significant at less than the 1 percent level. Figure 18 shows the residual plot for the selected model. The dispersion in the plot provides enough evidence to support the normality and standard deviation assumptions embedded in regression analysis.

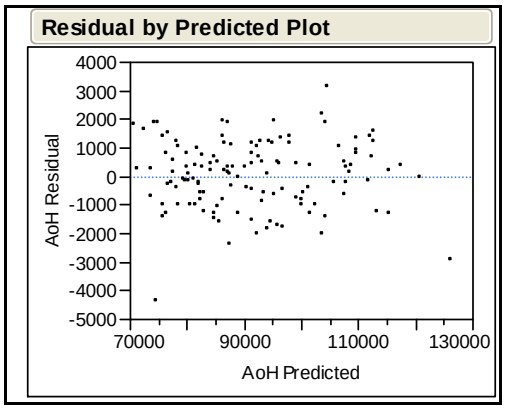

Figure 18. Selected model residual plot

A high value of RSquare can lead to a phenomenon known as overfitting a model. This phenomenon describes a situation where a model is able to explain variability, but is not able to predict. The risk of overfitting the data using this model is limited because the regression model chosen is a second-degree polynomial, and the number of terms is relatively small compared to the sample size. Figure 19 shows the AoH actual versus predicted plot, using the selected model.

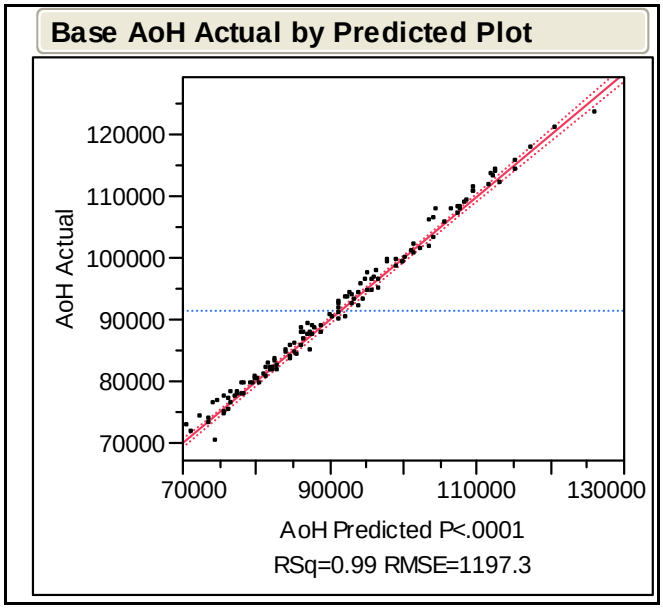

Figure 19. AoH actual versus predicted plot

One strength of the model is its ability to show relationships and interactions between important factors and AoH. The model shows that the interaction between spares and service time is significant. Figure 20 displays the interaction profiles graphically.

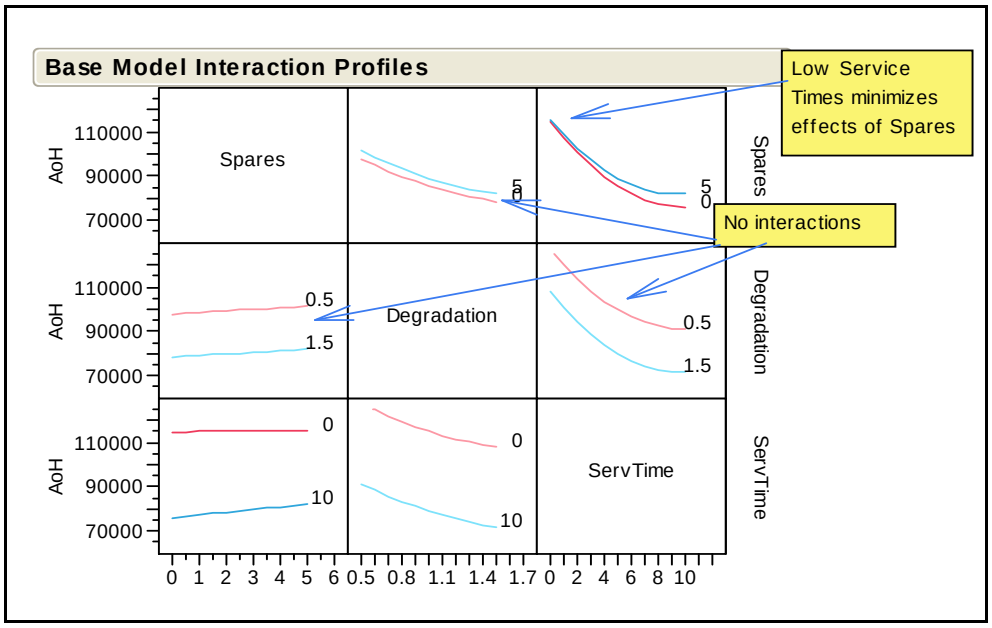

Figure 20. Main model graphical interactions

Figure 20 shows how low values of service time reduce the effect of lower spares. Such an interaction makes sense if SecReps are repaired quickly, it does not matter how many spares are in the system. Additionally, the figure shows (in broken lines) the interaction between spares and degradation. Since the lines are either close to parallel, or do not appear to ever intersect, their interaction is not significant, meaning the value in one does not significantly affect the outcome in terms of the other.

5. Partition Analysis

Now that the metamodel is selected, a partition analysis can be used to gain further insight from the data. A partition analysis is made up of successive partitions of the data according to the relationship between the predictors and the MOE values. The main benefits of this technique are that (1) it can explore relationships without the need for a parametric model, (2) it can easily handle large sets of data, and (3) the results are very easy to communicate to decision makers. Figure 21 shows the AoH data partition, and does a good job of providing a pictorial view of thresholds affecting system

performance. A quick glance allows users to discern parameter levels that can result in great or bad performance, or it can be used to look at change points or thresholds in the data.

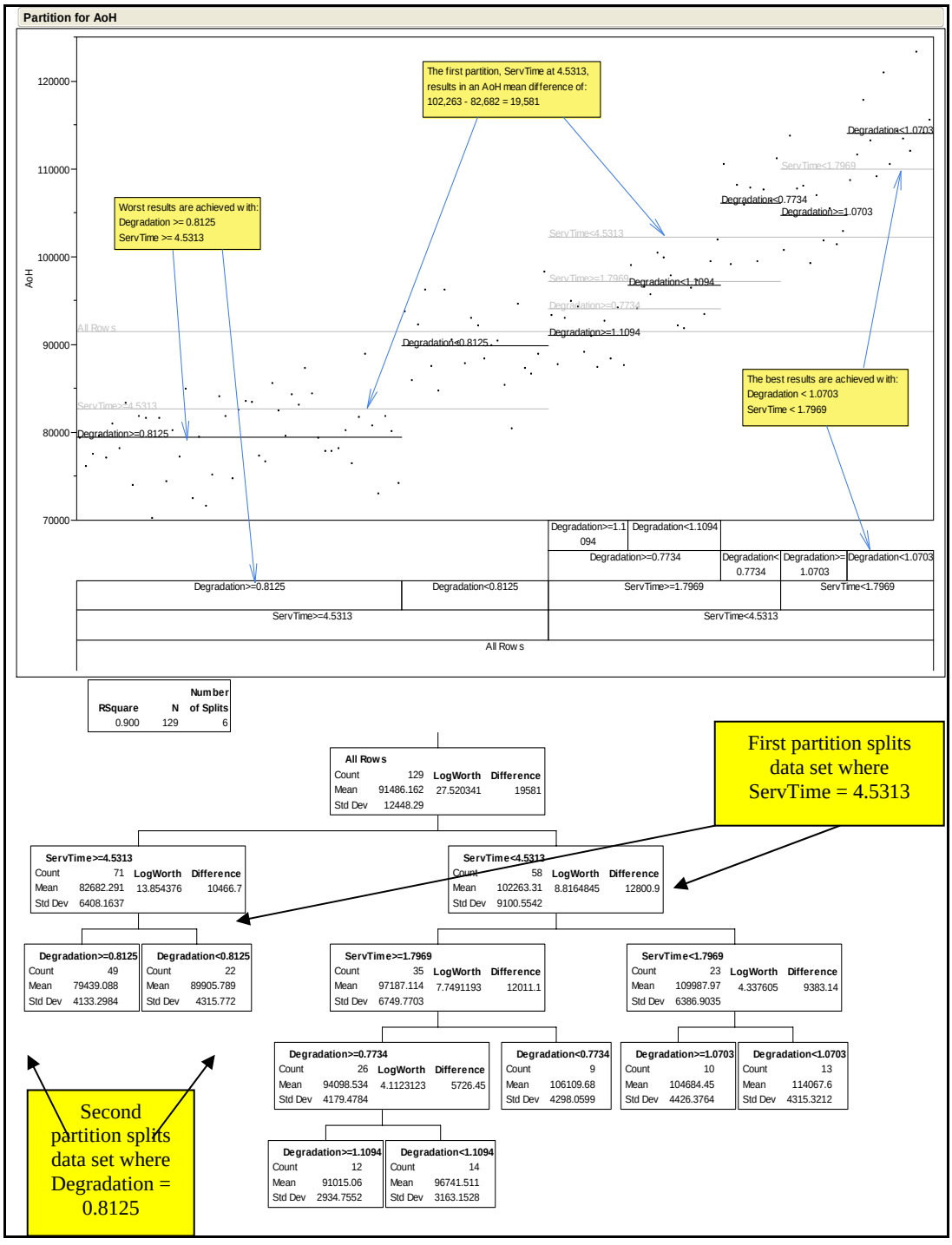

Figure 21. AoH data partition

The partition figure shows that service time is the most influential factor on the whole data set. In the first partition, the data set is divided by choosing the most influential parameter within that data set; which for this split is service time; it is split at 4.5313[4] times the baseline model service time. The two resulting data sets possess different characteristics, with sample means equal to 82,682 and 102,263 achieved operating hours (AoH), respectively. For the next partition, the system looks at both data sets in search of the most influential parameter. This time, the system finds that the partition with the smaller mean has the most influential parameter—degradation. Two new data sets are created from that partition, and the resulting means are 79,439 and 89,905 achieved operating hours, respectively. Each subsequent partition finds the most influential parameter among all existing data sets, and splits the data at the threshold point.

Analysts can use this technique to isolate points of interest in the data to determine what parameter levels cause those results. Figure 21 shows two examples of these, and the best and worst results are isolated to identify the conditions leading to such performance. In the case of best results, degradation is less than 1.07 times the baseline degradation and service time is less than 1.80 times the baseline service time. This is the kind of information that can make a difference in the development of a COA, or it could significantly help a decision maker do a better job by explicitly identifying critical performance thresholds.

In another example of how to use partition trees, the author takes a closer look at the extreme points included in COA2 AoH data set. The object is to identify thresholds conducive to such extreme behaviors. Figure 22 shows the partition tree for COA2 AoH data, where every extreme point is included on the left-most spares partition. One way to read this figure is to look at the thresholds included in that partition, i.e., every outlier takes place under the following circumstances:

- Spare < 2
- Degradation >= 0.9688
- ServTime >= 5.2344

[4] The number 4.5313 has no units; it is a factor used to adjust Service Time values used during each design. The baseline value is adjusted by multiplying it by this factor.

As an analyst, it is easy to explain to a decision maker the meaning of this partition. The portion of sample space that produced these outliers is now identified by the values above.

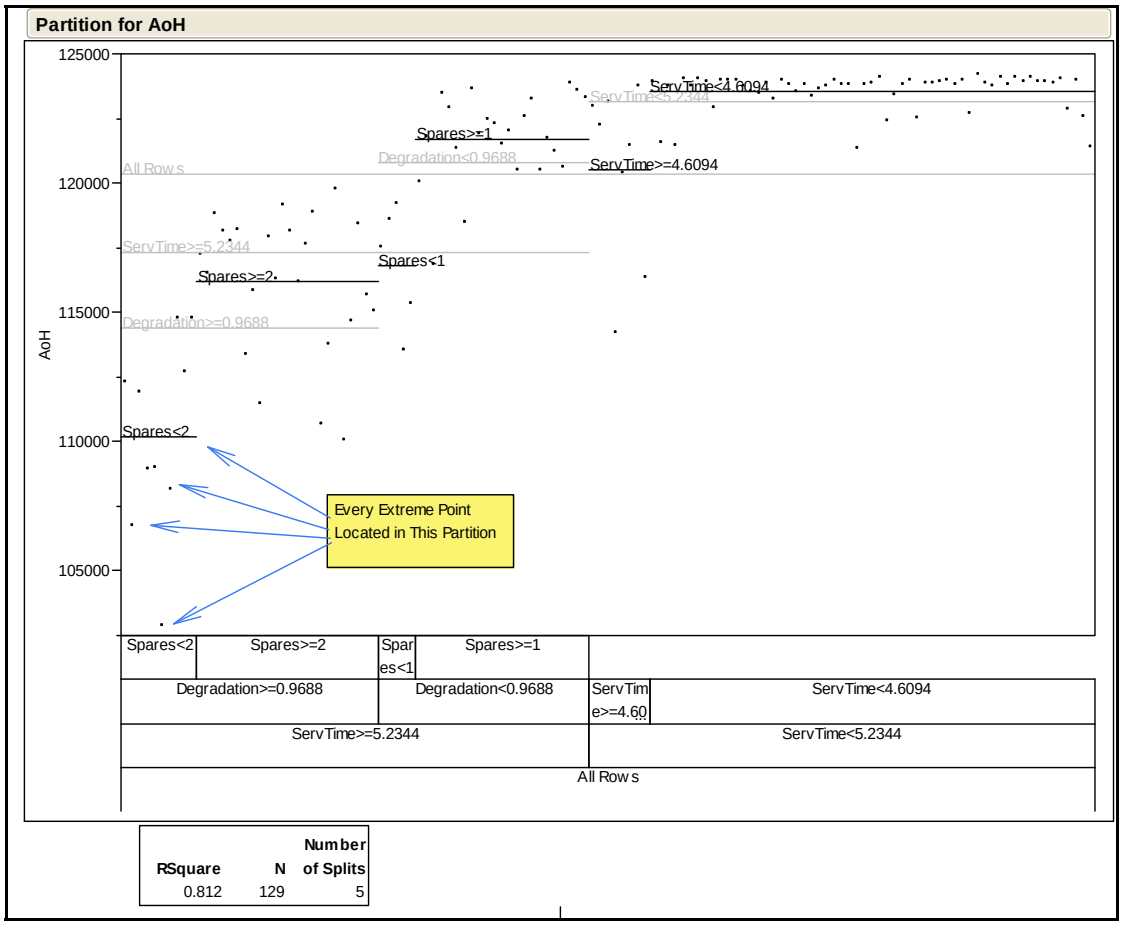

Figure 22. COA2 AoH data partition

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. RESEARCH SUMMARY

The main achievement of this effort is the development of a computer-based application capable of integrating sophisticated DOE techniques with the capabilities of TLCM-AT, in an effort to automate modeling and simulation of LCM functions. Capable of operating in a "closed-loop" form, the resulting application can execute a well-designed experiment from start to finish, without the need for any human intervention. Results presented in this thesis illustrate how employing this application can significantly increase the value of TLCM-AT to the USMC by enabling analysts to perform sensitivity analysis of proposed policies. Moreover, the research shows how the application can be used to compare the merits of different COAs, such as the comparative analysis performed in Chapter V, which can be used during development and selection of robust policies.

Users can modify the source code included in this thesis, in an effort to automate implementation of DOE, or other M&S techniques, where modifications of a database are needed to control input and output. Anyone with a good knowledge of Java programming and a strong understanding of SQL and relational databases can easily modify the application developed here for use in other efforts. There are no limits to the number of ways that the application can be used to make the most of TLCM-AT's capabilities.

Chapter V provides a simple, yet powerful, example of the kind of studies that can be done using TLCM-AT to analyze LCM functions. The analysis presented serves to demonstrate the kind of process that can be employed by decision makers to gain insights into the synergies of a fleet of systems, which could lead to better and more informed decisions, and improved system readiness. The process enhances the capabilities of TLCM-AT by applying DOE in a "closed-loop" structure, followed by the use of analytical techniques to examine the data and gain insights from it.

B. TLCM-AT

TLCM-AT is a discrete-event simulation tool developed to assist USMC program managers charged with the task of analyzing the impact of LCM decisions on fleet readiness and availability. The creators of TLCM-AT, Clockwork Solutions, adapted an existing tool in use by the United States Army to provide the USMC with the ability to predict performance metrics within operations, maintenance, and supply for a single asset or series/fleet of assets. Young (2008) performed the first exploratory analysis of TLCM-AT in a DOE environment.

Based on the findings by Young (2008) and the results of this thesis, it is apparent that TLCM-AT behaves in a rational manner, i.e., the values of the MOEs, as represented by the output data, match the expectations of an LCM professional. An inspection of the output data shows that correlations between pairs of MOEs mostly follow a logical pattern. Examples include the correlation between number of shipments (Sh) and platform events (En). It makes sense that if platform events increase, the number of shipments increases as well. The same can be concluded for the correlation between and Pt, and Sh and Pt.

At the same time, there are some unforeseen results that warrant further research. The main unexpected outcome found during this research is the behavior of Ao, which did not vary as much as expected by the author. During Young's (2008) study, the only MOE collected was Ao. He implemented DOE by modifying the same factors as in this study, but in his research, the modifications applied to 25 SecReps. During this study, the DOE modifications applied to all 175 SecReps, and it even applied to the two new and improved parts acquired for the scenarios. The author expected a significantly larger variance in Ao, but the unexpected result might be related to the way TLCM-AT measures Ao and the way the scenarios were implemented. TLCM-AT defines Ao as the number of available platforms to operate, divided by the number of assigned platforms in a given period. An available platform is one that is populated, which means all of its components are installed. According to Dr. Gurvitz (2008), the definition of populated does make a distinction between faulty and operating installed components.

Equally surprising is how Ao and AoH output is related to En. As we have seen, the simulation produces consistently high Ao whenever En has either high or low outputs. As En output moves toward the middle range, the output of Ao becomes less predictable. The same relationship exists between AoH and En. Intuition says that a low En value should relate to a high Ao and AoH value. The author could not explain why these phenomena occur.

The level of complexity in a TLCM-AT based model is worrisome. Successful implementation of a scenario requires a level of expertise that not many would have the time or motivation to acquire. This complexity is related to the level of fidelity in the model. As has been noted, a model represents most components that make up a platform. Each one is represented by a serial number and is directly tied to their parent unit, a higher assembly, or an end item. For this research, the four scenarios used were created by Dr. Peter Figliozzi, a modeler and analyst for Clockwork Solutions. Any attempt to model a scenario without the proper understanding of the TLCM-AT logic would very likely generate inaccurate results.

C. PROTOTYPE APPLICATION

TLCM-AT uses Access databases to manipulate the inputs and outputs of a model designed to represent a holistic view of a system. A typical model database consists of over 120 input and output tables, many of which contain over 240,000 line entries. Once again, the reason for this complexity is that TLCM-AT tracks each component by serial number, i.e., each end item, SecRep, and even some consumables, that make up a platform. Manipulating such a complex database is very burdensome. Any attempt to use TLCM-AT during a DOE analysis requires a method of modifying these databases sequentially for each design point. The solution is to apply the created computer application that automates database manipulation, so DOE analysis can be performed without the need for manual human interaction.

The main challenge during development of the application was mastering SQL. Anyone considering using the application must be proficient in SQL or unintended results can occur. One lesson learned was the need to understand how to identify different data types within a database, e.g., text or number, in order to produce successful SQL statements.

D. DATA ANALYSIS

The scenarios built for this effort required the deployed LAVs to operate for a number of hours, and once that goal was achieved, the platforms stopped operating. These are realistic scenarios, but they produced data sets that were of limited value. COAs 1-3 achieved their required operating hours for a significant portion of the sample space. A better approach would be to cut the number of available LAVs or significantly increase the number of required operating hours. This way, the simulation runs would produce more useful data sets, since the output would include a better breakdown of which scenarios are better than the others.

The analysis performed in Chapter V introduces an example of how analysts can use the data collected during this kind of study to provide decision makers with information critical for the development of COAs and policies. TLCM-AT is very capable of performing "what-if" scenario studies, where multiple COAs can be compared against a given MOE. The principal disadvantage of this kind of analysis is that the results of such a study are dependent on a very narrow set of circumstances. The probabilities of encountering the same set of circumstances in real life are practically zero, and therein lies the importance of DOE. A DOE analysis explores the same alternatives, but over a wide range of circumstances, enabling analysts to discern how sensitive a COA is to changes in the environment. No one can predict what precise set of circumstances real life will bring, but the goal of a well-developed DOE (one that uses reasonable parameter ranges) is to include a reasonable space of possibilities in the study, which should include the circumstances of real life. The comparative analysis performed

in Chapter V shows how simple it is to determine that COA2 is the most robust. Analysts can use the graphical representations to see how a COA performs over the whole sample space, rather than looking at a single or few "what-if" scenario study results.

Regression analysis provides analysts with insights into the inner workings of a system of systems. This study includes several regression models designed to explain the behavior of a data set; an effort that includes everything from simple regression to polynomial models. Decision makers can use regression analysis to determine which critical factors affect a system, and can interpret parameter estimates produced during regression analysis as the marginal cost, or benefit, per unit of increase in a parameter, assuming everything else remains the same. These marginal cost values help decision makers to understand the expected benefit of making a resource investment in an effort to improve system performance. The results of this analysis show that investing in ways to lower service times would likely have the greatest effect on achieved operating hours as long as the cost associated with that investment is acceptable. Maintenance managers can implement reductions in service time in different ways, including increases in capacity or personnel, better training, better tools, implementation of lean work habits, etc.

Data partition analysis is a powerful tool, capable of providing insights into a data set that other methods do not provide. Analysts can use partition analysis to isolate points of interest, outliers, best performance, worst performance, etc. The value of partition analysis is that it easily identifies the set of circumstances that lead to a particular situation. It can also be used to identify important thresholds affecting system performance. The analysis performed on the AoH data set revealed that if the value of service time is limited to less than 4.5313, the mean AoH is 102,263. On the other hand, if service time is greater than 4.5313, the mean of AoH is 82,682—a difference of over 19,000 achieved operating hours by simply limiting the service time. Another example of how analysts can use partition analysis is the result of the COA2 analysis, where all extreme values were isolated. The conditions leading to the extreme values are: spare levels fewer than two, degradation greater than 0.9688, and service time greater than 5.2344.

E. FOLLOW-ON RESEARCH

Further research using DOE and TLCM-AT needs to include cost as an MOE. Cost data is implemented in TLCM-AT differently from all the other data collected for this thesis. Any effort to use cost as part of a DOE analysis would require the assistance of Clockwork Solutions. Other research should include analysis that isolates the sample space, where the extreme values occur in the COA2 AoH data set. For this effort, high and low levels on the NOLH should be limited to those values that lead to the extreme value behavior.

APPENDIX A. JAVA APPLICATION

A. UPDATEDATABASE CLASS

1. Source Code

The *UpdateDataBase* class contains the Main Method from which the application runs. The Main Method coordinates inputs, design implementation into the working model, launches the simulation tool, and collects and saves an output file usable for later analysis. Every other Java class included in the application runs from this Main Method. The source code for the *UpdateDataBase* class follows below:

```
import java.io.*;

public class UpdateDataBase{

	public static void main(String[] args) {

		// Instantiate a List object to create an array of
		Secondary Repairables, boolean for header row or not
		List secRepList = new List("SecReps_COA3.csv", false);
		String[] secReps = secRepList.getDegraders();

		// Instantiate a List object to create an array of SRANs,
		boolean for header row or not
		List sranList = new List("srans_base.csv", false);
		String[] srans = sranList.getDegraders();

		// Instantiate a NOLH object to create a design array,
		boolean for header row or not, int is number of factors
		NOLH designList = new NOLH("ThesisHOLH.csv", false, 5);
		String[][] design = designList.getDesign();

		try{
			PrintWriter outputStream = new PrintWriter(new
			FileOutputStream("TLCM-AT_Results.csv"));
			outputStream.append("Design ,Spares ,IQ ,Capacity
			,Degradation ,ServTime ,Ao ,AoH ,En ,SBn ,Sh ,MTBF
			,Pt");//Printing output headings
			outputStream.println();

		for (int x = 0; x < 129; x++) {

			// Create a copy of the source file using CopyFile
			class
```

```
CopyFile file = new CopyFile("LAV 3042 v5 jungle COA 3 no out.mdb", "C:\\Program Files\\Clockwork Solutions\\TLCM-AT 5.2\\smalldb1.mdb");
file = null;

//Instantiate an UpdateSpares object to modify spares levels for a given design
UpdateSpares spare = new UpdateSpares("jdbc:odbc:smalldb1", "sun.jdbc.odbc.JdbcOdbcDriver", design, secReps, x, srans);
spare.doUpdate();
spare = null;

// Instantiate an UpdateAbilityToRepair object to modify ability to repair at the depot level
UpdateAbilityToRepair repair = new UpdateAbilityToRepair("jdbc:odbc:smalldb1", "sun.jdbc.odbc.JdbcOdbcDriver", design, x);
repair.doUpdate();
repair = null;

// Instantiate an UpdateServerTimes object to modify length of repairs for each degrader
UpdateServerTimes serv = new UpdateServerTimes("jdbc:odbc:smalldb1", "sun.jdbc.odbc.JdbcOdbcDriver", design, x);
serv.doUpdate();
serv = null;

// Instantiate an UpdateCapacity object to modify capacity constraints for repairs of each degrader
UpdateCapacity cap = new UpdateCapacity("jdbc:odbc:smalldb1", "sun.jdbc.odbc.JdbcOdbcDriver", design, x);
cap.doUpdate();
cap = null;

// Instantiate an UpdateRepairDeg object to modify LRUs' unscheduled removal rates
UpdateRepairDeg deg = new UpdateRepairDeg("jdbc:odbc:smalldb1", "sun.jdbc.odbc.JdbcOdbcDriver", design, x);

deg.doUpdate();
deg = null;
System.gc();

RunProgram run = new RunProgram("C:\\Program Files\\Clockwork Solutions\\TLCM-AT 5.2\\modelr.exe -30");
run.run();
run = null;

// Instantiate an UpdateOutput object to collect the
```

```
                        results from the database
                        UpdateOutput     result     =     new     UpdateOutput
                        ("jdbc:odbc:smalldb1",
                        "sun.jdbc.odbc.JdbcOdbcDriver");
                        int n = x+1;
                        outputStream.append(n+" ,"+design[x][0]+"
                        ,"+design[x][1]+"  ,"+design[x][2]+"  ,"+design[x][3]+"
                        ,"+design[x][4]+" ,");
                        double[] stat = new double[7];
                        stat = result.getResult();
                        for(int i = 0; i < 7; i++){
                              outputStream.append(stat[i]+" ,");
                        }

                        outputStream.println();
                        System.gc();
                        int r = x+1;;
                        System.out.println("Completed design number "+r);
                  }

                  outputStream.close();
                  }
                  catch (FileNotFoundException e) {
                        System.out.println("File not found.");
                  }
            }// End of Main Method
}// End of UpdateDataBase Class
```

2. How It Works

The application instantiates a *List* object using as an argument a .csv file listing all SecReps and a boolean expression determined by the presence of a header row in the input list. *List* takes the .csv file and transforms it into an array of SecReps. To develop the SecRep .csv file, users can query the **Object type* table within the model being used using the criteria greater than 700000000 and less than 900000000 for *Object type*. For the list of SRANs the process is very similar. Users can create the SRAN .csv file by looking at the **Base Names* table within the model. The author's list includes only the MEF and jungle bases.

Using the *Orthogonal and Nearly Orthogonal LH Worksheet* file (Sanchez, 2005), users can develop a .csv file with a NOLH design. The application instantiates a *NOLH* object using the .csv file, a boolean expression determined by the presence of header row on the file, and an integer representing the number of factors varied in the design. The result is a two-dimensional array containing the DOE to be used in the simulation.

Java uses the *PrintWriter* package to create an output file, which, in this case, is saved as *TLCM-AT_Results.csv*. The *PrintWriter* object appends the header row to the output as follows: *Design ,Spares ,IQ ,Capacity ,Degradation ,ServTime ,Ao ,AoH ,En ,SBn ,Sh ,MTBF ,Pt*. These headers include the design number, the input parameters as listed on the DOE design array, and the MOE's outputs.

A *for* loop iterates over every design point in the design array; it is important to set the loop to run as long as the number of design points. A *CopyFile* object creates a working copy of the model being used in our simulation. Copying the file has two purposes: (1) to preserve the baseline model intact so subsequent designs are easier to implement, and (2) to create a working model named *smalldb1.mdb,* which is the only possible file name to be used in our simulation. TLCM-AT only uses that file name whenever the tool is launched from the command line; the only available argument for command line operation is the number of histories. Also, the file needs to be located in the same directory as the tool's executable file.

At this point, Java updates the working file one parameter at a time. Java uses *UpdateSpares, UpdateAbilityToRepair, UpdateServerTimes, UpdateCapacity, and UpdateRepairDegradation* to perform each parameter adjustment. In each case, the object created performs the *doUpdate()* method to complete the update. Each class has a slightly different set of arguments, but these are discussed below. The application launches the tool by instantiating a *RunProgram* object. *RunProgram* objects take as arguments the command line statement used to launch TLCM-AT. The number at the end of the command line argument, in this case 30, is where users determine the number of replications to run for each design.

An *UpdateOutput* object takes the working model to extract the data used in calculating the MOEs of interest. The *PrintWriter* object adds the design parameter levels to the output file, followed by the MOE values. *UpdateOutput* object creates these values and saves then into an array of size seven. The application retrieves the array into a newly created one and uses a *for* loop to populate the output file. At this point, Java continues to iterate through every design point using the main *for* loop. After every design has been simulated, the application closes the *PrintWriter* object and saves the output file.

B. LIST CLASS

1. Source Code

The List class takes any set of items listed in a .csv file and produces a single-dimension array containing the listed items. Our Java application uses List to convert the SecRep and SRAN .csv files into arrays for use as inputs for database modification. The source code for the List class follows below:

```
import java.io.*;

public class List{

	private int numItems;
	private String[] list;

	public List (String file, boolean header){//Constructor
		numItems = countFileLength(file, header);// Count number of
		records to size the array
		list = populateList(file, header);
	}

	public int countFileLength(String filename, boolean
	hasHeaderRow){
		int count = 0;
		try{
			BufferedReader inputStream = new BufferedReader(new
			FileReader(filename));
			String line = inputStream.readLine( );
			if ((line != null) && hasHeaderRow)
				line = inputStream.readLine( ); // skip to next
				line without counting

			while (line != null){
				count++;
				line = inputStream.readLine();
			}
			inputStream.close();
		}
		catch(FileNotFoundException e){
			System.err.println("File opening problem.");
			System.err.println(e.getMessage());
			System.exit(-1);
		}
		catch(IOException e){
			System.out.println("Error reading from file
			"+filename);
			System.out.println(e.getMessage());
			System.exit(-1);
		}
```

```
            return count;
      }//End of countFileLength

      public int getNumItems(){
            return numItems;
      }

      public String[] populateList(String filename, boolean header) {
            String[] items = new String[numItems];
            try   {
                  BufferedReader inputStream = new BufferedReader(new
                  FileReader(filename));
                  String line = inputStream.readLine( );
                  // if header row, skip the first line, by reading the
                  next
                  if (header) {
                        line = inputStream.readLine();
                  }
                  for(int x = 0; x < numItems; x++){
                        items[x] = line;
                        line = inputStream.readLine( );
                  }
                  inputStream.close( );
            }
            catch(FileNotFoundException e){
                  System.err.println("File opening problem.");
                  System.err.println(e.getMessage());
                  System.exit(-1);
            }
            catch(IOException e){
                  System.err.println("Error    reading    from    file
                  "+filename);
                  System.err.println(e.getMessage());
                  System.exit(-1);
            }
            return items;
      }// End of populateList method

      public String[] getDegraders() {
            return list;
      }

}//End of List Class
```

2. How It Works

The constructor for *List* takes as parameters a String object and a boolean expression. The String object contains a single column .csv file listing the items to be included in the output array. The boolean expression determines the existence of a header row in the input file. There are two instance variables in this class: (1) int

numItems and (2) String [] *list*. Using the method *countFileLength*, the constructor sets the value of *numItems*. Subsequently, the constructor invokes *populateList* to create the output array.

The *countFileLength* method takes the same parameters as the constructor. A *BufferedReader* object reads the input file. The method uses an *if* statement to skip a line if a header row exists and a *while* loop to count the number of items on the list. After every line is counted, *countFileLength* returns an integer object representing the number of items on the input file.

Once again, *populateList* takes the same parameters as the constructor. The method starts by declaring a String array and uses the value of *numItems* to determine the size of the array. The rest of the method is very similar to *countFileLength*, except that in this case, it is populating the array instead of counting line items. Furthermore, the method uses a *for* instead of a *while* loop because the number of items in the list is known.

C. NOLH CLASS

1. Source Code

The *NOLH* class performs the same function as the *List* class except that it produces a two- versus a single-dimension array. *NOLH* reads a .csv file representing every design point on a DOE and creates a two-dimensional array to be used as input during database modification. The source code for the *NOLH* class follows below:

```
import java.io.*;
import java.util.StringTokenizer;

public class NOLH {

	private String[][] design;
	private int count;
	private int numFactors;

	public	NOLH	(String	file,	boolean	header,	int
	factors){//Constructor
		numFactors = factors;
		count = countFileLength(file, header);
		design = new String[count][5];
```

```
        readDesign(file, header); //To populate design
    }

    public static int countFileLength(String filename, boolean
    hasHeaderRow) {
        int count = 0;
        try {
            BufferedReader inputStream = new BufferedReader(new
            FileReader(filename));
            String line = inputStream.readLine( );
            if ((line != null) && hasHeaderRow)
                line = inputStream.readLine( ); // skip to next
                line without counting
            while (line != null){
                count++;
                line = inputStream.readLine( );
            }
            inputStream.close( );
        }catch(FileNotFoundException e){
            System.err.println("File opening problem.");
            System.err.println(e.getMessage());
            System.exit(-1);}
        catch(IOException e){
            System.out.println("Error reading from file
            "+filename);
            System.out.println(e.getMessage());
            System.exit(-1);}

        return count;
    }//End of countFileLength

    public void readDesign(String filename, boolean header) {

        try {
            BufferedReader inputStream = new BufferedReader(new
            FileReader(filename));
            String line = inputStream.readLine( );

            // if header row, skip the first line, by reading the
            next
            if (header) {
                line = inputStream.readLine();
            }
            String delim = ",";
            for(int x = 0; x < count; x++){
                StringTokenizer parser = new
                StringTokenizer(line, delim);
                for (int y = 0; y < numFactors; y++){
                    design[x][y] = parser.nextToken();
                }
                line = inputStream.readLine();
            }
            inputStream.close( );
        }
        catch(FileNotFoundException e)
```

```
            {
                  System.err.println("File opening problem.");
                  System.err.println(e.getMessage());
                  System.exit(-1);
            }
            catch(IOException e)
            {
                  System.err.println("Error    reading    from    file
                  "+filename);
                  System.err.println(e.getMessage());
                  System.exit(-1);
            }
      }//End of readDesign

      public String[][] getDesign() {
            return design;
      }

      public int getCount() {
            return count;
      }
}//End of NOLH Class
```

2. How It Works

NOLH's constructor uses three parameters: a String object, a boolean expression, and an integer number. The String object contains the DOE information on a .csv file using rows to represent design points and columns to represent levels of varying factors. A booloean expression determines the existence of a header row in the file, while the integer represents the number of factors being modified.

NOLH has three instance variables: (1) String [] [] *design*, (2) int *count*, and (3) int *numFactors*. The constructor's integer parameter sets the value of *numFactors*. Using the boolean expression and the String object, the constructor invokes the *countFileLength* method to determine the number of design points on the DOE and sets the value of *count*. With the known value of *count*, the constructor sets enough memory aside to populate the *design* array and the *readDesign* method populates it.

D. NOLH CLASS

1. Source Code

This class is designed using information posted on the Sun Developer Network (2007) Website as a template. It copies any given file and saves it under a given name. The application uses this class to make a copy of the baseline model and saves it as the working model on the same directory of the TLCM-AT tool. The source code for the *CopyFile* class follows below:

```
import java.io.*;

public class CopyFile {

      public CopyFile(String srFile, String dtFile){
          try{
            File f1 = new File(srFile);
            File f2 = new File(dtFile);
            InputStream in = new FileInputStream(f1);
            OutputStream out = new FileOutputStream(f2);
            byte[] buf = new byte[1024];
            int len;
            while ((len = in.read(buf)) > 0){
              out.write(buf, 0, len);
            }
            in.close();
            out.close();
            buf = null;
            System.gc();
            f1 = null;
            f2 = null;
          }
          catch(FileNotFoundException ex){
            System.out.println(ex.getMessage() + " in the specified
            directory.");
            System.exit(0);
          }
          catch(IOException e){
            System.out.println(e.getMessage());
          }
        }
}
```

2. How It Works

The constructor takes two parameters: a String object representing the source file name and directory, and a String object representing the destination file name and

directory. If the directory is the same as the directory of the Java program, there is no need to include directory information on the parameter. This program copies a file regardless of file format; the copy is made byte by byte.

E. UPDATESPARES CLASS

1. Source Code

UpdateSpares class controls the number of spares that are added to each experiment, based on a given design point. Spares are never removed from a baseline model, the class only adds spares. To accomplish this, *UpdateSpares* inserts a number of objects into the **Object attributes initial* table. The application has to insert each object into the table with its own object type, serial number, and location where the spare will be stored. The source code for the *UpdateSpares* class follows below:

```
import java.sql.*;
import java.util.*;

public class UpdateSpares{

	private String dBurl;
	private String dBdriver;
	private String[][] des;
	private String[] secRep;
	private int exp;
	private String[] srans;

	public UpdateSpares(String dBurl, String dBdriver, String[][]
	des, String[] secReps, int exp, String[] sranList){
		this.dBurl = dBurl;
		this.dBdriver = dBdriver;
		this.secRep = secReps;
		this.des = des;
		this.exp = exp;
		this.srans = sranList;
	}

	/**
	 * @param args
	 */
	public void doUpdate(){

		try{
			Class.forName(dBdriver);
			Connection                connection                =
			DriverManager.getConnection(dBurl);
```

```
                  int serNo = 1000000;

                  // Create an array of objects and SRAN combinations

                  String [][] newSpares = new
                  String[srans.length*secRep.length][2];
                  int arrayRow = 0;
                  for(int x = 0; x < secRep.length; x++){
                        for(int y = 0; y < srans.length; y++){
                              newSpares[arrayRow][0] = secRep[x];
                              newSpares[arrayRow][1] = srans[y];
                              arrayRow++;
                        }
                  }

                  // Insert new spares in *Object attributes initial
                  table
                  arrayRow = 0;
                  PreparedStatement addSecReps;
                  for(int x = 0; x < Integer.valueOf(des[exp][0]); x++){
                  //Iterate over SecReps
                        for(int y = 0; y < newSpares.length; y++){ //
                        Iterate over number of spares per SecReps
                        addSecReps = connection.prepareStatement("INSERT
                        INTO [*Object attributes initial] ([Object ID],
                        [Object type], [AFHRv], [AFHRn], [Atacn],
                        [Atacv], [Aeotn], [Aeotv], [Awown], [Awowv],
                        [Parent Pp], [F], [Arrival time ta],
                        "[SRAN], [Completed Repairs], [Probabilistic
                        age], [TreeCode], [TreeParent]) VALUES
                        ('"+serNo+++"', "+newSpares[y][0]+", 0, 0, 0, 0,
                        0, 0, 0, 0, 1, 0, "+newSpares[y][1]+", 0, 0,
                        '0', '0')");
                        addSecReps.executeUpdate();
                        addSecReps = null;
                        }
                  }
                  newSpares = null;
                  connection.close();
                  connection = null;

            }// End of Try Loop
            catch (ClassNotFoundException cnfe){
                  System.err.println(cnfe);
            }
            catch (SQLException sqle) {
                  System.out.println(sqle);
                  while(sqle != null){
                        sqle = sqle.getNextException();
                        System.err.println(sqle);}
            }
      }//End of doUpdate
}//End of Class
```

2. How It Works

UpdateSpares's constructor takes six parameters: (1) a String object representing a Uniform Resource Locator (URL) for the data source, (2) a String object representing a URL for the ODBC driver, (3) a two-dimension array of String objects representing the DOE, (4) a single-dimension array of String objects representing the list of SecReps, (5) an integer number representing the design point, and (6) a single-dimension array of String objects representing the list of SRANs. All six instance variables take their corresponding value from one of the parameters mentioned above within the constructor.

The class implements one *void* method, *doUpdate()*, which performs all the functions required from this class. It is important to note here, that before trying to use this class, users need to set up an Access database driver using the ODBC Data Source Administrator dialog box as explained in Chapter III, Section D. To load the desired ODBC driver, the application invokes the method *forName()* from the *Class* class using the value of *dBdriver* as the argument. Using a *Connection* object, users can maintain a connection to the database of interest, such connection is critical prior to executing any SQL statements. The application uses the integer value *serNo* to assign individual serial numbers to each new spare.

UpdateSpares creates an array of objects and SRAN combinations; each SecRep is matched with each SRAN. The array *newSpares* is of size equal to the number of SecReps, times the number of SRANs included on the analysis. The application instantiates a *PreparedStatement* object using the *Connection* object created at the beginning. The argument for the *Connection* class *prepareStatement()* method is the SQL statement. During this process, the application iterates over the individual SecRep and SRAN combinations using the inner *for* loop, and over the number of spares to add using the outer *for* loop. The *PreparedStatement* object has to invoke its *executeUpdate()* method to complete each SQL statement.

F. UPDATEABILITYTOREPAIR CLASS

1. Source Code

UpdateAbilityToRepair class implements the changes to the Depot spares program according to the values set forth by the design point. These changes take place within the **Depot Spares Program* table. For each scenario, the application updates the number of parts that can be inducted into the different Depots per quarter. The source code for the *UpdateAbilityToRepair* class follows below:

```
import java.sql.*;

public class UpdateAbilityToRepair{

	private String dBurl;
	private String dBdriver;
	private String[][] des;
	private int exp;

	public UpdateAbilityToRepair(String dBurl, String dBdriver,
	String[][] des, int exp){
		this.dBurl = dBurl;
		this.dBdriver = dBdriver;
		this.des = des;
		this.exp = exp;
	}

	public void doUpdate() {
		try {
			Class.forName(dBdriver);
			Connection connection =
			DriverManager.getConnection(dBurl);

			//Perform updates
			String updateRepairAbility = "UPDATE [*Depot Spares
			Program] SET [Quantity] = "+des[exp][1]+" WHERE
			[Object Type]> 700000 AND [Object Type]< 900000";
			PreparedStatement updateLevels =
			connection.prepareStatement(updateRepairAbility);
			updateLevels.executeUpdate();
			updateRepairAbility = null;
			updateLevels = null;

			connection.close();
			connection = null;
			des = null;
		}
		catch (ClassNotFoundException cnfe) {
			System.err.println(cnfe);}
		catch (SQLException sqle) {
```

```
                    System.err.println(sqle);}
        }//End of doUpdate
}//End of Class
```

2. How It Works

The constructor receives four parameters: (1) a String object representing a URL for the data source, (2) a String object representing a URL for the ODBC driver, (3) a two-dimension array of String objects representing the DOE, and (4) an integer number representing the design point. All four instance variables take their corresponding value from one of the parameters mentioned above within the constructor. *UpdateAbilityToRepair* implements the *doUpdate()* method and is responsible for performing every function within this class.

The *doUpdate()* method starts with the identification of the data source and creation of a connection, just like in the *UpdateSpares* class. The application instantiates a *PreparedStatement* object using the *Connection* object created at the beginning. The argument for the *Connection* class *prepareStatement()* method is the SQL statement. Later, the *PreparedStatement* object executes the update. Notice how, in this case, there are no loops involved; Java is able to perform the update all at once, using the specially created SQL statement.

G. UPDATESERVERTIMES CLASS

1. Source Code

The *UpdateServerTimes* class updates the server times associated with repairing each SecRep by multiplying the current value by the value on the design point. As expected, these updates take place in the **Server times* table. The source code for the *UpdateServerTimes* class follows below:

```
import java.sql.*;
import java.text.DecimalFormat;

public class UpdateServerTimes{

        private String dBurl;
        private String dBdriver;
        private String[][] des;
```

```
private int exp;

public UpdateServerTimes(String dBurl, String dBdriver, String[][]
des, int exp){
      this.dBurl = dBurl;
      this.dBdriver = dBdriver;
      this.des = des;
      this.exp = exp;
}

public void doUpdate () {

            try {
            Class.forName(dBdriver);
            Connection                connection                =
            DriverManager.getConnection(dBurl);

            PreparedStatement            currentvals            =
            connection.prepareStatement ("SELECT [Object type],
            [SRAN ID], [SERVER TYPE], [Tsf P1] FROM [*Server
            times] WHERE [Object type]> 700000000 AND [Object
            Type]< 900000000");
            ResultSet curValues = currentvals.executeQuery();

            while(curValues.next()) {
                  DecimalFormat         form        =         new
                  DecimalFormat("0.0000");
                  String                  var                 =
                  form.format(Double.valueOf(curValues.getString(4
                  ))*Double.valueOf(des[exp][4]));

                  String updateServerTimes = "UPDATE [*Server
                  times] SET [Tsf P1] = "+var+" WHERE [Object
                  type] = "+curValues.getString(1)+" AND [SERVER
                  TYPE]= '"+curValues.getString(3)+"' AND [SRAN
                  ID] = "+curValues.getString(2);
                  PreparedStatement         updateLevels        =
                  connection.prepareStatement(updateServerTimes);
                  updateLevels.execute();
                  updateServerTimes = null;
                  updateLevels = null;
                  form = null;
                  var = null;
            }

            curValues = null;
            currentvals = null;
            connection.close();
            connection = null;
            des = null;
      }
      catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe);
      }
      catch (SQLException sqle) {
```

```
                System.err.println(sqle);
            }
        }//End of doUpdate
}//End of Class
```

2. How It Works

The implementation of *UpdateServerTimes* is very similar to *UpdateAbilityToRepair*. The constructor receives four parameters: (1) a String object representing a URL for the data source, (2) a String object representing a URL for the ODBC driver, (3) a two-dimension array of String objects representing the DOE, and (4) an integer number representing the design point. All four instance variables take their corresponding value from one of the parameters mentioned above within the constructor. *UpdateServerTimes* implements the *doUpdate()* method and is responsible for performing every function within this class.

The *doUpdate()* method starts with the identification of the data source and creation of a connection, just like in the *UpdateSpares* class. The application instantiates a *PreparedStatement* object using the *Connection* object created at the beginning. The argument for the *Connection* class *prepareStatement()* method is an SQL statement designed to query the current server times. The remainder of the operation consists of a *while* loop, which controls each required update.

A String object takes the value of the current server time, multiplied by the design point value. The *Double* class method *valueOf()* converts the String objects to Double objects so the multiplication can take place. A new *PreparedStatement* object performs the update using the newly created SQL statement and invoking the *execute()* method.

H. UPDATESERVERTIMES CLASS

1. Source Code

The class *UpdateCapacity* modifies the number of LRUs that an I-level facility can process at a time. This modification takes place on the **Capacity* table and simulates investments in I-level capacity, including improvements or reductions in personnel and

infrastructure. Our baseline LAV model currently models unlimited I-level capacity; this Java class inserts those limits according to the values included in the DOE. The source code for the *UpdateCapacity* class follows below:

```
import java.sql.*;

public class UpdateCapacity{

	private String dBurl;
	private String dBdriver;
	private String[][] des;
	private int exp;

	public UpdateCapacity(String dBurl, String dBdriver, String[][]
	des, int exp){
		this.dBurl = dBurl;
		this.dBdriver = dBdriver;
		this.des = des;
		this.exp = exp;
	}

	public void doUpdate() {

		try {
			Class.forName(dBdriver);
			Connection connection =
			DriverManager.getConnection(dBurl);

			String SQLStatements = "INSERT INTO [*Capacity]
			VALUES('20000', '-1', '2', '"+des[exp][2]+"')";   //
			The 20000 above signifies that all capacity levels
			happen at the I-Level
			Statement statement = connection.createStatement();
			statement.executeUpdate(SQLStatements);
			statement = null;
			SQLStatements = null;

			connection.close();
			connection = null;
			des = null;
		}
		catch (ClassNotFoundException cnfe){
			System.err.println(cnfe);
		}
		catch (SQLException sqle){
			System.err.println(sqle);
		}
	}//End of doUpdate
}//End of Class
```

2. How It Works

The implementation of *UpdateCapacity* is very similar to *UpdateAbilityToRepair*. The constructor receives four parameters: (1) a String object representing a URL for the data source, (2) a String object representing a URL for the ODBC driver, (3) a two-dimension array of String objects representing the DOE, and (4) an integer number representing the design point. All four instance variables take their corresponding value from one of the parameters mentioned above within the constructor. *UpdateCapacity* implements the *doUpdate()* method and is responsible for performing every function within this class.

The *doUpdate()* method starts with the identification of the data source and creation of a connection, just like in the *UpdateSpares* class. This *UpdateCapacity* implementation applies the same limit to every I-level facility; as a result, the process is very simple and does not require the use of any loops. A String object is created to represent the SQL statement. The **Capacity* table contains four columns: *SRAN ID, Group Code, Ind level,* and *Cap.* A *20000* in the *SRAN ID* signifies that all I level have the same limit; the *-1* is a special entry meaning that the capacity limit applies to all LRUs. The *2* signifies I level and the last value is the new limit, represented by the design array. A *Statement* object uses the SQL as argument to execute the update.

I. UPDATEREPAIRDEG CLASS

1. Source Code

The *UpdateRepairDeg* class modifies the unscheduled removal rates of every SecRep. The Java implementation performs the update by multiplying the given value on the DOE by the current value on the baseline model. Performing these adjustments can model improvements in maintenance practices, or it can be used to model LRU reliability improvements. The source code for the *UpdateRepairDeg* class follows below:

```
import java.sql.*;

public class UpdateRepairDeg{

	private String dBurl;
	private String dBdriver;
	private String[][] des;
	private int exp;

	public UpdateRepairDeg(String dBurl, String dBdriver, String[][]
	des, int exp){
		this.dBurl = dBurl;
		this.dBdriver = dBdriver;
		this.des = des;
		this.exp = exp;
	}

	public void doUpdate() {
		try {
			Class.forName(dBdriver);
			Connection connection =
			DriverManager.getConnection(dBurl);

			PreparedStatement currentvals =
			connection.prepareStatement
			("SELECT [LRU type], [Platform type], [Base],
			[Completed Repairs], [Age Unit], [Rate], [Shape] FROM
			[*Unscheduled Removal rates] WHERE [LRU type] >
			700000000 AND [LRU type] < 900000000");
			ResultSet curValues = currentvals.executeQuery();

			while(curValues.next()) {
			  String var = String.valueOf (Double.valueOf
			  (curValues.getString(6))*Double.valueOf(des[exp][3])
			  );
			  String SQLStatements = "UPDATE [*Unscheduled Removal
			  rates] SET [Rate] = "+var+" WHERE [LRU type] =
			  "+curValues.getString(1)+" AND [Platform type] =
			  "+curValues.getString(2)+" AND [Base] =
			  "+curValues.getString(3)+" AND [Completed Repairs] =
			  "+curValues.getString(4)+" AND [Age Unit] =
			  '"+curValues.getString(5)+"' AND [Shape] =
			  "+curValues.getString(7);
			  Statement statement = connection.createStatement();
			  statement.executeUpdate(SQLStatements);
			  var = null;
			  statement = null;
			  SQLStatements = null;
			}

			connection.close();
			connection = null;
			des = null;
		}
		catch (ClassNotFoundException cnfe) {
```

```
                System.err.println(cnfe);}
            catch (SQLException sqle) {
                        System.err.println(sqle);}
      }//End of doUpdate
}//End of Class
```

2. How It Works

The implementation of *UpdateRepairDeg* is very similar to *UpdateAbilityToRepair*. The constructor receives four parameters: (1) a String object representing a URL for the data source, (2) a String object representing a URL for the ODBC driver, (3) a two-dimension array of String objects representing the DOE, and (4) an integer number representing the design point. All four instance variables take their corresponding value from one of the parameters mentioned above within the constructor. *UpdateRepairDeg* implements the *doUpdate()* method and is responsible for performing every function within this class.

The *doUpdate()* method starts with the identification of the data source and creation of a connection, just like in the *UpdateSpares* class. Java instantiates a *PreparedStatement* object using an SQL as argument. The SQL statement queries the **Unscheduled Removal rates* table for every SecRep related entry by using the criteria *[LRU type] > 700000000 AND [LRU type] < 900000000*. A *while* loop controls the update since each entry is done individually. Java multiplies the existing *Rate* value for each SecRep by the value on the DOE, and the result is used as the new *Rate* on the **Unscheduled Removal rates* table.

J. RUNPROGRAM CLASS

1. Source Code

RunProgram does exactly that; it takes an executable file as an argument and launches whatever application is associated with it. The source code for the *RunProgram* class follows below:

```
public class RunProgram{

      private String fileLoc;
```

```
	public RunProgram(String fileLocation) {
		fileLoc = fileLocation;
	}

	public void run(){
		try {
			Runtime rt = Runtime.getRuntime();
			// This will launch the .exe file included in the
			argument
			Process p = rt.exec(fileLoc);
			System.out.println("TLCM-AT ended properly
			"+p.waitFor());
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}
}
```

2. How It Works

The constructor takes a String object that represents the location and name of an executable file. The instance variable *fileLoc* takes the value of the argument on the constructor.

The *run()* method launches the application using a *Runtime* object, which allows the Java application to interface with Windows XP. *System.out.println("TLCM-AT ended properly "+p.waitFor())* gives the user an indication that the application completed successfully. In case of successful completion, the text "*TLCM-AT ended properly 123456789*" shows up on the screen. The method *waitFor()* causes the current thread to wait, if necessary, until the process represented by this *Process* object has terminated.

K. UPDATEOUTPUT CLASS

1. Source Code

UpdateOutput processes all the output requirements for use in our analysis. The list of MOEs retrieved from the model after each simulation run includes:

- Availability (Ao) – Systems availability percentage for a period of time
- Achieved Operating Hours (AoH) – Achieved operating hours
- Events (En) – Number platform events due to failures
- New Spare Buys (SBn) – Number of new spare buys

- Shipments (Sh) – Number of shipments between bases
- Mean Time Between Failures (MTBF) – Ratio between total achieved operating hours and platform events due to failures
- Task Performed (Pt) – Number of tasks performed by all levels

The class makes the connections to the model database to retrieve the data and uses the *SimpleStats* class to perform the statistical processes. The source code for the *UpdateOutput* class follows below:

```
import java.sql.*;

public class UpdateOutput {

	private double[] result = new double[7];

	public UpdateOutput(String dBurl, String dBdriver){
		doUpdate(dBurl, dBdriver);
	}

	public void doUpdate(String dBurl, String dBdriver) {

		try {
			Class.forName(dBdriver);
			Connection connection =
			DriverManager.getConnection(dBurl);

			SimpleStats stat =  new SimpleStats();
			int year = 2009;
			int quarter = 4;

			//Calculate Ao for Quarter 12
			PreparedStatement currentvals =
			connection.prepareStatement ("SELECT [SRAN], [Type],
			[Year], [Qtr], [Week],[Availability] FROM [out
			Availability] WHERE [Year]= "+year+" AND [QTR] ="+
			quarter);
			ResultSet curValues = currentvals.executeQuery();

			while(curValues.next()) {
				double s = curValues.getDouble(6);
				stat.newobs(s);
			}
			result[0]= stat.sampleMean();
			stat = null;//Clear stat for use during next
			claculation
			//End of Ao Calculations

			//Begin AoH Calculations
			currentvals = connection.prepareStatement ("SELECT
			[SRAN], [Type], [Year], [Qtr], [Achieved] FROM [out
			Flying Hours] WHERE [Year]= "+year+" AND [QTR]
			="+quarter);
			curValues = currentvals.executeQuery();
```

```
stat = new SimpleStats();
while(curValues.next()) {
      double s = curValues.getDouble(5);
      stat.newobs(s);
}
result[1]= stat.getSampleTotal();
stat  =  null;//Clear  stat  for  use  during  next
claculation
//End of AoH Calculations

//Begin En Calculations
currentvals  =  connection.prepareStatement  ("SELECT
[SRAN  ID],  [Platform  type],  [Year],  [Qtr],
[Unscheduled] FROM [out Aircraft events] WHERE [Year]=
"+year+" AND [QTR] ="+quarter);
curValues = currentvals.executeQuery();

stat = new SimpleStats();
while(curValues.next()) {
      double s = curValues.getDouble(5);
      stat.newobs(s);
}
result[2]= stat.getSampleTotal();
stat  =  null;  //Clear  stat  for  use  during  next
claculation
//End of En Calculations

//Begin SBn Calculations
currentvals  =  connection.prepareStatement  ("SELECT
[Type], [SRAN], [Year], [Qtr], [Buys] FROM [out New
Buys] WHERE [Year]= "+year+" AND [QTR] ="+quarter);
curValues = currentvals.executeQuery();

stat = new SimpleStats();
while(curValues.next()) {
      double s = curValues.getDouble(5);
      stat.newobs(s);
}
result[3]= stat.getSampleTotal();
stat  =  null;  //Clear  stat  for  use  during  next
claculation
//End of SBn Calculations

//Begin Sh Calculations
currentvals  =  connection.prepareStatement  ("SELECT
[From  SRAN],  [To  SRAN],  [Type],  [Year],  [Qtr],
[Shipments]  FROM  [out  Shipments]  WHERE  [Year]=
"+year+" AND [QTR] ="+quarter);
curValues = currentvals.executeQuery();

stat = new SimpleStats();
while(curValues.next()) {
      double s = curValues.getDouble(6);
      stat.newobs(s);
```

```
                }
                result[4]= stat.getSampleTotal();
                stat = null; //Clear stat for use during next
                claculation
                //End of Sh Calculations

                //Begin MTBF Calculations
                result[5]= result[1]/result[2];
                //End of MTBF Calculations

                //Begin Pt Calculations
                currentvals = connection.prepareStatement ("SELECT
                [Task], [Object type], [SRAN ID], [Year], [Qtr], [Avg]
                FROM [out Tasks Performed] WHERE [Year]= "+year+" AND
                [QTR] ="+quarter);
                curValues = currentvals.executeQuery();

                stat = new SimpleStats();
                while(curValues.next()) {
                      double s = curValues.getDouble(6);
                      stat.newobs(s);
                }
                result[6]= stat.getSampleTotal();

                stat = null;
                currentvals = null;
                curValues = null;
                connection.close();
                connection = null;
          }
          catch (ClassNotFoundException cnfe) {
          System.err.println(cnfe);}
          catch (SQLException sqle) {
                System.err.println(sqle);}
    }

    public double[] getResult() {
          return result;
    }

    public void setResult(double[] result) {
          this.result = result;
    }
}
```

2. How It Works

The implementation of *UpdateOutput* has a constructor that receives two parameters: (1) a String object representing a URL for the data source, and (2) a String object representing a URL for the ODBC driver. There is one instance variable; an array of *double* objects of size seven used to save the output values.

The *doUpdate()* method performs all the functions in this class and it is invoked by the constructor. After it receives the same parameters as the constructor, the method starts with the identification of the data source, the creation of a connection, and it instantiates a *SimpleStats* object to perform all the statistics. Two integer objects determine the year and quarter from which to retrieve the data.

To calculate Ao, the application uses a *PreparedStatement* object to query the *out Availability* table. The SQL used as argument retrieves every entry on that table with a year equal to 2009 and quarter equal to four. A *while* loop controls the submission of data to the *SimpleStats* object for statistical analysis. At the end, the application invokes the *SimpleStats sampleMean()* method to populate the first item on the *double* array.

In the case of AoH, En, SBn, Sh, and Pt, the application creates new SQL statements that query the *out Flying Hours, out Aircraft events, out New Buys, out Shipments,* and *out Tasked Performed* tables, where year equals 2009 and quarter equals four. The rest of the process here is similar to that of systems availability except, in these cases, the statistic saved is the sample total versus sample mean.

For MTBF, the application takes the saved AoH and divides it by the saved En.

L. SIMPLESTATS CLASS

1. Source Code

SimpleStats is a statistical program the author created during his first Java class. It provides users with basic statistical measures of interest. The source code for the *SimpleStats* class follows below:

```
public class SimpleStats {

	private double sampleMean;
	private double sampleVariance;
	private int sampleSize;
	private double min;
	private double max;

	public SimpleStats() {
		reset();
	}

	public void reset() {
```

```
            sampleMean = Double.NaN;
            sampleVariance = Double.NaN;
            min = Double.POSITIVE_INFINITY;
            max = Double.NEGATIVE_INFINITY;
            sampleSize = 0;
      }

      public void newobs(double x) {
            sampleSize++;
            if (sampleSize == 1) {
                  sampleMean = x;
                  sampleVariance = 0.0;
                  max = x;
                  min = x;
            }
            if (sampleSize > 1) {
                  sampleVariance = ((((double) sampleSize - 2) /
                  (sampleSize - 1) * sampleVariance) + (((x -
                  sampleMean) * (x - sampleMean)) / sampleSize));
                  sampleMean = (sampleMean + (x - sampleMean) /
                  sampleSize);
                  min = Math.min(min, x);
                  max = Math.max(max, x);
            }
      }

      public double sampleMean() {
            return (sampleMean);
      }

      public double sampleVariance() {
            return (sampleVariance);
      }

      public double sampleStdDev() {
            return (Math.sqrt(sampleVariance));
      }

      public int sampleSize() {
            return (sampleSize);
      }

      public double min() {
            return (min);
      }

      public double max() {
            return (max);
      }

      public double getSampleTotal() {
            return sampleMean*sampleSize;
      }
}
```

2. How It Works

The constructor invokes the *reset()* method to make sure all values are set to the initial condition. Every process takes place within the *newobs()* method. The remaining methods are all getter methods.

APPENDIX B. NOLH DESIGN

In order to maximize the efficiency and space-filling effect of this experiment, the author uses the orthogonal and nearly orthogonal LH worksheet (Sanchez, 2005) to develop Tables 9 through 11 listing each design points. The worksheet is an Excel-based tool developed to ease in the design of large-scale simulation experiments.

low level	0	0	0	0.5	0
high level	5	30	30	1.5	10
decimals	0	0	0	4	4
factor name	Spares	IQ	I Cap	Deg	ST
	1	13	12	0.9531	3.3594
	4	9	13	0.9609	0.9375
	2	23	0	0.7734	4.1406
	3	27	9	0.8672	4.375
	0	12	17	0.7344	1.0156
	4	13	21	0.5078	3.9844
	2	30	23	0.7891	1.5625
	3	21	27	0.5625	3.4375
	0	1	8	0.7031	1.9531
	5	2	7	0.7656	1.1719
	0	29	14	0.7813	3.125
	5	30	8	0.8828	1.4063
	2	8	29	0.6797	4.0625
	4	7	28	0.8516	0.8594
	1	16	29	0.7266	4.6875
	4	23	30	0.5313	1.7188
	1	5	6	1.2422	0.3906
	5	11	8	1.2813	4.2188
	1	23	8	1.0234	2.1094
	3	29	11	1.25	1.25
	1	6	25	1.4531	4.9219
	3	8	19	1.4219	3.75
	1	22	23	1.4766	3.2813
	4	19	29	1.4844	1.875
	1	7	14	1.1094	3.2031
	5	0	14	1.0703	1.4844
	1	21	8	1.5	0.5469
	5	28	11	1.2031	1.6406
	1	14	23	1.0547	3.5938
	3	14	30	1.1953	4.8438
	2	22	21	1.2578	0.7031
	3	19	28	1.3906	1.3281
	0	11	10	0.8359	5.4688
	4	15	4	0.5938	5.3906
	2	28	2	0.9141	7.2656

Table 9. NOLH Design (Part 1)

low level	0	0	0	0.5	0
high level	5	30	30	1.5	10
decimals	0	0	0	4	4
factor name	Spares	IQ	I Cap	Deg	ST
	3	26	5	0.9375	6.3281
	0	10	19	0.6016	7.6563
	4	4	15	0.8984	9.9219
	2	28	28	0.6563	7.1094
	4	25	26	0.8438	9.375
	2	15	1	0.6953	7.9688
	4	11	13	0.6172	7.3438
	2	20	6	0.8125	7.1875
	3	24	4	0.6406	6.1719
	1	1	15	0.9063	8.2031
	4	13	19	0.5859	6.4844
	0	24	20	0.9688	7.7344
	3	26	24	0.6719	9.6875
	2	12	6	1.4609	7.4219
	4	3	4	1.0781	7.8125
	2	16	9	1.2891	5.2344
	5	25	4	1.375	7.5
	2	6	18	1.0078	9.5313
	4	4	27	1.125	9.2188
	1	21	16	1.1719	9.7656
	4	26	25	1.3125	9.8438
	1	3	10	1.1797	6.0938
	3	12	3	1.3516	5.7031
	1	25	12	1.4453	10
	3	20	13	1.3359	7.0313
	2	10	27	1.1406	5.5469
	3	3	18	1.0156	6.9531
	2	18	25	1.4297	7.5781
	5	17	20	1.3672	8.9063
	3	15	15	1	5
	4	17	18	1.0469	6.6406
	1	21	17	1.0391	9.0625
	3	7	30	1.2266	5.8594
	2	3	21	1.1328	5.625
	5	18	13	1.2656	8.9844
	1	17	9	1.4922	6.0156
	3	0	7	1.2109	8.4375
	2	9	3	1.4375	6.5625
	5	29	22	1.2969	8.0469
	0	28	23	1.2344	8.8281
	5	1	16	1.2188	6.875
	0	0	23	1.1172	8.5938
	3	22	1	1.3203	5.9375
	1	23	2	1.1484	9.1406
	4	14	1	1.2734	5.3125
	1	8	0	1.4688	8.2813
	4	25	24	0.7578	9.6094

Table 10. NOLH Design (Part2)

low level	**0**	**0**	**0**	**0.5**	**0**
high level	**5**	**30**	**30**	**1.5**	**10**
decimals	**0**	**0**	**0**	**4**	**4**
factor name	**Spares**	**IQ**	**I Cap**	**Deg**	**ST**
	0	19	22	0.7188	5.7813
	4	7	22	0.9766	7.8906
	2	1	19	0.75	8.75
	4	24	5	0.5469	5.0781
	2	22	11	0.5781	6.25
	4	8	7	0.5234	6.7188
	1	11	1	0.5156	8.125
	4	23	16	0.8906	6.7969
	0	30	16	0.9297	8.5156
	4	9	22	0.5	9.4531
	0	2	19	0.7969	8.3594
	4	16	7	0.9453	6.4063
	2	16	0	0.8047	5.1563
	3	8	9	0.7422	9.2969
	2	11	2	0.6094	8.6719
	5	19	20	1.1641	4.5313
	1	15	26	1.4063	4.6094
	3	2	28	1.0859	2.7344
	2	4	25	1.0625	3.6719
	5	20	11	1.3984	2.3438
	1	26	15	1.1016	0.0781
	3	2	2	1.3438	2.8906
	1	5	4	1.1563	0.625
	3	15	29	1.3047	2.0313
	1	19	17	1.3828	2.6563
	3	10	24	1.1875	2.8125
	2	6	26	1.3594	3.8281
	4	29	15	1.0938	1.7969
	1	17	11	1.4141	3.5156
	5	6	10	1.0313	2.2656
	2	4	6	1.3281	0.3125
	3	18	24	0.5391	2.5781
	1	27	26	0.9219	2.1875
	3	14	21	0.7109	4.7656
	0	5	26	0.625	2.5
	3	24	12	0.9922	0.4688
	1	26	3	0.875	0.7813
	4	9	14	0.8281	0.2344
	1	4	5	0.6875	0.1563
	4	27	20	0.8203	3.9063
	2	18	27	0.6484	4.2969
	4	5	18	0.5547	0
	2	10	17	0.6641	2.9688
	3	20	3	0.8594	4.4531
	2	27	12	0.9844	3.0469
	3	12	5	0.5703	2.4219
	0	13	10	0.6328	1.0938

Table 11. NOLH Design (Part 3)

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Cioppa, T.M. (2002, September). Efficient nearly orthogonal and space-filling experimental designs for high-dimensional complex models. PhD. Dissertation. Monterey, CA: Naval Postgraduate School.

Cioppa, T.M., & Lucas, T.W. (2006). Efficient nearly orthogonal and space-filling Latin hypercubes. *Technometrics*.

Clockwork Solutions, Inc. (1992-2007). Total life cycle management assessment tool (TLCM-AT). *User's Manual Ver. 5.2.* Austin, TX: Clockwork Solutions, Inc.

Clockwork Solutions, Inc. (1992-2005). Aircraft total life cycle assessment software tool (ATLAST). *Technical Reference Manual Ver. 5.0.* Austin, TX: Clockwork Solutions, Inc.

Clockwork Solutions, Inc. (2007, August). TLCM-AT training material, Chapter 2. Austin, TX: Clockwork Solutions, Inc.

Devore, J.L. (2004). *Probability and statistics for engineering and the sciences.* Belmont, CA: Thomson Learning.

EPA Victoria. (2006). *A wealth of business value, life cycle management.* Retrieved July 12, 2008, from http://www.epa.vic.gov.au/lifecycle/default.asp

Gurvitz, N. Naaman.Gurvitz@clockwork-solutionsus.com. RE: Undelivered Mail Returned to Sender. August 21, 2008.

Horton, I. (2005). *Ivor Horton's beginning Java™ 2 JDK 5 Edition*. Indianapolis, IN: Wiley Publishing.

Kleijnen, J.P.C., Sanchez, S.M., Lucas, T.W., & Cioppa, T.M. (2005). A user's guide to the brave new world of designing simulation experiments. *INFORMS Journal on Computing*, *17*, No. 3, 263-289.

Law, A., & Kelton, W.D. (1999). *Simulation modeling and analysis.* Boston, MA: McGraw-Hill.

Sanchez, S.M. (2005). NOLHdesigns spreadsheet. Retrieved July 23, 2008, from http://diana.cs.nps.navy.mil/SeedLab/

SECNAVINST 5400.15C. (2007, September). Department of the Navy (DON) Research and Development, Acquisition, Associated Life-Cycle Management, and Logistics Responsibilities and Accountability.

Sun Developer Network. (2007). Developer forums, Java programming - copy file. Retrieved July 25, 2008, from http://forums.sun.com/thread.jspa?messageID=2834416

Wikipedia. (2007). Data farming. Retrieved July 31, 2008, from http://en.wikipedia.org/wiki/Data_farming

Young, E. (2008). *Total life cycle management – assessment tool: An exploratory analysis.* Master's thesis. Monterey, CA: Naval Postgraduate School.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Maj Matthew Reuter
   Headquarters, USMC
   Installation & Logistics
   Washington, D.C.

4. Capt David Vaughan
   Headquarters, USMC
   Installation & Logistics
   Washington, D.C.

5. Nicholas Linkowitz
   Headquarters, USMC
   Installation & Logistics
   Washington, D.C.

6. Ronald Brassard
   Headquarters, USMC
   Installation & Logistics
   Washington, D.C.

7. Maj Robert Charlton
   Headquarters, USMC
   Installation & Logistics
   Washington, D.C.

8. Maj Stephen Mount
   Marine Corps Systems Command
   Quantico, Virginia

9. Hugh Saint
   Clockwork Solutions
   Austin, Texas

10. Larry Paige
Concurrent Technologies Corporation
Stafford, Virginia

11. Professor Thomas W. Lucas
Naval Postgraduate School
Monterey, California