ABSTRACT

Title of thesis:	IMPACT OF HARDWARE OBSOLESCENCE ON SYSTEM SOFTWARE FOR SUSTAINMENT-DOMINATED ELECTRONIC SYSTEMS
Degree Candidate:	Arindam Goswami
Degree and Year:	Master of Science, 2004
Thesis directed by:	Associate Professor Peter A. Sandborn

The last two decades have witnessed manufacturers of sustainment-dominated long field life electronic systems incorporate Commercial Off the Shelf (COTS) technology products into their systems on a large scale. Many of these products, however, have lifetimes of significantly shorter duration than the systems they are incorporated into and as a result become obsolete long before the system's intended duration of useful life is over.

This problem is especially prevalent in avionics and military systems, where systems may encounter obsolescence problems even before they are fielded and always during their support life. Manufacturing that takes place over long periods of time exacerbates this problem.

Many part obsolescence mitigation strategies exist including: lifetime buy, last-time buy, part replacement, aftermarket source, uprating, emulation, re-engineering, salvage, and ultimately redesign of the system. Design refresh (or redesign) has the advantage of treating multiple existing and anticipated obsolescence problems concurrently and additionally allows for functional upgrades.

Hitherto, there have been studies concentrated on determining the optimum combination of different obsolescence strategies by using life cycle cost as the deciding criterion. However, these studies take into account *only* hardware life cycle costs. In many systems, such as avionics systems, software life cycle costs (redesign, rehosting and requalification) have a significant bearing on total life cycle cost. Thus software redesign due to part obsolescence triggered hardware redesign should also be addressed during life cycle management planning.

This thesis describes a methodology and it’s implementation for determining the hardware part obsolescence impact on life cycle sustainment costs for system software based on future production projections, maintenance requirements and part obsolescence forecasts. The methodology extends the MOCA (Mitigation of Obsolescence Cost Analysis) methodology/tool that determines the optimum design refresh plan during the field-support-life of the

product in order to minimize life cycle cost. The design refresh plan consists of a set of design refresh activities and their respective calendar dates.
The methodology incorporates the use of two software commercial cost analysis models: PRICE S and COCOMO.

The methodology developed in this thesis has been validated using a Navy test case (VH-60N Digital Cockpit Upgrade Program). It has also been applied to Honeywell International, Inc.’s AS900 engine controller. The results obtained demonstrate the necessity of taking software redesign analysis into account during life cycle management planning.

IMPACT OF HARDWARE OBSOLESCENCE ON SYSTEM SOFTWARE FOR SUSTAINMENT-DOMINATED ELECTRONIC SYSTEMS

by

Arindam Goswami

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2004

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who have helped me along the way to the point that I find myself at as I write this thesis. For starters, I wish to thank my advisor, Dr. Peter Sandborn, for his guidance and support and a million other things that have not only helped me during my research but will also stand me in good stead all my life. As it is with pretty much every student in the Sandborn group, to him I owe a lot. I am also grateful to the members of my dissertation committee: Dr. Donald Barker for being very patient and accommodating as I wrote my thesis and postponed my defense more than once and Dr. Omar Ramahi for having agreed to be on my committee at such short notice.

Thanks are due to all my lab mates and friends (together this list is big enough for me to not take individual names – but you all know who you are), past & present, for all their support.

Finally, I would like to thank my family for their love and support. I can never put into words all the gratitude that I feel – so, I shall not even try.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 - Introduction

1.1 The Use of Commercial Off the Shelf (COTS) Technology in Avionic Systems

Electronic components are ubiquitous in airplane systems: they are found in almost every system, including those that are primarily mechanical, hydraulic, and pneumatic. The solid-state electronics industry has grown in parallel with the airplane industry. Until the mid nineteen eighties, commercial aerospace manufacturers depended on a well-developed military electronic components and specifications infrastructure to assure long-term availability of components that met their needs. This was possible because the military market sector alone comprised about 25% of the total market for electronic components [Condra, 1999]; it was responsible for a good deal of the device innovation, and therefore owned many device designs. As a result, military and commercial aerospace electronic design, manufacturing, procurement, operation, maintenance, and support decisions had traditionally been based on two assumptions:

1. The supply of electronic components specified to operate in aerospace environments is unlimited; and
2. Component designs will remain stable for long periods of time.

However, these assumptions are no longer true, especially sine 1992, when many major manufacturers of electronic components, including Motorola, Intel, and Philips, exited the military market [Wright et. al., 1997]. As shown in Figure 1.1, the entire aerospace industry now consumes less than one per cent of the electronic components produced. The major component markets are computers, consumer electronics, and others, which do not have the demanding environmental or long production life cycle requirements of aerospace products.

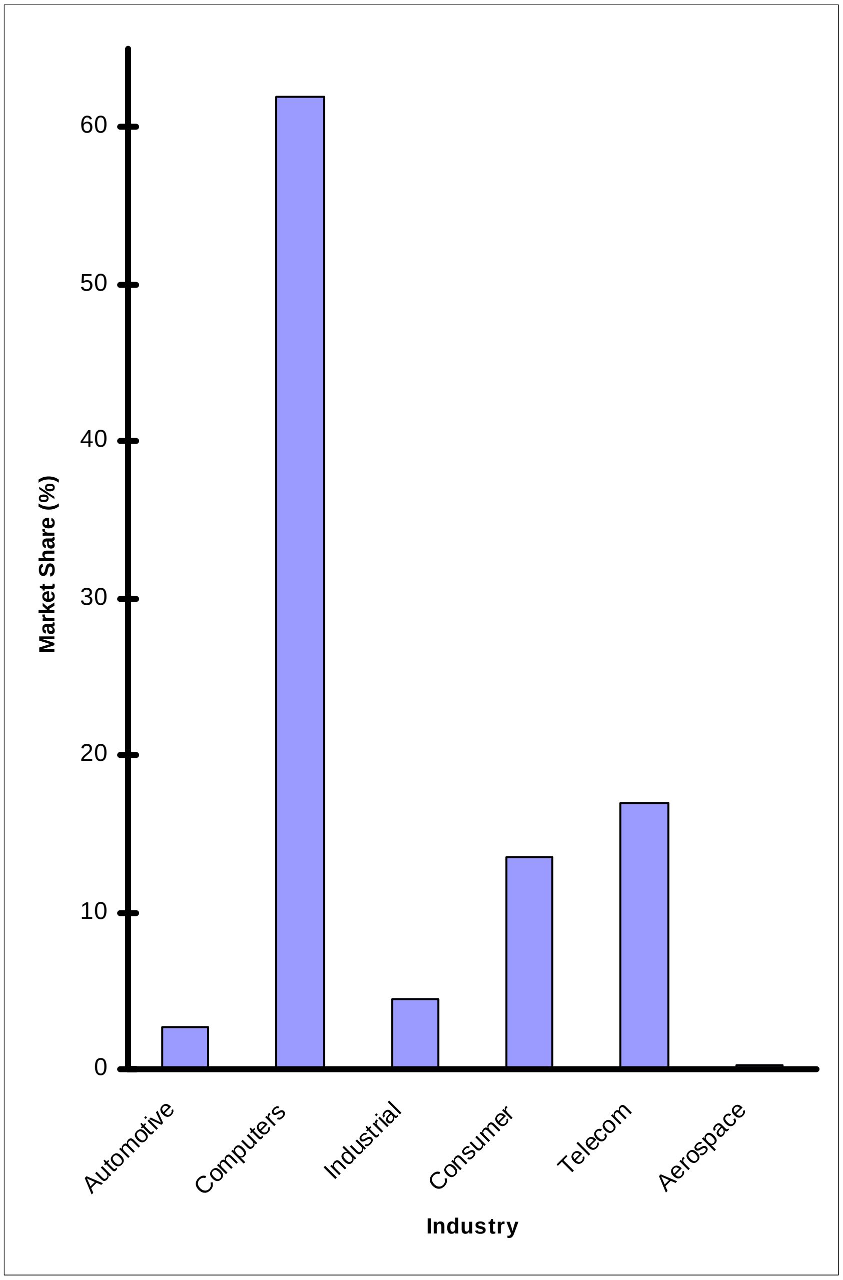


Figure 1.1: Industry wise usage of electronic components in 1999 [Condra, 1999]

As a result of these trends, the availability of components actually specified for aerospace applications is decreasing. The response of the aerospace industry has been to incorporate, on a large scale, commercial off the shelf (COTS) technology in its systems. COTS refers to commercially available products that can be purchased and integrated into existing systems with little or no customization and thereby lead to cost savings as compared to designing new parts every time.

Especially in the case of military applications, the focus on commercial support of aviation systems is driven by the modernization needs faced by all the branches of the military at a time when resources for acquisition of new defense systems are increasingly constrained. The procurement lull in new system acquisition, and the increasing reliance on aging platforms far past their original planned life cycle, is expanding the need for a concerted effort to upgrade and maintain existing systems. Innovative approaches for the support of existing systems can be used to produce life cycle savings, reduce cycle times and improve performance [JACG, 1999].

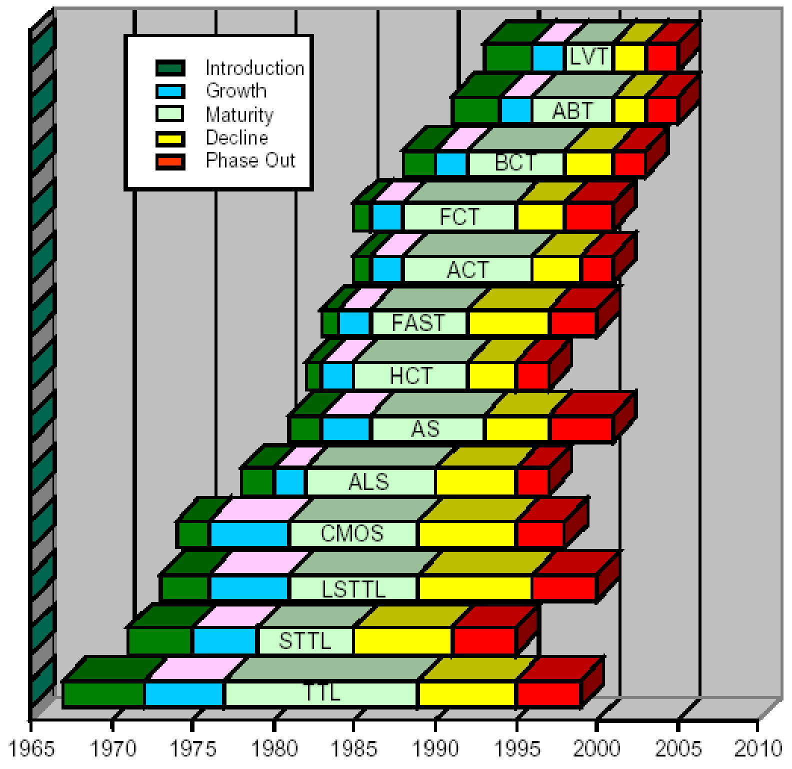


Figure 1.2: Shrinking trend in component technology life cycles [Condra, 1999]

However, there are many challenges associated with doing so. Figure 1.2 shows that the life cycles of all integrated circuit technologies are shrinking, almost to the point where the term *component technology life cycle* is meaningless [Baca, 1997]. Even stable component designs are modified constantly to reduce cost, improve yields, and enhance performance. The modifications are evaluated and characterized for high volume applications, such as computers, but the applications of low volume users such as aerospace are rarely considered. The lifetime of a typical jet airplane will encompass many generations of electronic component design, as illustrated in Figure 1.3. For example, the F/A-22, which entered dedicated initial operational test and evaluation (DIOT&E) in 2003, was already suffering due to this life cycle mismatch, with technology refreshment required before the aircraft has even reached operational service [MAT, 2003].

Thus we are confronted with a situation in which the aerospace industry depends on electronic components, but can no longer count on sources of stable supply that are specified for its specific applications. The aerospace industry must learn how to use components conceived and produced for other industries (these components having a much shorter life cycle than the system they go into) while keeping total life cycle costs at an acceptable level. This mismatch becomes especially critical because over their extended lifetime these systems are faced with the following situations [Singh, 2001]:

a) Field failures - Defects in the system may cause it to fail before its field life is completed necessitating the manufacturing of additional spare units.
b) Design changes - Additional requirements may be added after the design stage is completed.

c) Production spread out over the lifetime of the product - Production may be distributed over time either in a planned fashion or as the result of additional orders for the product.

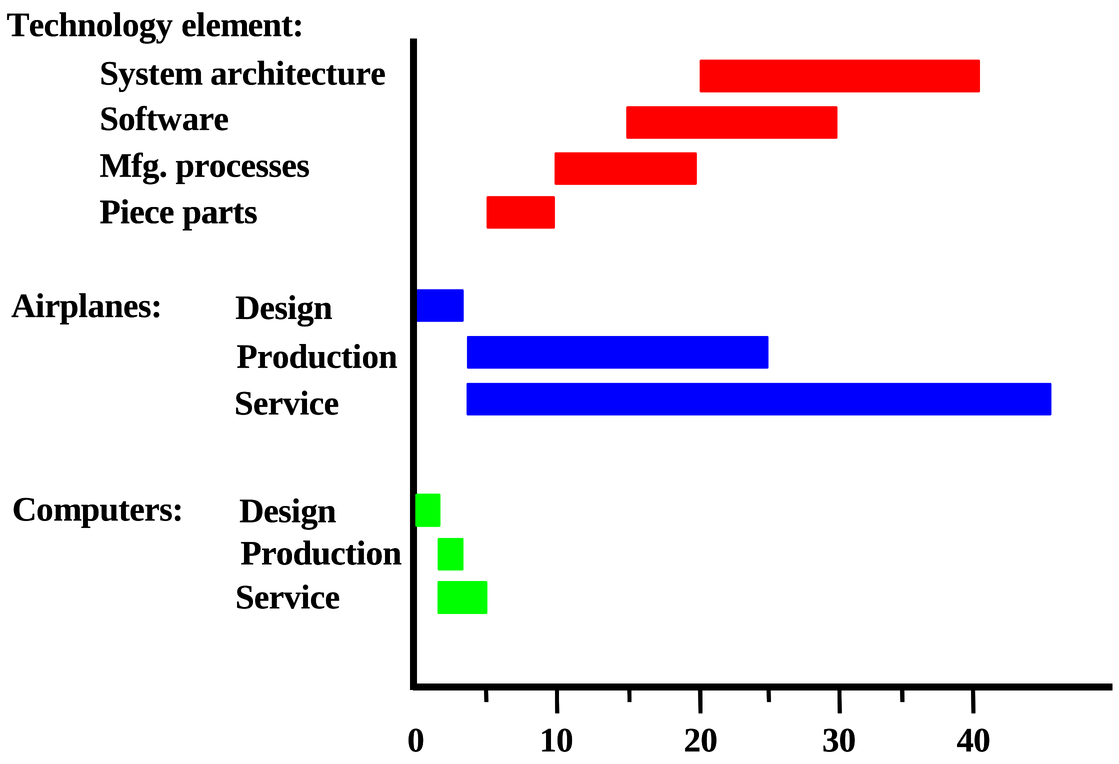


Figure 1.3: Technology element lifetimes compared to product lifetimes for airplanes and computers [Condra, 1997]

1.2 The Problem of Component Obsolescence

The aerospace industry has responded to the changing scenario by implementing activities that can be broadly grouped into three categories: (1) How to anticipate occurrences of component obsolescence; (2) How to react to occurrences of component obsolescence; and (3) How to quantitatively estimate, during the design process, the impact of the life cycle mismatch between avionic systems and the components that go into them. It is important to understand the problem of part obsolescence in order to fully appreciate these activities.

A part is obsolete when the technology that defines the part is no longer implemented. Diode Transistor Logic (DTL) and Resistor Transistor Logic (RTL) parts are examples of obsolete part technologies. There are many reasons why parts go obsolete, including [Singh, 2001]

- Manufacturing of the part is not cost effective for the current manufacturers.
- The manufacturers have better opportunities in the market for other products and therefore their interests migrate.
- The market is governed by products with large-scale production and therefore manufacturers refuse to produce smaller volume specialized products (if the requirement for the obsolete part(s) is relatively small).
- The raw material or equipment needed for the manufacturing of obsolete parts is hard to maintain or procure.
- Changes in legislation make the materials or processes associated with a part impractical or illegal to continue using.

There may be more specific reasons why parts go obsolete depending upon the part and the market conditions, however in most cases it is the technology push (i.e., development and introduction of a new technology) and the manufacturer’s interests, which lead to part obsolescence.

An appreciation of these driving factors along with an understanding of component technology trends (for example, Moore’s law), taking into account technology roadmaps and building flexibility into the original design (for example, partitioning designs to place high-risk components on throwaway modules) will enable system designers to be better prepared to tackle the problem of component obsolescence.

1.3 Motivation For Tackling Part Obsolescence in a Proactive Manner

Component obsolescence entails issues that have a direct and significant impact on the total life cycle cost. It influences both hardware and software (as discussed in the next section). Consider, for example [Singh, 2001], that a manufacturer, which manufactures a long field life system, has a need to manufacture additional systems. Assume in this case that some of the parts required to manufacture the system are obsolete. The manufacturer would try to procure those obsolete parts from external vendors, who purchased and stored obsolete parts for later sale. These vendors inflate the price of the obsolete parts they have been storing and therefore these obsolete parts are then procurable at some cost penalty to the manufacturer. The manufacturer has little choice but to incur the cost increase. Now the manufacturer has two options for handling the increased cost. One option (not possible in many cases because of contractual obligations) is to carry over the extra cost of the obsolete parts to the buyer (customer). The second option (most likely the real world scenario) is that the manufacturer bears the extra expenses of the obsolete part itself. This extra cost increases the life cycle cost to the manufacturer. It can be clearly understood that for aircraft with 30-plus year production lives, unforeseen part obsolescence costs can mount significantly if one depends on reactive solutions. On the one hand is the option of facing up to the obsolescence problem in a purely "reactive manner". This entails performing obsolescence management only when intimation about an obsolescence event is received - no planning is done at an earlier stage to mitigate the obsolescence of parts. When a notice indicating that a part is going to become obsolete in the near future is received from the part manufacturer, various obsolescence management strategies are evaluated and the most economically feasible option is chosen. However, the costs incurred on account of such unplanned activities can be quite large. It is much more prudent to be pro-active and take cognizance of the problem beforehand and consider all possible combinations of different obsolescence mitigation strategies which can minimize sustainment costs. As can be seen from the above example, the manufacturer will be motivated to explore different obsolescence mitigation approaches in order to minimize the total costs incurred by him. These mitigation approaches, e.g., short-term approaches (for example, last-time buy) or long-term approaches (for example storing the obsolete part for the lifetime of the system or eliminating the need for that part completely by redesigning it) then need to be ranked against each other so that the most cost effective mitigation or combination of various mitigation strategies can be adopted. Two broad categories of these strategies are discussed below.

1.3.1 Short-Term Obsolescence Mitigation

Short-term obsolescence mitigation involves taking necessary steps to delay the effect of part obsolescence on product sustainment costs until more rigorous measures can be taken to remove the part obsolescence problem, e.g., buy enough parts to last until the part is either replaced by a similar part or the product is redesigned to eliminate use of the part. This strategy is termed as “last-time buy”.

1.3.2 Long-Term Obsolescence Mitigation

Long-term obsolescence mitigation is defined as measures taken to either eliminate the obsolete part from the product permanently or to sustain the supply of the obsolete part throughout its lifetime by storing it. The elimination of the obsolete part may involve complete removal of the obsolete parts from the product by redesigning the product, replacement of obsolete parts with a suitable equivalent substitute part, replacement the obsolete parts with an emulated version of the part, or thermal uprating of a non-obsolete commercial version of the part [Wright, 1997].

Several combinations of different obsolescence mitigation strategies can be adopted at different “time-points” during the lifetime of a system. A tool called MOCA (discussed in Chapter 3) has been developed at the University of Maryland, which through a detailed cost analysis determines the best possible combination vis-à-vis total life cycle cost.

1.4 The Impact Of Part Obsolescence On System Software

Total life cycle cost, generated using life cycle cost analysis (LCCA), is used as a metric to quantify the impact of various factors that have a bearing on it. Life cycle cost is defined as the sum of present values of investment costs, capital costs, installation costs, energy costs, operating costs, maintenance costs, and disposal costs over the lifetime of the project, product, or measure [McArthur, 1989]. LCCA is particularly suited to the evaluation of design alternatives that satisfy a required performance level, but that may have differing investment, operating, maintenance, or repair costs; and possibly different life spans. LCCA can be applied to any capital investment decision, and is particularly relevant when high initial costs are traded for reduced future cost obligations. The MOCA tool allows its users to choose a combination of several possible obsolescence mitigation approaches and then based on the production plan, other system characteristics and system qualification costs carries out a cost analysis - the final output of which is a design refresh plan that defines the best “time-points” in the life of a system at which to carry out design refreshes. A design refresh in the context of MOCA and also this thesis is defined, [Singh, 2001], as an activity that involves bringing about changes in the design of a system to incorporate one or more of the following:

a) New or improved functionality

b) Removal of part obsolescence and other related issues

c) Aesthetic improvement of product

d) Ease of use or more user-friendly

e) Reliability improvement of the system

f) To make the system more maintainable and repairable

g) Change in documentation/engineering drawings

h) Incorporate new technology/innovations

Hitherto, MOCA did not take into account costs arising due the impact of a design refresh activity on the system software. A system's hardware and software components together enable an implementation of its functionality. One or more of the interlinked hardware components execute the functional subtasks - the logic of this execution being determined by the software components. Therefore, there is a link between these components and the functional description of the system. This link is depicted in Figure 1.4.

A design refresh event, as defined above, during the lifetime of a system may result in the introduction of new hardware components or redesign of existing ones. A need to modify the system software arises as a result of that. This modification of system software can entail high costs. These costs are due not only to modification of existing code but also development of new code and software re-qualification. Therefore, there is a need to associate these changes in hardware with the changes in software precipitated by them in order to quantify the impact of the latter on life cycle cost. The MOCA tool represents an effort to address the problem of hardware obsolescence. The work embodied in this thesis aims at extending this effort to quantifying the impact of software change precipitated a change in hardware incorporate the resulting analysis into MOCA.

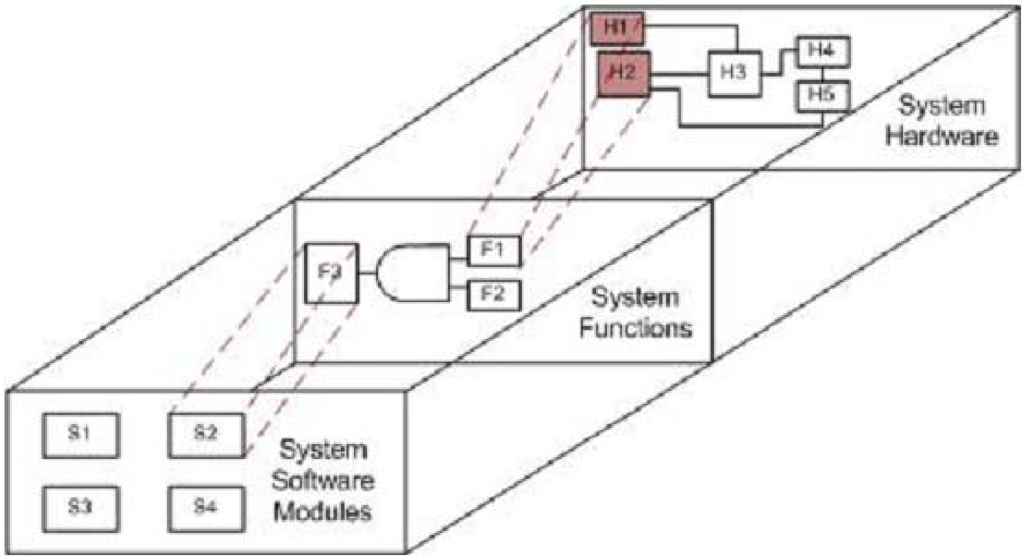


Figure 1.4: The link between functional description and hardware and software components of the system [Szabó, 2000]

1.5 The Objective of this Thesis

The starting point for the work done in this thesis is the set of different design refresh plans generated by MOCA. MOCA ranks these different design refresh plans according to the total life cycle cost that each one entails. However, as mentioned above, this life cycle cost did not include the cost accrued on account of the system software that might need to be changed during a design refresh activity. This problem has been addressed here. The objective of this thesis was to determine the impact of part obsolescence on system software for long field life electronic systems based on the production plan, maintenance requirements and part obsolescence predictions. This impact was to be quantified in terms of acquisition, modification and support costs. These costs were then added to the costs associated with design refresh and re-qualification to obtain a net cost. This net cost was to be used by MOCA in its existing cost analysis algorithms to schedule design refreshes during the life of the product in the field. The design refresh plan was to provide the number of design refresh activities and their respective calendar dates to minimize the life cycle sustainment cost of the product.

The approach to associating system software changes to the hardware obsolescence events, which precipitate them, is discussed in detail in Chapter 2. Chapter 2 also documents the degree to which the steps in the solution have been implemented in this work. The MOCA tool is discussed in Chapter 3. The various interfaces are explained. The cost model used is discussed in detail therein. Chapter 4 contains a case study. The output from the analysis performed on the case study example is compared with some real world data. The refined insights into the problem at hand yielded by this comparison are discussed. Chapter 5 contains a discussion of the contributions made by this thesis, the existing shortcomings in the present model and the future work that needs to be done.

Chapter 2– Solution Architectures

2.1 Hardware Obsolescence Precipitated Change in System Software

As discussed in Chapter 1, cost analysis and careful planning in the initial stages of system design can help mitigate component obsolescence related costs in long field life systems. The studies carried out in the University of Maryland have hitherto concentrated on hardware costs alone that arise from component obsolescence. However, for many sophisticated systems, functionality is software driven. Therefore, there is a link between the system software, which “thinks” and the system hardware which “acts”. At a hardware component obsolescence event, the linkage between hardware and software may result in a need to change portions of the system software. Software changes can be categorized into three types [Wong, 1996]:

a) Adaptive changes,

b) Perfective changes, and

c) Corrective changes.

Adaptive changes accommodate technological improvements/changes. Perfective changes seek to make future evolution somehow better, more manageable and less costly. Corrective changes focus on detecting, tracking, and diagnosing defects and their root causes. The effective management and execution of these changes is critical and defines a significant portion of the system’s life cycle cost. The software changes being dealt with in this thesis are primarily of the “adaptive” type. At a hardware obsolescence event the impact of this change has to be quantified and factored into the existing cost analysis. Two approaches for doing so will be discussed in this chapter (Sections 2.2 and 2.3).

2.2 Software Design Refresh by Quantifying Change in the Number of Source Lines of Code (SLOC) Due to a Hardware Part Obsolescence Event

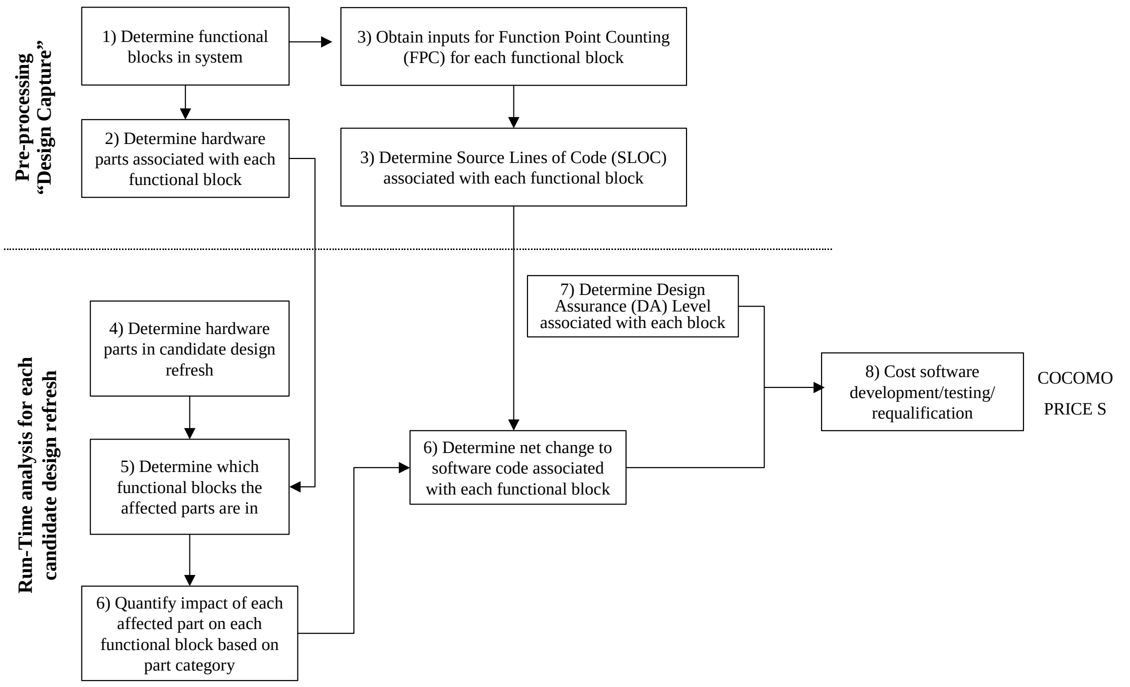

Pre-processing "Design Capture"
1) Determine functional blocks in system
3) Obtain inputs for Function Point Counting (FPC) for each functional block
2) Determine hardware parts associated with each functional block
3) Determine Source Lines of Code (SLOC) associated with each functional block
Run-Time analysis for each candidate design refresh
7) Determine Design Assurance (DA) Level associated with each block
4) Determine hardware parts in candidate design refresh
8) Cost software development/testing/ requalification
COCOMO
PRICE S
5) Determine which functional blocks the affected parts are in
6) Determine net change to software code associated with each functional block
6) Quantify impact of each affected part on each functional block based on part category

Figure 2.1: Software design refresh by Fault tree based analysis of functional blocks.
(Note: The numbers in the boxes correspond to the sequence in which the respective steps have been explained below)

There are certain characteristics of a software functional block (explained below) that influence software development costs. These characteristics are modified at the time of a design refresh activity depending on how the hardware parts participating in it are modified. The software redesign approach illustrated in this section works by capturing all the characteristics of the system (of the Hardware parts as well as the functional blocks), gauging how these are modified during design refresh and then using the modified values, in conjunction with commercial software cost estimating tools such as

PRICE S or COCOMO, to calculate software development costs. The block diagram for this approach is shown in Figure 2.1 and is described below.

Pre-processing "Design Capture":

1) Determine functional blocks in the system and their attributes - Determining functional blocks involves understanding what the major functions performed by the system software are. The system is partitioned into functional blocks such that each of these blocks can be thought of as performing one particular function. These blocks are not necessarily physical partitions. A functional block might, and in most cases will, include hardware parts that are physically remote from each other. Partitioning the system into functional blocks seems logical in light of the fact that system software is modular in nature, i.e., it contains modules - each of which performs a different function. Each functional block can be thought of as a mapping onto a software module that has some fixed attributes.

Two of these attributes are implementation language (C++, ADA, etc.) and Source Lines of Code (SLOC)[1]. They are used in the quantitative analysis as discussed in Chapter 3. Another set of fixed attributes (Inputs, Outputs, Logical Files, Interface files and Inquires) is used for function point counting. These will be explained in Appendix A. "SLOC modifier" is an attribute that varies during run-time and gives a measure of how much SLOC associated with a functional block needs to be changed at an obsolescence event. It does this by quantifying the impact of the parts that participate in the block, which have gone obsolete.

[1] Code refers to the symbolic arrangement of data or instructions in a computer program. It is a series of statements written in some human readable computer programming language. SLOC is a metric that is used to measure the number of such statements in a software program.

As mentioned above, the final software cost estimation is performed using tools such as PRICE S or COCOMO. This requires that certain other inputs pertaining to the functional blocks, needed by the cost estimating models employed by these tools, be acquired as well. These inputs, which will be explained in Chapter 3, are entered at this stage.

2) Determine hardware parts associated with each functional block – In this step, information about participation of a hardware part in a functional block is extracted from the parts list (described in Chapter 3). Information extracted includes, among other things:

a) Which block(s) the hardware part participates in, and
b) The level of participation (characterized by “High”, “Medium”, “Low” or “None”) and the number of instances of this hardware part in the functional block(s) in question.

Consider, as an example, a computer. It uses a monitor as an output device for display purposes. The Operating System (OS) contains a software module dedicated to manage the “display” functionality. We treat the “display” function as one functional block and the monitor as one of the hardware parts involved in realizing this functionality. If a significant amount of code in the software module is dedicated to “driving” the monitor; then, in the event that this particular monitor becomes obsolete, a large amount of software code may have to be re-written. This necessitates that, in order to capture this large degree of dependence, it be recorded, during pre-processing, the fact that the hardware part (monitor) participates in the functional block (display) and *also* that the level of participation of the former in the latter is “High”. This step can be thought of as one that determines the mapping between the hardware parts of the system and the functional blocks.

***3) Determine Function/Feature Point Count inputs for the functional blocks and hence the SLOC associated with each block* -** In order to understand the methodology of “software design refresh by quantifying the change in the Number of Source Lines of Code (SLOC)”, it is important to grasp the concept of Function/Feature Point Counting.

Function/Feature Point Counting is a technique by which, based on certain characteristics specific to the functionality of a software module (in the context of this thesis – the functionality of a functional block), the metric Function Point Count is generated – which in turn lends itself to an estimation of the size of the module, i.e., number of lines of code needed to implement the desired functionality, depending, of course, on the language used for coding (e.g., C++, JAVA).

A Function Point Count is a metric used to measure the functionality, and, from it the size of a software system. It can be used in the early stages of development. Function Point Counting begins by identifying the components of the system as seen by the end-user. These components are the inputs, outputs, inquiries, interfaces to other systems, and logical internal files. There may be several instances of each component. The components are then classified as simple, average, or complex. Numerical values are assigned to each component type depending on the number of instances of it in the system and the classification (simple, average or complex) of each of these instances. These values are then added and the resulting total is called “Unadjusted Function Point total (UFP)”.

Complexity factors described by 14 general systems characteristics, such as reusability, performance, and complexity of processing are scored on a scale of 0 - not present, 1 - minor influence, to 5 - strong influence. The sum of scores of all the complexity factors is multiplied by the UFP to get the net function point count. This count can then be related by empirical factors to system size.

One might question, “Why go through this software sizing procedure when the number of lines of code for each functional block is available upfront? ” The reason for this is that when a hardware part is redesigned, and some *new* code is to be developed as a result of that, it is not always possible to directly estimate how many lines of code will be needed. It is better to estimate the impact that the redesigned part will have on the functionality of the functional block(s), which it participates in. This changed functionality can, in turn, be used to calculate the change in function point count and from this the number of new lines of code can be estimated. This estimate is more realistic.

The details of Function Point Analysis can be found in Appendix A. The discussion therein contains an introduction to the history of Function/Feature Point counting, pertinent definitions, the factors used in this technique and a detailed explanation of the technique itself, i.e., the procedure to obtain the “counts” and size software code using them.

At this point, it is important to keep in mind that as per the solution architecture developed in this thesis, the system designer supplies the quantitative inputs required for function point analysis for all the functional blocks; using which the SLOC (and also the *change* in SLOC at any point during the system's lifetime) associated with each one of them can be calculated.

Steps 1, 2 and 3, i.e., determining the functional blocks in the system, acquisition of data on how the hardware parts to relate to these blocks, and the function point analysis inputs for the functional blocks together constitute "Design Capture" of the system.

Run-time Analysis for each candidate design refresh

4) Determine hardware parts affected in candidate design refresh plan - As mentioned in Chapter 1, the starting point for the implementation of the methodology[2] developed in this thesis is the design refresh plan generated by MOCA. This design refresh plan contains, among other things, a list of parts that have to be changed/modified at a particular date during the system's lifetime. This list of affected parts is imported for the purpose of software redesign analysis.

5) Determine which functional blocks the affected parts are in – This information is obtained from the mapping developed in Step 2 during the pre-processing phase. It should be noted that a particular hardware part might belong to more than one functional block. For example, a microprocessor in a computer will participate in several functions to be performed by the system. In a case like that an analysis of the impact on each functional block has to be carried out separately.

At this stage, all the information obtained in the abovementioned steps is sorted to obtain, for each functional block, a list of affected hardware parts that participate in it.

[2] Hereafter referred to as the software redesign analysis

6) Quantify impact of each affected part on each functional block based on part category and hence SLOC change for the latter- Each of these affected parts belongs to a certain part category [Singh, 2001]. This input indicates the type of part being used. Several part categories are available in MOCA. These are: Microcircuit, Diode, Transistor, Integrated circuit, Semiconductor, Assorted and Custom Defined. The "Assorted" part type is used to represent an aggregate of parts and their instances. All the parts, which do not have any obsolescence or maintenance issues (e.g., most passives and mechanical devices), are lumped together into a single part to reduce computation time. A "Custom Defined" part is a part type for which no single standard part type could be used. When a new part is synthesized as a result of a design refresh, the obsolescence date is reset based on a default and a lifetime is obtained based on the part category of the modified part.

Software redesign analysis is carried out for every functional block. Based on the category that each of the affected parts belongs to and the number of instances of that part in the functional block being considered, a value is obtained for "SLOC modifier" (*for that particular hardware part*), which is a measure of how much SLOC associated with a functional block needs to be changed during design refresh on account of the hardware part in question. The rationale for this step is that a complex part (such as a microprocessor) going obsolete will result in a larger code change than a simple part (such as a resistor) going obsolete.

The appropriate function point inputs for each functional block can be *recalculated* using the "SLOC modifier" values for all the affected parts that participate in it. Using these modified function point inputs the amount of *new* SLOC to be generated, i.e., the net change in SLOC ($\Delta KSLOC$), can be calculated for each functional block.

***7) Determine Design Assurance (DA) level associated with each block* -** These levels are a measure of the criticality and complexity of the function performed by each of these blocks. They have a strong bearing on the re-qualification costs of the system. DA levels ‘A’ through ‘E’ correspond to failure condition classifications catastrophic, hazardous, major, minor, and no effect, respectively. During the analysis a Design Assurance (DA) level associated with the affected functional blocks is calculated. For each functional block the DA level is identified based on system Functional Hazard Assessment [Beland, 2000]. Prior to the start of the analysis, an assessment of complexity is made for each functional block. Typically, software code dealing with Math functions or String manipulation is assigned a lower DA level (i.e., E), while functional blocks that deal with functions such as Real time command and control operations are assigned higher DA levels.

***8) Cost software development and testing* –** The final costing is done using either the PRICE S model or COCOMO. For this purpose, all the requisite parameters (acquired during “design capture”), including the ones with modified values (on account of the above steps) are sent to one of these tools[3]. Cost values for new software development for

[3] In the case of PRICE S, the communication between it and MOCA takes place via a dynamic link library (DLL) included in MOCA. DLLs are a collection of small programs, which can be called upon when needed by the executable program (exe) that is running. The DLL lets the executable communicate with some other program and contains source code to do particular functions.

all the affected functional blocks are summed up and the total is added to the life cycle cost.

These steps are repeated for all the candidate design refreshes. Prior to the introduction of software redesign analysis, MOCA would rank different design refresh plans (each plan containing one or more design refreshes – the total for each plan being the sum of costs entailed on account of all the design refreshes contained in it) based on total cost. The same thing is done now – with the difference that the totals contain a "software redesign cost" as well – which may change the rankings.

2.3 Software Design Refresh by Fault Tree Based Analysis of Functional Blocks

This approach is similar in many regards to the one outlined in Section 2.2. Functional blocks and Function Point analysis are used in this approach too. "Design Capture" as explained in the previous section is a portion of this approach as well. The difference lies in the fact that this methodology attempts to make use of, if available, the fault tree for the system software.

A Software Fault Tree is a graphical technique for identifying and documenting the combinations of lower level software events that allow a top-level event to occur [Leveson, 1995]. Software Fault Tree Analysis (SFTA) is a deductive, top-down method used to analyze system functionality. An example of a fault tree based functional description of a software system is shown in Figure 2.2. In this three-tier fault tree, Function 1, Function 2, Function 3 and Function 4 are bottom events; Function 5 and Function 6 are intermediate events and Function 7 is the top event. An OR gate signifies that a function is implemented if *any* of the functions below it are realized while an AND gate signifies that a function is implemented only if *all* the functions below it are realized.

The example of Figure 2.2 is a very simple one. An actual system is likely to have more tiers with several other intermediate events of different types with more complex relationships denoted by the various logic gates.

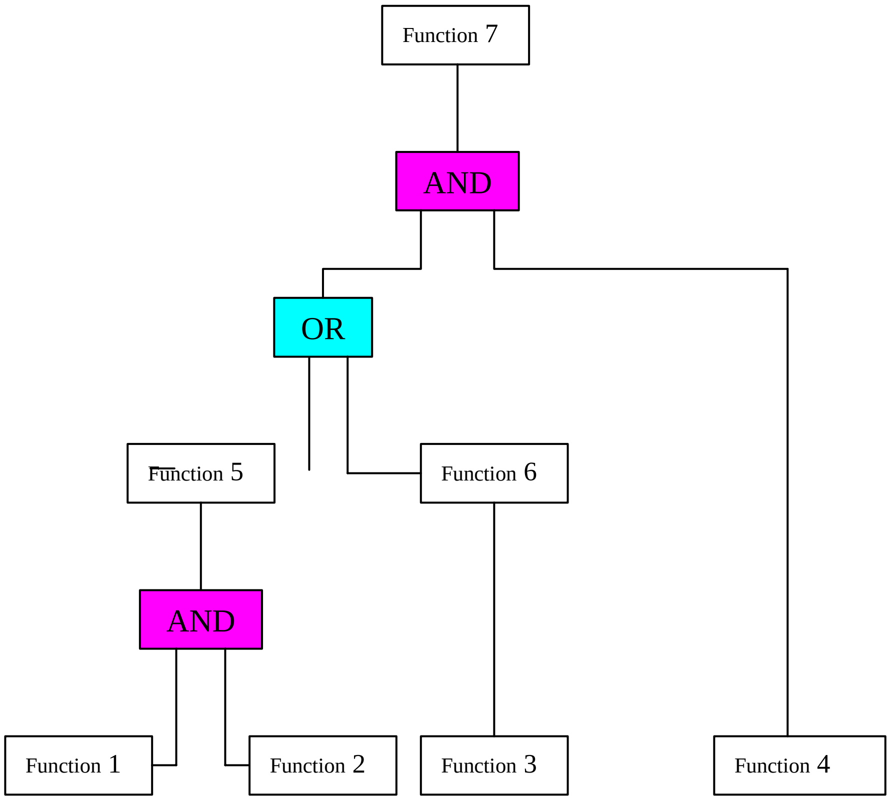


Figure 2.2: Fault Tree for a hypothetical system

For the purpose of the methodology discussed in this section, a fault tree can allow automation of the procedure to determine criticality levels of the functional blocks (analogous to DA level in the previous approach). This is because “criticality” values can be assigned to functional blocks based on where they exist in the hierarchy of the tree – with higher values assigned to blocks at the top of the tree.

Another advantage is that a fault tree gives a better idea about the interdependencies between the different functional blocks – something that was not taken into account by the previous method. This can have a significant bearing on qualification costs because if one functional block is redesigned then all the functional blocks connected to and above it on the tree may have to be qualified as well.

The block diagram for using fault-tree based analysis for software redesign approach is shown in Figure 2.3.

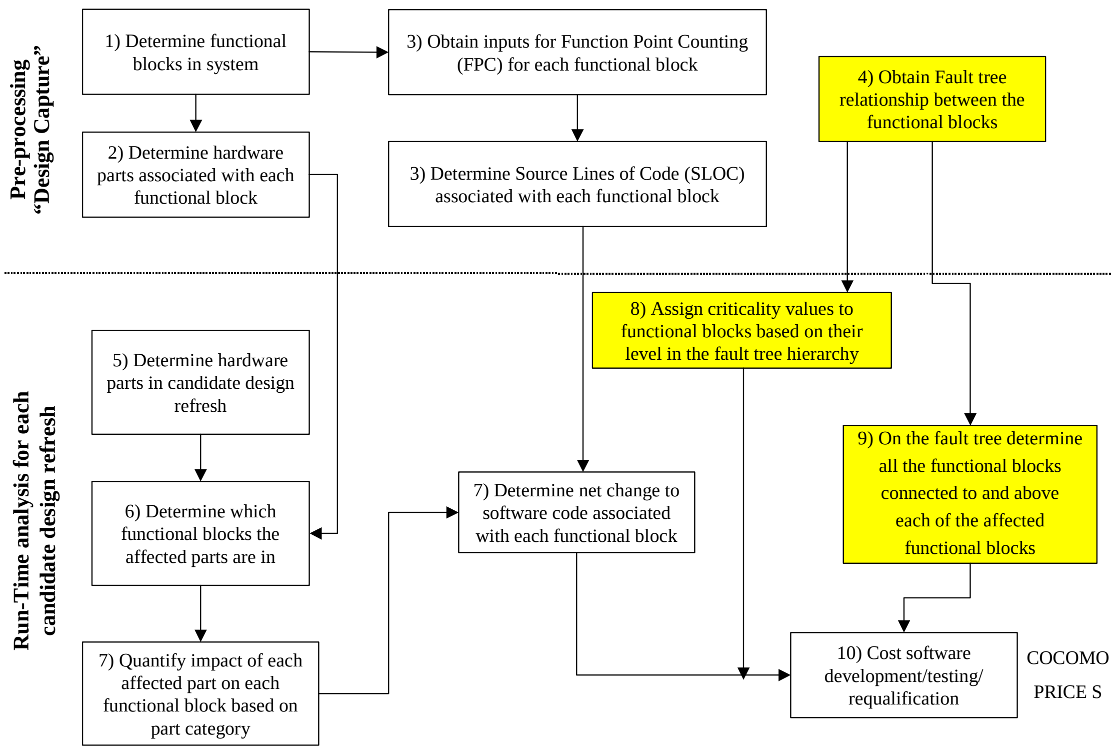

Pre-processing
"Design Capture"
1) Determine functional blocks in system
2) Determine hardware parts associated with each functional block
3) Obtain inputs for Function Point Counting (FPC) for each functional block
3) Determine Source Lines of Code (SLOC) associated with each functional block
4) Obtain Fault tree relationship between the functional blocks
Run-Time analysis for each candidate design refresh
5) Determine hardware parts in candidate design refresh
6) Determine which functional blocks the affected parts are in
7) Quantify impact of each affected part on each functional block based on part category
7) Determine net change to software code associated with each functional block
8) Assign criticality values to functional blocks based on their level in the fault tree hierarchy
9) On the fault tree determine all the functional blocks connected to and above each of the affected functional blocks
10) Cost software development/testing/ requalification
COCOMO
PRICE S

Figure 2.3: Software design refresh by Fault tree based analysis of functional blocks.
(Note: The numbers in the boxes correspond to the sequence in which the respective steps have been explained below)

It should be noted that some of the steps are the same as in the previous approach. Table 2.1 lists all the steps that are common to the two approaches. Their explanation is the same as in the previous section. The new steps in this approach, indicated by the shaded boxes in the block diagram, will be explained below.

Table 2.1: Enumeration of steps common to the two approaches for software design refresh explained in Sections 2.1 and 2.2

Step Number in Section 2.2	Step Number in Section 2.3	Remarks
1	1	Determine functional blocks in system
2	2	Determine hardware parts associated with each functional block
3	3	• Obtain inputs for Function Point Counting (FPC) for each functional block • Determine Source Lines of Code (SLOC) associated with each functional block
4	5	Determine hardware parts in candidate design refresh
5	6	Determine which functional blocks the affected parts are in
6	7	• Quantify impact of each affected part on each functional block based on part category • Determine net change to software code associated with each functional block
8	10	Cost software development/testing/requalification

As mentioned above, for Steps 1-3 refer to Section 2.2

4) Obtain Fault tree relationship between the functional blocks – This requires an understanding of how the functional blocks are connected to each other. The system designer should have an appropriate blueprint for the software architecture of the system. This blueprint can be specified in the form of a tabular arrangement. For the fault tree shown in Figure 2.2, disregarding the logic gates for the moment, this tabular arrangement is shown in Table 2.2

Parent	**Child**	**Remarks**	**Tier**
Function 7	Function 5	Top_Node	Tier 3
Function 7	Function 6	Top_Node	Tier 3
Function 7	Function 4	Top_Node	Tier 3
Function 6	Function 3		Tier 2
Function 5	Function 2		Tier 2
Function 5	Function 1		Tier 2
Function 4			Tier 1
Function 3			Tier 1
Function 2			Tier 1
Function 1			Tier 1

This tabular arrangement can be stored in the form a Text file that can then be read and converted into a Fault tree. This can be done by using the following two-part algorithm starting with the above table in the form of a text file:

```
Find_ Top _Node (file)
      Step 1 Scan "Remarks" column for Top _Node
      Step 2 For each row with Top _Node found do
         Step 2.1  function = function listed in "Parent" column
         Step 2.2  Create node in tree for function and call it Top_Node
         Step 2.3  Set function_tier = entry in "Tier" column
         Step 2.4  Fault_Tree_Create (function, file)
```

```
Fault_Tree_Create (function, file)
      Step 1 Scan "Parent" column for function
      Step 2 For each row with function found do
         Step 2.1 child =  Function listed in "Child" column
         Step 2.2 Connect child to function
         Step 2.3 Set child_tier = entry in "Tier" column
         Step 2.4 child =  Fault_Tree_Create (child, file)
```

After the fault tree has been created, a relationship analysis along the lines of Software Failure Modes, Effects and Criticality Analysis (SFMECA) can be carried out [Dehlinger, 2004] to determine the appropriate inclusion of Logic Gates between the different functional blocks.

Steps 1, 2, 3 and 4, i.e., determining the functional blocks in the system, acquisition of data on how the hardware parts to relate to these blocks and the function point analysis inputs for the functional blocks, and "reading" the software fault tree together constitute "Design Capture" of the system.

Steps 5, 6 and 7 correspond to the Steps 4, 5 and 6 in Section 2.2 respectively.

8) Assign criticality values to functional blocks based on their level in the hierarchy of the fault tree- These values are a measure of the criticality and complexity of the function performed by each of these blocks. They have a strong bearing on the re-qualification costs of the system. These values can be assigned on the basis of the following information extracted from the fault tree:

a) The tier at which the functional block exists. A higher tier function should be assigned a greater weight than a lower tier one.
b) Type of "Logical" connection to the function at the tier immediately above. A greater weight should be assigned if the connection is of the "AND" (because of

the indispensability of the lower tier function implied by an AND gate) type than if it is of the “OR” type.

The purpose of this weighting is to establish the significance and criticality of the individual functions in the overall scheme of things.

9) Determine all the functional blocks connected to and above each of the affected functional blocks – In this step we determine, for each affected functional block, i.e., one for which new code is being developed, which are the other blocks that need to be requalified. The rationale for this step is that in addition to an affected block, which has to go through the entire software development cycle that includes design, development and testing; all functional blocks *dependent* on it have to be requalified as well. Information pertaining to the dependency can be extracted from the fault tree by locating all the functional blocks connected to and above it in the tree. This method has the inherent flexibility to take into account only those connected blocks that are a specified number of tiers above the affected functional block. Also, if the affected block is connected to a block in the tier above by an “OR” gate, requalification for the latter may not be so critical. Thus the analysis can be tweaked in tune with the user’s specifications.

Step 10, i.e., to cost software development and testing is the same as Step 8 in Section 2.2.

2.4 Remarks

The solution scheme of Section 2.2 alone has been implemented in this thesis. The other approach (Section 2.3) entails the problem of reading a fault tree with its logical connections into MOCA. It should be noted that this could not be performed successfully and was an impediment in the implementation of the methodology being described in this section.

However, It must be mentioned that this approach continues to be a potential candidate method for mapping software to hardware, which, of course, lies at the heart of the software design refresh methodology.

Chapter 3 - Software Redesign Analysis Applied to Design Refresh Planning

3.1 Introduction

A methodology and its implementation (Mitigation of Obsolescence Cost Analysis - MOCA) has been developed at the University of Maryland [Singh and Sandborn, 2002] for determining the part obsolescence impact on life cycle sustainment costs for the long field life electronic systems based on future production projections, maintenance requirements and part obsolescence forecasts. Based on a detailed cost analysis model, the methodology determines the optimum design refresh plan during the life cycle (design, production, and operation and support) of the product. The design refresh plan consists of the number of design refresh activities, their respective calendar dates and content to minimize the life cycle sustainment cost of the product. The methodology supports user determined short- and long-term obsolescence mitigation approaches on a per part basis and allows for inputs to be specified as probability distributions that can vary with time. Outputs from this analysis are used as inputs to the PRICE System's PRICE H/L commercial software tools for predicting life cycle costs of systems. As mentioned in Chapter 2, this tool hitherto took into account the impact of hardware obsolescence on only the system hardware. This thesis introduces into MOCA the ability to account for the impact of hardware obsolescence on system software as well. In this chapter, the implementation of the software redesign analysis within MOCA is described.

Some of the commonly used terms in this chapter are:

1) System - The entity for which the sustainment cost is being evaluated. A system may be composed of multiple boards.[4] The System represents the top node in the System-Board-Part hierarchy.
2) Board - Subsystems that contain multiple parts. All parts that belong to the system must belong to at least one board. The system part list is obtained by accumulating the part list from each board.
3) Part - A part represents a unique part number, name or identification number in the system. It is an entity that resides on the boards. Each board may have multiple identical instances of a single part. A single part may also appear within multiple boards.
4) Component - A specific instance of a hardware part. Usually distinguished from other instances by the board it is in and its physical location on a board.
5) Functional Block – A portion, or module, of system software dedicated to performing one specific function. Each functional block can be mapped onto a software module. These blocks do not necessarily correspond to physical partitions.
6) Event - An occurrence that affects the system life cycle – could be part obsolescence, design refresh (defined in Chapter 1) or a reorder[5].

In Section 3.2, the overall MOCA methodology and the incorporation of the software redesign approach of Section 2.2 into it will be explained. Interfaces for collecting inputs

[4] MOCA accommodates an arbitrary hierarchy of boards within boards. The system is the "container" for the entire board hierarchy.

[5] An event that brings an additional number of instances of the system into existence. It can be due to planned production of units spread over the lifetime of the system or additional orders during the system's field support life.

pertinent to software redesign analysis will be described in Section 3.3. Wherever appropriate, these inputs will be discussed. Discussion of the cost models employed by the PRICE S and COCOMO tools is the subject of Section 3.4

3.2 Overall MOCA Methodology And The Incorporation Of Software Redesign Analysis

The MOCA tool is a JAVA application created at the University of Maryland [Singh and Sandborn, 2002]. MOCA, which is an acronym for Mitigation of Obsolescence based Cost Analysis, can be used during the design stage of the life cycle, to predict the cost of sustaining the system. Its primary use is to compare, in terms of sustainment cost, several life cycle management strategies for a design. From an overall system point of view, the model helps to focus designers on the specific parts/boards, the obsolescence of which may cause problems and escalate life cycle cost.

3.2.1 The Problem Addressed by MOCA

Suppliers of low-volume electronics products need to support their products over extended time periods. In many cases, the parts that are used in these products are obsolete prior to the end of the product’s life. The system manufacturer (sustainer) is then faced with a series of alternatives. Often the decision boils down to whether to make a "last-time buy" of parts necessary to last until the next design refresh, a “lifetime buy” of parts necessary to last through the remaining predicted life cycle of the product, count on being able to purchase required parts from third parties (whether they are the original part, emulated, or a replacement part), or to initiate an immediate design refresh in order to design the part out of the product (i.e., by replacement or elimination). Design refresh is often very expensive, requiring extensive engineering, qualification testing, and certification. Making the problem worse, revisions to software necessitated by the new parts can also involve significant engineering effort (and re-qualification), sometimes exceeding that required for hardware design and qualification.

On the one hand, system sustainers do not wish to pay increased prices for obsolete parts and do not wish to risk losing the ability to sustain a system, but on the other hand, too many design refreshes to avoid obsolescence issues may be extremely expensive. Somewhere between these extremes lies an optimum balance between non-design refresh obsolescence mitigation solutions and design refreshes. MOCA performs a tradeoff analysis to find this optimum point.

3.2.2 MOCA Analysis Algorithm

The algorithm employed by MOCA can be broken down into the following steps:

Step 1: Determine system details – relevant information pertaining to the system, constituent subsystems (boards), and individual parts is collected. This information includes forecasting part obsolescence.

Step 2: Determine the order of known events – the events (reorders, part obsolescence and redesigns) affecting the system are determined and arranged in an ordered list called an 'Event List'

Step 3: Schedule design refresh activities at various stages in the life of the product based on the planned production schedule.

For each design refresh activity:

Step 4: Calculate costs accrued by affected system hardware

Step 5: Calculate costs accrued by affected system software

Step 6: Sum the costs due to events during the product's life for each design refresh plan generated in Step 4 and Step 5

Step 7: Rank the various design refresh plans

These steps have been elaborated on below.

Step 1: Determine system details

Three categories of inputs are necessary to populate the model.

a) Part inputs: Part costs, reliability data, part characterization data, functional block participation data and obsolescence data.

b) Board (any subsystem containing parts) inputs: Board assembly, disassembly, test, qualification and total acquisition costs, and the list of parts assigned to each board. The overall part list for the system is obtained by combining the part lists of each board.

c) System inputs: Dates for the beginning and end of the system's life

In addition to these, solution control inputs are used to control various parameters used in computing the sustainment cost and planning the design refreshes. Appendix B explains all the inputs mentioned above in detail.

Step 2: Determine the order of known events

The events that a system undergoes in its lifetime include part obsolescence events, reorder events and design refresh events. All these events contribute to the sustainment cost. Figure 3.1 shows the simplified MOCA view of events that are relevant to design refresh optimization.

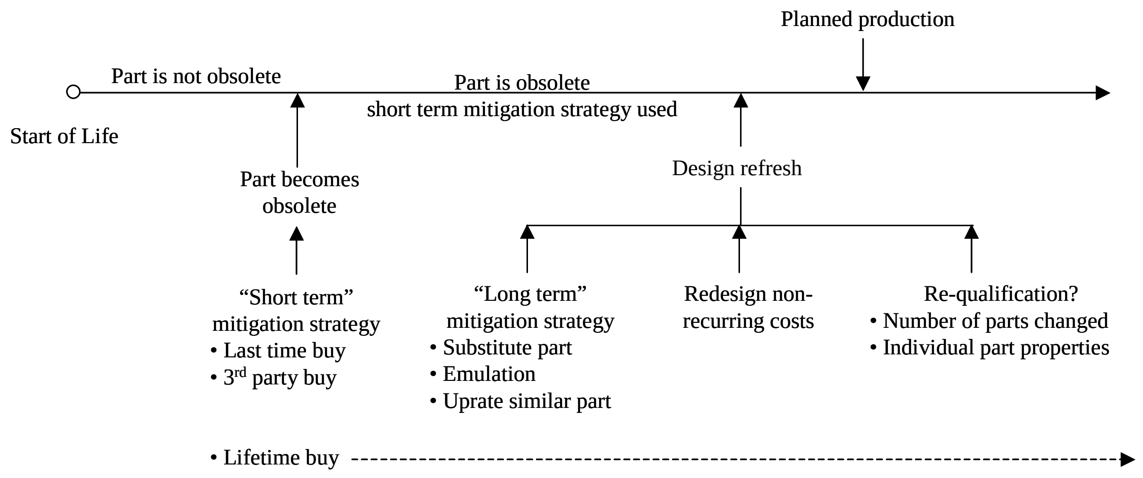


Figure 3.1: MOCA system timeline

a) Part obsolescence Events: A part is obsolete when the technology that defines the part is no longer implemented. Part obsolescence has been explained in Section 1.2. The obsolescence date of a part in MOCA is the date after which the part is no longer available from the original manufacturer (from which the part was procured for the current design).

Obsolescence can be handled in a variety of ways in MOCA:

- Purchase and store the obsolete part(s) (lifetime buy, in-store) and use whenever required
- Replace the obsolete part(s) with the same part(s) but with different procurement cost(s) (aftermarket source)
- Replace the obsolete part(s) with different part(s) with different or similar cost(s) but with same functionality (part substitution, emulation, reverse engineering)
- Design refresh of the system to eliminate the need of the obsolete part(s)

b) Reorders (Planned Production): The reorder date represents the start of processing of an order request or start of a planned production. With every reorder, a new quantity of systems comes into existence. Each order will correspond to a different start date on the system timeline. A reorder can occur due to production spread over the lifetime of the product. The production can be for units, which were planned initially at the design stage, or additional orders during the product's field support life.

c) Design Refresh: The design refresh date refers to the completion of all the design refresh activities carried on the product. In MOCA, a "Design Refresh" is equivalent to redefining the part set used in the system. "Design Refresh" has been defined in Section 1.4. Every time a design refresh event is specified, a new part set needs to be specified, and the new part set is used for all calculations beyond that point. Any future reorders made are manufactured on the basis of the new design.

Step 3: Scheduling "design refresh" activities

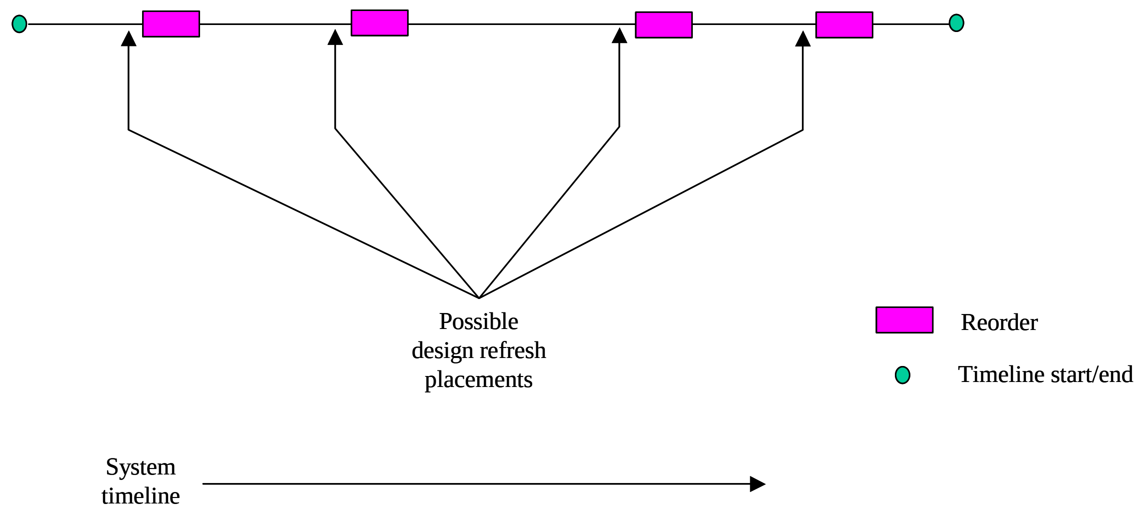


Figure 3.2: Associating design refreshes with reorders [Singh, 2001]

MOCA schedules design refreshes immediately before the reorder events. This is because scheduling it at any other time point before the reorder involves the risk of parts becoming obsolete between the design refresh and the reorder event. This step minimizes the time span between a design refresh and the next reorder, thereby eliminating the need to perform another design refresh because of any probable part obsolescence during that time period. Thus design refreshes can be thought of as being associated with reorders. This is depicted in Figure 3.2. For the case shown in Figure 3.2, a total of 15 design refresh plans are possible. These plans are enumerated in Table 3.1:

Table 3.1: Enumeration of possible design refresh schedules for the case depicted in Figure 3.2

Point(s) on system timeline at which "design refresh" event is inserted	Remarks
A	Design refresh plan with ONE design refresh activity
B	Design refresh plan with ONE design refresh activity
C	Design refresh plan with ONE design refresh activity
D	Design refresh plan with ONE design refresh activity
A+B	Design refresh plan with TWO design refresh activities
A+C	Design refresh plan with TWO design refresh activities
A+D	Design refresh plan with TWO design refresh activities
B+C	Design refresh plan with TWO design refresh activities
B+D	Design refresh plan with TWO design refresh activities
C+D	Design refresh plan with TWO design refresh activities
A+B+C	Design refresh plan with THREE design refresh activities
A+B+D	Design refresh plan with THREE design refresh activities
A+C+D	Design refresh plan with THREE design refresh activities
B+C+D	Design refresh plan with THREE design refresh activities
A+B+C+D	Design refresh plan with FOUR design refresh activities

Step 4: Calculate costs accrued by affected system hardware

- All the events are inserted into the system timeline in the order of their dates of occurrence.
- At a part obsolescence event, the part is treated according to its individual user-specified obsolescence mitigation strategy. The part's original cost is multiplied with an obsolescence mitigation factor that depends on the type of mitigation strategy adopted and the new cost is rolled up to the board level. This "rolling-up" reflects the fact that eventually a higher cost will be incurred to procure that board. The system cost is also modified to reflect the parts/board cost changes due to obsolescence events.
- At a reorder event the reorder cost is calculated based on the quantity of systems/boards reordered. At this stage the cost of the system reflects the contribution of all the events prior to the reorder event in question, which have resulted in an increase in the system cost.
- At a design refresh event all parts that have become obsolete prior to it are identified. MOCA also lets the user specify an input called "look-ahead time". This input signifies that whenever a design refresh takes place, MOCA looks-ahead for forecasted part obsolescence issues and pro-actively removes those part obsolescence problems at the current design refresh opportunity. There is a tradeoff involved over here. By having a large design refresh look-ahead time the number of design refreshes can be reduced. On the other hand by design refreshing parts that are not yet obsolete there is a risk of incurring extra cost for no improvement, i.e.,

there is a possibility that the obsolete part that was proactively design refreshed is never required in the future.

Thus parts that have become obsolete prior to the design refresh event and those that fall inside the "look-ahead" range from the date of the design refresh event are combined into a list of affected parts.

Based on the obsolescence mitigation strategies of the affected parts they can be either redesigned at the design refresh or left as they are for the system's support lifetime.

If they are to be redesigned, user-specified inputs pertaining to the complexity level of the parts and their associated design refresh effort are used to make required changes to the system at a design refresh event and also to determine the cost of the design refresh event itself. New parts without any immediate obsolescence problems are used to replace the obsolete parts. The obsolescence date for the new part is determined by a user input called TACTech lifecode and the average mean lifetime of the part category that the new part belongs to. All the data for the old part in the database is modified with the corresponding values of the new part.

Step 5: Calculate costs accrued by affected system software

The solution architecture developed in Section 2.2 is applied here. The list of affected parts generated in Step 4 is an input to this step.

For each of the affected parts an assessment is made of its impact on all the functional blocks that it participates in. The cost value returned by PRICE S or COCOMO (which reflects the total effort to develop new code - for all the functional blocks - on account of one particular type of affected part) is rolled up to the board level. It should be noted that a single type of part might be present on more than one board. Thus, for each board in the system, for rolling up the values to the board level, the PRICE S/COCOMO generated cost value is multiplied by a factor given by the following equation:

$$factor = \left(\frac{\text{Number of instances of the affected part on the board}}{\text{Number of instances of the affected part in the system}} \right)$$

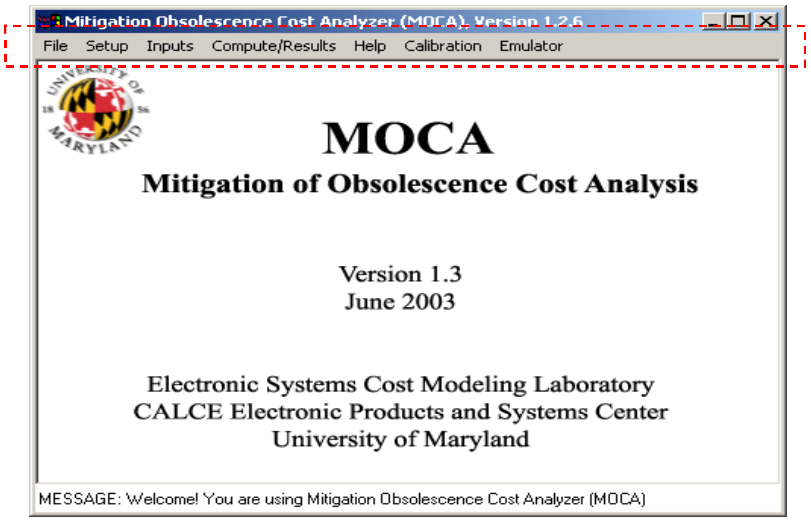
The new cost of the boards, and hence of the system, reflects the cost increase of system software accounting for software redesign.

Step 6: Total costs accrued by the system in Step 4 and Step 5 are added up. This is done for each design refresh plan.

Step 7: The various design refresh plans are ranked on the basis on minimum life cycle cost.

3.3 MOCA Interfaces For Collecting Inputs Pertinent To Software Redesign Analysis [MOCA user's guide, 2003]

The MOCA tool is a JAVA application. It provides a Graphical User Interface to allow the user to enter various inputs required for solution processing. Some of these inputs have been mentioned briefly in the discussion above. A detailed explanation for all of the inputs is included in Appendix B. The user interfaces for collecting the inputs *pertinent to Software Redesign Analysis* will be depicted in this section. Clicking the appropriate buttons on the menu bar (shown in Figure 3.3) of the tool opens up these interfaces.


Mitigation Obsolescence Cost Analyzer (MOCA), Version 1.2.6
File Setup Inputs Compute/Results Help Calibration Emulator

MOCA
Mitigation of Obsolescence Cost Analysis

Version 1.3
June 2003

Electronic Systems Cost Modeling Laboratory
CALCE Electronic Products and Systems Center
University of Maryland

MESSAGE: Welcome! You are using Mitigation Obsolescence Cost Analyzer (MOCA)

Figure 3.3: The MOCA tool (*The Menu Bar is enclosed in the boxed region*)

3.3.1 Board-Specific Parts List

For every board in the system, MOCA stores a list of parts that belong to that board. Included within the parts list information is[6]:

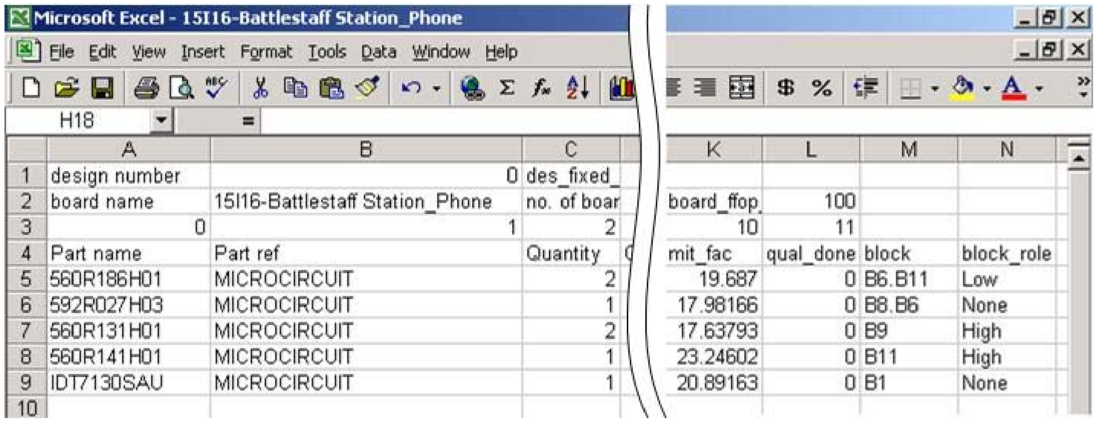
- *Part Category*: This input indicates the type of part being used. Several part categories are available in MOCA. These are: Microcircuit, Diode, Transistor, Integrated circuit, Semiconductor, Assorted and Custom Defined. The “Assorted” part type is used to represent an aggregate of parts and their instances. All the parts, which do not have any obsolescence or maintenance issues (e.g., most passives and mechanical devices), are lumped together into a single part to reduce computation time. A “Custom Defined” part is a part type for which no single standard part type could be used. When a new part is synthesized as a result of a design refresh, the obsolescence date is reset based on a default and a lifetime is obtained based on the part category of the modified part.

[6] Only those part properties that are important for the software-costing problem have been discussed here. A complete description of all the properties/units can be found in Appendix B.

- *Block Participation*: This indicates which functional block(s) the part participates in. In other words, it indicates which functional blocks require the hardware part in question to implement their functionality.
- *Block Role*: As noted above, "block participation" indicates which functional block a hardware part participates in. The input "block role" contains information on the significance of the hardware part in that functional block, relative to the other hardware parts in it. MOCA allows four possible choices for "block role". These are High, Average, Low and None signifying, as the names would suggest, the relative contribution of a hardware part in implementing the functionality of the functional block in question. Assigning a value to the input "block role" would draw on the system designer's experience to judge the relative significance of each hardware part vis-à-vis every functional block that it participates in.

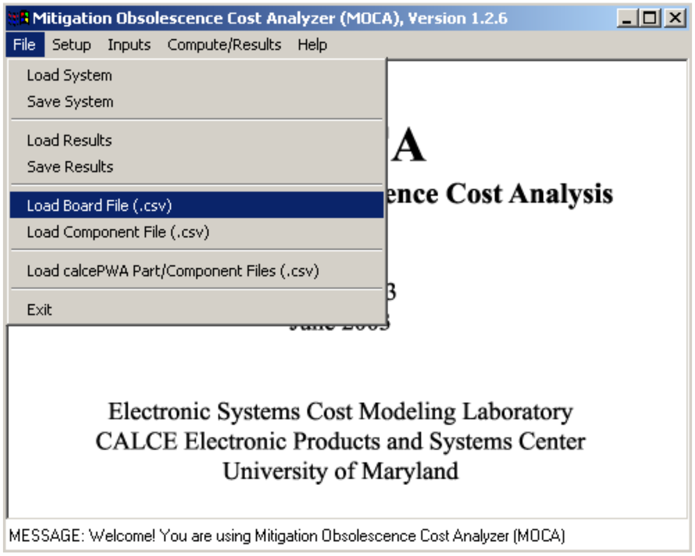
The inputs mentioned above are used to determine, when a hardware part is affected, which of the functional blocks are affected and to what degree. All this data is contained in the parts list. These lists are board specific, i.e., there is a part list for each board (containing information on all the parts contained by *that* board). The overall part list for the system is obtained by combining the part lists of each board.

The part lists for the boards are entered into MOCA using comma-delimited files. These are created using Excel spreadsheets. An example is shown in Figure 3.4
B6, B8, etc. in column M, are the names of functional blocks. In cases where a hardware part participates in more than one block, the names of both of these blocks is entered into the "block" column separated by a "." Symbol. These files can be loaded into MOCA using the interface shown in Figure 3.5.



	A	B	C	K	L	M	N
1	design number	0	des_fixed_				
2	board name	15I16-Battlestaff Station_Phone	no. of boar	board_ffop	100		
3	0	1	2	10	11		
4	Part name	Part ref	Quantity	mit_fac	qual_done	block	block_role
5	560R186H01	MICROCIRCUIT	2	19.687	0	B6.B11	Low
6	592R027H03	MICROCIRCUIT	1	17.98166	0	B8.B6	None
7	560R131H01	MICROCIRCUIT	2	17.63793	0	B9	High
8	560R141H01	MICROCIRCUIT	1	23.24602	0	B11	High
9	IDT7130SAU	MICROCIRCUIT	1	20.89163	0	B1	None
10							

Figure 3.4: An example of a comma-delimited file for loading a board specific parts list into MOCA

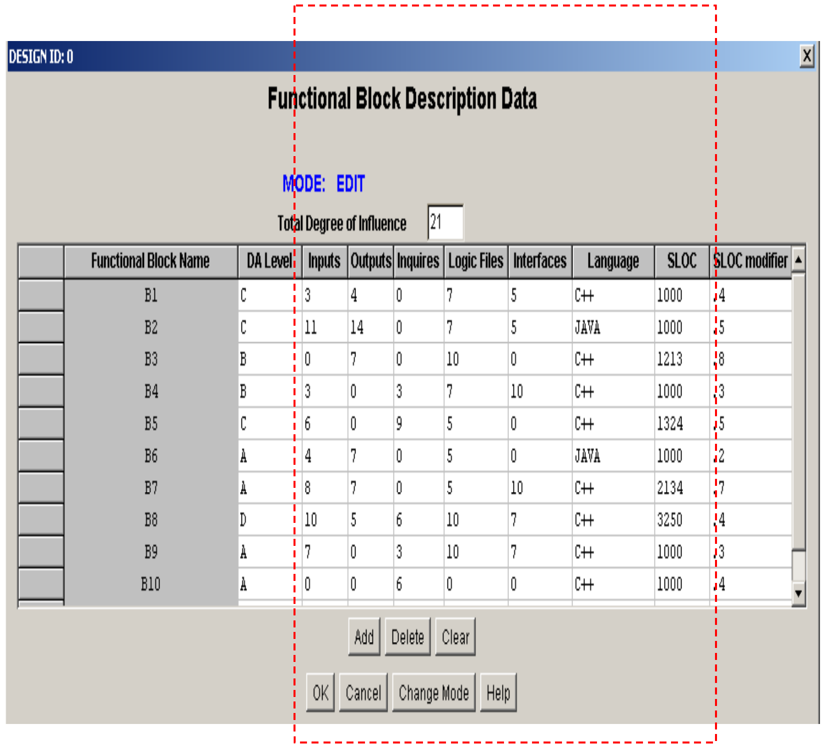

Mitigation Obsolescence Cost Analyzer (MOCA), Version 1.2.6
File
Setup
Inputs
Compute/Results
Help
Load System
Save System
Load Results
Save Results
Load Board File (.csv)
Load Component File (.csv)
Load calcePWA Part/Component Files (.csv)
Exit
A
ence Cost Analysis
Electronic Systems Cost Modeling Laboratory
CALCE Electronic Products and Systems Center
University of Maryland

MESSAGE: Welcome! You are using Mitigation Obsolescence Cost Analyzer (MOCA)

Figure 3.5: MOCA interface for loading a board-specific part list

3.3.2 Functional Blocks Data

Inputs for the functional blocks are collected using the interface shown in Figure 3.6. The user-inputs required for the purpose of describing these functional blocks are the ones shown in the boxed region. These are, Source Lines of Code (SLOC), Implementation Language (described in Appendix B), and the function point analysis parameters (Inputs, Outputs, Inquiries, Logic Files, Interfaces and Algorithms fields, all of which are explained in Appendix A).



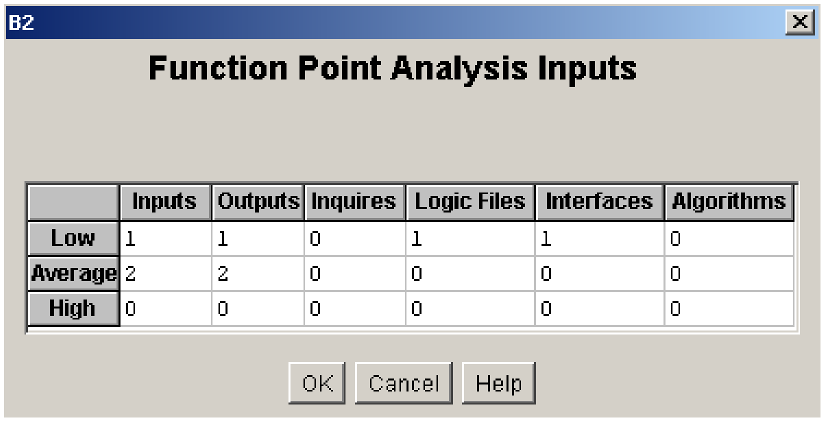
Functional Block Description Data

MODE: EDIT

Total Degree of Influence 21

Functional Block Name	DA Level	Inputs	Outputs	Inquires	Logic Files	Interfaces	Language	SLOC	SLOC modifier
B1	C	3	4	0	7	5	C++	1000	.4
B2	C	11	14	0	7	5	JAVA	1000	.5
B3	B	0	7	0	10	0	C++	1213	.8
B4	B	3	0	3	7	10	C++	1000	.3
B5	C	6	0	9	5	0	C++	1324	.5
B6	A	4	7	0	5	0	JAVA	1000	.2
B7	A	8	7	0	5	10	C++	2134	.7
B8	D	10	5	6	10	7	C++	3250	.4
B9	A	7	0	3	10	7	C++	1000	.3
B10	A	0	0	6	0	0	C++	1000	.4

Figure 3.6: Interface for collecting Functional Blocks inputs



Function Point Analysis Inputs

	Inputs	Outputs	Inquires	Logic Files	Interfaces	Algorithms
Low	1	1	0	1	1	0
Average	2	2	0	0	0	0
High	0	0	0	0	0	0

Figure 3.7: Inputs required for Function point Counting

The functional point analysis parameters are assigned values by clicking on the Functional Block Name. This opens up the dialog box shown in Figure 3.7.

3.3.3 PRICE S/COCOMO Data

Depending on which life cycle software cost analysis model (PRICE S or COCOMO) is used (both will be described in detail in the next section) there are some additional inputs that have to be provided.

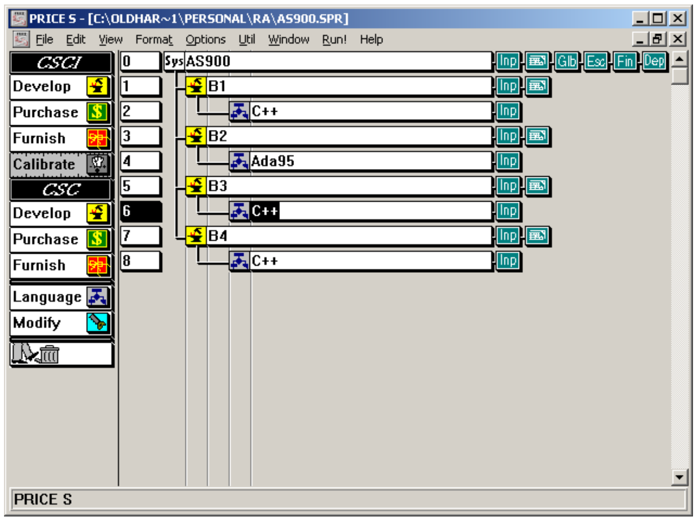
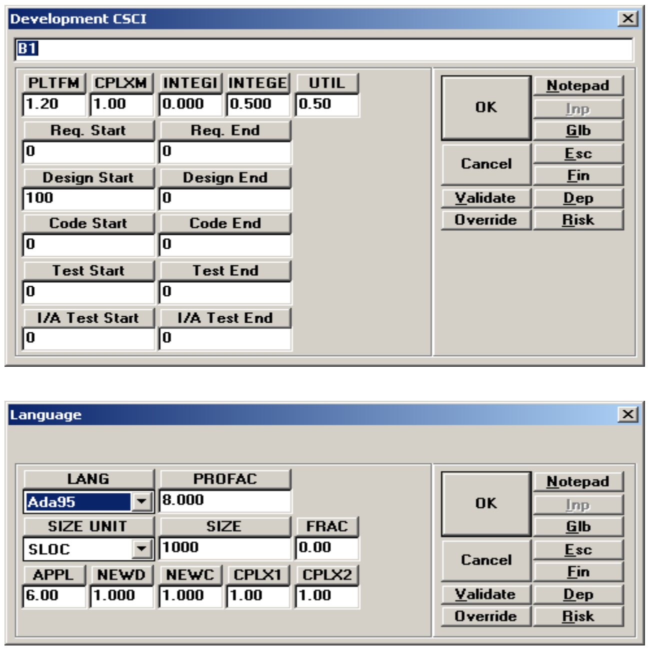


Figure 3.8: PRICE S EBS

PRICE S inputs are not collected directly by MOCA. Instead a PRICE S file is created with all the required inputs. This file is invoked during the MOCA analysis if PRICE S is to be used for modeling software costs. A sample PRICE S file, i.e., the Estimating Breakdown Structure (EBS), is shown in Figure 3.8. The EBS defines the structure of the software project. In this file “AS900” is the system and B1, B2, B3 and B4 are the names of the functional blocks.

The PRICE S interfaces for collecting some of the main inputs are shown in Figure 3.9. These inputs will be discussed in the next section.



Development CSCI

B1

PLTFM	CPLXM	INTEGI	INTEGE	UTIL
1.20	1.00	0.000	0.500	0.50

Req. Start	Req. End
0	0
Design Start	**Design End**
100	0
Code Start	**Code End**
0	0
Test Start	**Test End**
0	0
I/A Test Start	**I/A Test End**
0	0

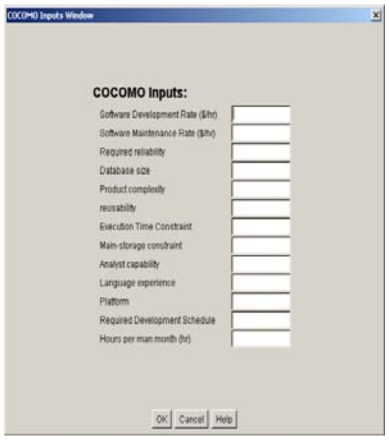
Language

LANG	PROFAC	
Ada95	8.000	
SIZE UNIT	**SIZE**	**FRAC**
SLOC	1000	0.00

APPL	NEWD	NEWC	CPLX1	CPLX2
6.00	1.000	1.000	1.00	1.00

Figure 3.9: PRICE S interface for collecting the main inputs

If the COCOMO model is used, then the inputs can be entered into MOCA directly using the interface shown in Figure 3.10. These inputs will also be discussed in the next section.



COCOMO Inputs Window

COCOMO Inputs:

Software Development Rate ($/hr)
Software Maintenance Rate ($/hr)
Required reliability
Database size
Product complexity
reusability
Execution Time Constraint
Main-storage constraint
Analyst capability
Language experience
Platform
Required Development Schedule
Hours per man month (hr)

Figure 3.10: MOCA interface for collecting COCOMO inputs

3.4 Cost Analysis Models

At present there are two cost analysis models for software development costs and maintenance are employed by MOCA, i.e., PRICE S and COCOMO. This section contains a brief summary of the salient points of these two models and also a discussion of the inputs they require.

3.4.1 PRICE S

The PRICE S model was originally developed by RCA (initially RCA Price, then GE Price, then Martin Marietta Price Systems, then Lockheed Martin Price, and now an independent company) as one of a family of models for hardware and software cost estimation. Developed in 1977 by Freiman and Park, [Park, 1988] Price S was the first commercially available detailed parametric software cost model to be extensively marketed and used. In 1987, the model was modified and re-validated for modern software development practices.

PRICE S Inputs

The primary input for the PRICE-S model is Source Lines of Code (SLOC). It is a count of "non-blank, non-comment lines" in the text of the program’s source code. SLOC as a software metric is used to measure the “amount of code” in a software program. It is typically used to estimate the amount of effort that will be required to develop a program; as well as to estimate productivity once the software is produced. A caveat is that the coding language must be specified as well. This is because a different amount of SLOC may be required to code up the same functionality in different languages. SLOC may be input by the user or computed using function point sizing models. Both the options are available in the PRICE S model. Other key inputs include:

1. Application (APPL): a measure of the type (or types) of software, described by one of seven categories (mathematical, string manipulation, data storage and retrieval, on-line, real-time, interactive, or operating system).
2. Productivity Factor (PROFAC): A calibrated parameter, which relates the software program to the productivity, efficiency/inefficiencies, software development practices and management practices of the development organization.
3. Complexities (CPLXM, CPLX1, CPLX2): Three complexity parameters that relate the project to the expected completion time, based on organizational experience, personnel, development tools, hardware characteristics, and other complicating factors.
4. Platform (PLTFM): the operating environment, in terms of specification, structure and reliability requirements.
5. Utilization (UTIL): Percentage of hardware memory or processing speed utilized by the software.
6. New Design/New Code (NEWD/NEWC): Percentage of new design and new code.
7. Integration (Internal) (INTEGI): Effort to integrate various software components together to form an integrated and tested software module.
8. Integration (External) (INTEGE): Effort to integrate various software modules together to form an integrated and tested software system.
9. Schedule (DSTART/DEND): Software project start and/or end dates.
10. Optional Input Parameters: Financial factors, escalation, risk simulation.

Processing

The PRICE-S algorithms are published [Parametric Estimating Initiative (PEI) Parametric Estimating Handbook] in the paper entitled "Central Equations of PRICE S" which is available from PRICE Systems. It states that PRICE-S computes a "weight" of software based on the product of instructions and application inputs. The productivity factor and complexity inputs are very sensitive parameters, which affect effort and schedule calculations. Platform is an exponential input; hence, it can be very sensitive too. The model based on the type or category of instructions calculates a new weighted design and code value. Both new design and code affect schedule and cost calculations. Internal integration input parameters affect the software module cost and the schedule for integrating and testing the software modules. The external integration input parameter is used to calculate software-to-software integration cost and schedule.

Outputs

PRICE-S computes an estimate in person effort (person hours or months). Effort can be converted to cost in dollars or other currency units using financial factors parameters. Software development schedules are calculated for nine DOD-STD-2167A[7] phases [Parametric Estimating Initiative (PEI) Parametric Estimating Handbook]: System Concept through Operational Test and Evaluation. Six elements of costs are calculated and reported for each schedule phase: Design Engineering, Programming, Data, Systems Engineering Project Management, Quality Assurance, and Configuration Management. The PRICE-S model also contains several optional outputs including over thirty graphs, Gantt charts, sensitivity matrices, resource expenditure profiles and schedule reports. In

[7] A US Department of Defense standard that specifies the overall process for the development and documentation of mission-critical software systems.

addition, Microsoft Project files, spreadsheet files, and risk analysis reports can be generated. The risk analysis report is a Cumulative Probability Distribution and is generated using either Monte Carlo or Latin Hypercube simulation.

Calibration

The PRICE-S model can be run in ECIRP (PRICE backwards) mode to calibrate selected parameters. The most common calibration is that of the productivity factor, which, according to the PRICE-S manual, tends to remain constant for a given organization. It is also possible to calibrate platform, application, and selected internal factors.

3.4.2 COCOMO

In 1981, Boehm developed the essential algorithms of the *co*nstructive *co*st *mo*del (COCOMO) [Boehm, 1981]. Since COCOMO algorithms were first provided to the general public, many commercial software-estimating tools have been derived from the COCOMO estimation method. COCOMO remains the only software-estimating model whose algorithms are not treated as proprietary. Boehm also developed COCOMO II, a revision to his original model, which is also available to the public [Boehm, 1995].

COCOMO: Software Size Estimation and Reuse

COCOMO uses a variation of the following model to estimate the equivalent number of lines of code:

$$KSLOC = KNSLOC + KASLOC\frac{\left(AA + SU + 0.4DM + 0.3CM + 0.3IM\right)}{100} \qquad (3.1)$$

KNSLOC is the size of the new software component expressed in thousands of lines of code. *KASLOC* is the size of the adapted or existing commercial off-the-shelf (COTS) software component expressed in thousands of adapted source lines of code. Five adjusting factors affect the final value of *KSLOC*. *SU* is the software understanding increment that is expressed as a percentage (ranging from 10 to 60%). *AA* expresses the degree of assessment and assimilation needed to determine whether a fully reused software module is appropriate to the application and to integrate its description into the overall product description (ranging from 0 to 8%). These first two factors are only used in the nonlinear reuse model. The three following modification factors are used in both the linear and nonlinear reuse estimation models. *DM* is the percentage of the adapted software's design that is modified in order to adapt it to the new objectives and environment. Similarly, *CM* is the percentage of the adapted software's code that is modified in order to adapt it to the new objectives and environment. Finally, *IM* is the percentage of effort required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

COCOMO: Software Development Effort Estimation

The software development effort, S_E, expresses the development effort in terms of man months (3.2). *A* is the constant used to capture the multiplicative effects on effort with projects of increasing size, having a default value of 4.44 [DeBardelaben, 1998]. Researchers at Georgia Tech increased this value from Boehm's original default of 3.6 [Boehm, 1981]. *B* is a scale factor which accounts for the relative economies or diseconomies of scale encountered for software projects of different sizes, having a default of 1.2, suggesting a diseconomy. The F_i's, or effort adjustment factors, are those cost drivers which model the effect of personnel, computer, product, and project attributes on software cost. These multipliers are summarized in Table 3.1.

$$Cost_{dev} = S_E\left(hrspermm\right)\left(rate_{dev}\right) \tag{3.2}$$

To calculate development cost, the effort in man months, S_E, is multiplied by hours per man month, set at 152, and then multiplied by the development rate per hour.

$$Staff_{dev} = \frac{S_E}{S_S} \tag{3.3}$$

The development staffing effort can be found by dividing the development effort in man-months by the development schedule in months, giving a number of development staff required estimate.

$$Prod_{dev} = \frac{KSLOC}{S_E} \tag{3.4}$$

$$S_E = A\left(KSLOC\right)^B \sum_{i=1}^{19} F_i \tag{3.5}$$

The productivity of the software staff can also be estimated, measured in thousands of source lines of code per man-month by dividing the total source lines of code from (3.1) by the software development effort.

COCOMO: Software Maintenance Effort Estimation

Software maintenance effort is found in a similar manner to development effort. A new factor, ACT, is introduced, and several effort adjustment factors change. ACT is the annual change traffic, which corresponds to the fraction of the software product's source code that undergoes change during a typical year, either through addition or modification. Two of the cost drivers, required reliability and use of modern programming practices, have different productivity multipliers due to differences in their relative impact on development and maintenance. In addition, required development schedule is irrelevant to maintenance, so it is set at 1.0. Cost of maintenance is measured in the same way as cost of development, and the staffing is measured simply by dividing the effort in man-months by months in a year.

$$M_E = ACT\left(A(KSLOC)^B \sum_{i=1}^{17} F_i \right) \tag{3.6}$$

$$Cost_{maint} = M_E(hrspermm)(rate_{maint}) \tag{3.7}$$

$$Staff_{maint} = \frac{M_E}{12} \tag{3.8}$$

COCOMO: Software Schedule Estimation

The development time equation is:

$$S_S = C(S_{E_{nom}})^D \frac{SCED}{100} \tag{3.9}$$

C is the constant used to capture the multiplicative effects on time with projects of increasing effort having a default of 6.2 [DeBardelaben, 1998]. Researchers at Georgia Tech increased this value from Boehm's original value of 2.5 [Boehm, 1981]. D is a scaling factor which accounts for the relative economies or diseconomies of scale encountered for projects of different required efforts, having a default of 0.32. Finally, SCED is the percent compression or expansion to the nominal deployment schedule. S_{Enom} is found similarly to the software development effort equation for S_E, but in this case no effort adjustment factors are taken into account, as shown in (3.10).

$$S_{E_{nom}} = A(KSLOC)^B \tag{3.10}$$

The level of accuracy of the estimate provided by this software cost model is directly proportional to the user's confidence in the software size estimate and the description of the development environment. Through calibration in multiple areas, the potential risks associated with estimating software development can be effectively reduced.

Summary of the Major Inputs to the COCOMO Model

1. Required Reliability (RELY) - This is the measure of the extent to which the software must perform its intended function over a period of time. If the effect of a software failure is only slight inconvenience then RELY is low. If a failure would risk human life then RELY is very high.
2. Database Size - This measure attempts to capture the affect large data requirements have on product development. The rating is determined by calculating D/P. The reason the size of the database is important to consider is because of the effort required to generate the test data that will be used to exercise the program.

$$\frac{D}{P} = \frac{DataBaseSize(bytes)}{\Pr ogramSize(SLOC)} \qquad (3.11)$$

It is rated as low if D/P is less than 10 and it is very high if it is greater than 1000.

3. Required Reusability - This cost driver accounts for the additional effort needed to construct components intended for reuse on the current or future projects. This effort is consumed with creating more generic design of software, more elaborate documentation, and more extensive testing to ensure components are ready for use in other applications.

4. Execution-time constraint - This is a measure of the execution time constraint imposed upon a software system. The rating is expressed in terms of the percentage of available execution time expected to be used by the system or subsystem consuming the execution time resource. The rating ranges from nominal, less than 50% of the execution time resource used, to extra high, 95% of the execution time resource is consumed.
5. Main-storage constraint - This rating represents the degree of main storage constraint imposed on a software system or subsystem. Many applications consume whatever resources are available, making these cost drivers still relevant. The rating ranges from nominal, less that 50%, to extra high, 95%.
6. Analyst Capability - Analysts are personnel that work on requirements, high-level design and detailed design. The major attributes that should be considered in this rating are Analysis and Design ability, efficiency and thoroughness, and the ability to communicate and cooperate. Analysts that fall in the 15th percentile are rated very low and those that fall in the 95th percentile are rated as very high.
7. Applications Experience - This rating is dependent on the level of applications experience of the project team developing the software system or subsystem. The ratings are defined in terms of the project team's equivalent level of experience with this type of application. A very low rating is for application experience of less than 2 months. A very high rating is for experience of 6 years or more.

8. Language and Tool Experience - This is a measure of the level of programming language and software tool experience of the project team developing the software system or subsystem. Software development includes the use of tools that perform requirements and design representation and analysis, configuration management, document extraction, library management, program style and formatting, consistency checking, etc. In addition to experience in programming with a specific language the supporting tool set also effects development time. A low rating is given for experience of less than 2 months. A very high rating is given for experience of 6 or more years.
9. Software Development Schedule - This rating measures the schedule constraint imposed on the project team developing the software. The ratings are defined in terms of the percentage of schedule stretch-out or acceleration with respect to a nominal schedule for a project requiring a given amount of effort. Accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined due to lack of time to resolve them earlier. A schedule compress of 74% is rated very low. A stretch-out of a schedule produces more effort in the earlier phases of development where there is more time for thorough planning, specification and validation. A stretch-out of 160% is rated very high.

These inputs along with their valid ranges have been tabulated in Table 3.1

Table 3.2: Effort Adjustment Factors

Cost Driver	**Factors**
Product Attributes	
Required reliability	0.75 to 1.40
Database size	0.94 to 1.16
Product complexity	0.70 to 1.65
Required reusability	1.00 to 1.50
Computer Attributes	
Execution-time constraint	1.00 to 1.66
Main-storage constraint	1.00 to 1.56
Personnel Attributes	
Analyst capability	0.71 to 1.46
Applications experience	0.82 to 1.29
Language experience	0.95 to 1.14
Project Attributes	
Required development schedule	1.00 to 1.23

It should be noted that COCOMO is an "open model", i.e., it is not proprietary. For that reason, the COCOMO model's equations have been programmed into MOCA. This is in contrast with PRICE S, which being proprietary necessitates an external link up between MOCA and itself.

3.5 Remarks

The solution architecture explained in Section 2.2, in combination with the MOCA methodology and Cost analysis models (described in Sections 3.2 and 3.4 respectively) provides a framework to assess the impact of hardware part obsolescence on the life cycle cost of system software. This framework was implemented using the MOCA tool on two case studies. The case studies are described in Chapter 4.

Chapter 4– Results

4.1 Introduction

The application of the solution architecture discussed in Chapter 2 will be described in this chapter. The first attempt to apply the methodology was carried out on a Honeywell engine controller. The details of this example will be explained in Section 4.2. Although no software-specific data was available for the Honeywell example, the example served as a heuristic, which helped in detecting and demonstrating the salient features of the developed methodology. The methodology was then applied to a Navy test case provided by Price Systems. The cost value generated by MOCA in the Navy test case example was validated against the actual costs borne by the Navy. The Navy test case example and its validation are the subject of Section 4.3.

4.2 Application of the Methodology to the AS900's FADEC System

Honeywell International, Inc. manufactures the AS900 engine. The solution methodology discussed in the previous chapters was implemented on the AS900 engine's Full Authority Digital Electronic Controller (FADEC)[8]. The steps in this implementation were:

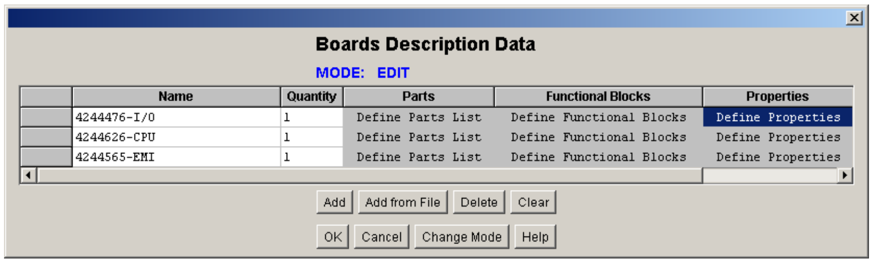
1) *Identification of elements at the "board-level" in the system hierarchy* – The following is a list of the boards in the system:

a) EMI

b) I/0

c) CPU

[8] Referred to as AS900 in the remainder of this chapter



Boards Description Data

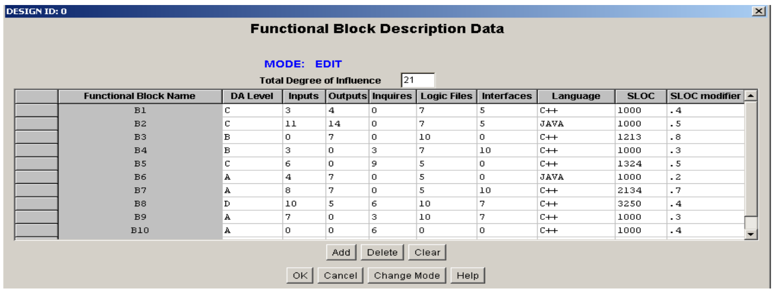
MODE: EDIT

	Name	Quantity	Parts	Functional Blocks	Properties
	4244476-I/O	1	Define Parts List	Define Functional Blocks	Define Properties
	4244626-CPU	1	Define Parts List	Define Functional Blocks	Define Properties
	4244565-EMI	1	Define Parts List	Define Functional Blocks	Define Properties

Figure 4.1: Boards Dialog box

These three boards were used in the analysis. It should be noted that in addition to the boards, the AS900 also contains two sensors and various mechanical elements that are necessary to assemble the boards into an enclosure. Once the board specific part lists had been loaded into MOCA using comma-delimited files as described in Section 3.3.1, board characteristics within MOCA could be accessed using the "Boards Dialog box" shown in Figure 4.1.

2) *Partitioning the system into functional blocks* – In the absence of any data on the system software, some fictional functions to be performed by the AS900 system were hypothesized. Each of these functions was designated as one functional block.



DESIGN ID: 0

Functional Block Description Data

MODE: EDIT

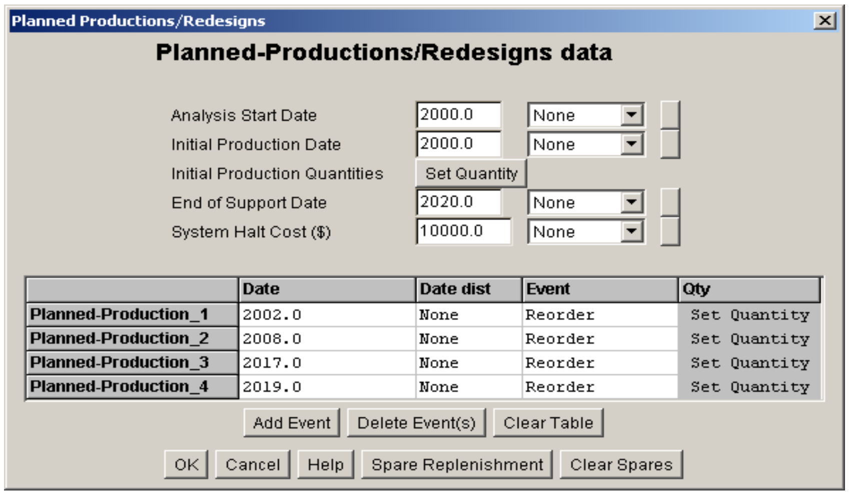
Total Degree of Influence 21

	Functional Block Name	DA Level	Inputs	Outputs	Inquires	Logic Files	Interfaces	Language	SLOC	SLOC modifier
	B1	C	3	4	0	7	5	C++	1000	.4
	B2	C	11	14	0	7	5	JAVA	1000	.5
	B3	B	0	7	0	10	0	C++	1213	.8
	B4	B	3	0	3	7	10	C++	1000	.3
	B5	C	6	0	9	5	0	C++	1324	.5
	B6	A	4	7	0	5	0	JAVA	1000	.2
	B7	A	8	7	0	5	10	C++	2134	.7
	B8	D	10	5	6	10	7	C++	3250	.4
	B9	A	7	0	3	10	7	C++	1000	.3
	B10	A	0	0	6	0	0	C++	1000	.4

Figure 4.2: Functional Blocks Dialog box

There were a total of 11 such blocks called B_1, B_2…B_{11}. "Functional blocks" characteristics within MOCA could be accessed using the "Functional Blocks Dialog box" shown in Figure 4.2. The Function point counting parameters and the PRICE S attributes (APPL, SLOC, PROFAC, CPLXM, PLTFM, UTIL, NEWC, INTEGI, INTEGE, DSTART) - discussed in Chapter 3 - for each of these functional blocks - were determined and entered into MOCA and the PRICE S database respectively.

3) *Entering Obsolescence data, cost data and schedule of planned productions* – As explained in Section 3.3.1, the parts that do not have obsolescence issues within the life span of the system were combined into "lumped parts" with a total cost and no obsolescence data, in order to simplify solution processing. Obsolescence dates for the parts were calculated using the CALCE obsolescence model. For each board, board-specific part data (quantity, obsolescence data, cost data, block participation and block role) was entered into MOCA using comma-delimited files as described in Chapter 3. A production schedule was also entered into MOCA as shown in Figure 4.3



Planned-Productions/Redesigns data

Analysis Start Date	2000.0	None
Initial Production Date	2000.0	None
Initial Production Quantities	Set Quantity	
End of Support Date	2020.0	None
System Halt Cost ($)	10000.0	None

	Date	Date dist	Event	Qty
Planned-Production_1	2002.0	None	Reorder	Set Quantity
Planned-Production_2	2008.0	None	Reorder	Set Quantity
Planned-Production_3	2017.0	None	Reorder	Set Quantity
Planned-Production_4	2019.0	None	Reorder	Set Quantity

Figure 4.3: Production Schedule for the AS900 system

Taking the base/start year as 2000, the reorder dates were set as 2002 (504), 2008 (664), 2017 (752) and 2019 (752). The number in the parenthesis denotes the quantity to be reordered at that date.

The MOCA tool was run after the above-mentioned pre-processing steps were carried out. The results obtained have been discussed below. It should be noted that for all the graphs shown below, the horizontal axis variable – "mean of redesign dates" – implies the mean of all the redesign dates in the design refresh plan in question, i.e., for a design refresh plan that has three redesigns planned in the years Y_1, Y_2 and Y_3, the total life cycle cost (which is the vertical axis variable) will be plotted for the year "Y" such that, $Y = \left(\frac{Y_1 + Y_2 + Y_3}{3}\right)$. By clicking on a point on the graph MOCA allows the user to determine what the individual dates within the design refresh plan are.

4.2.1 Part Obsolescence impact on life cycle cost (Software and Hardware) of the AS900 system

Figure 4.4 shows the result from an analysis that disregarded the impact of part obsolescence on system software. The solution that gives the minimum life cycle cost is a design refresh plan with 3 redesigns scheduled for the calendar dates 2002, 2008 and 2017. Figure 4.5 shows the result from an analysis that took into account the impact of part obsolescence on both system hardware and system software. In this case the solution for minimum life cycle cost is a design refresh plan with 2 redesigns scheduled for the calendar dates 2008 and 2017. Thus, the "best case" solution (optimum refresh plan) changes when we adopt the solution methodology developed in this thesis.

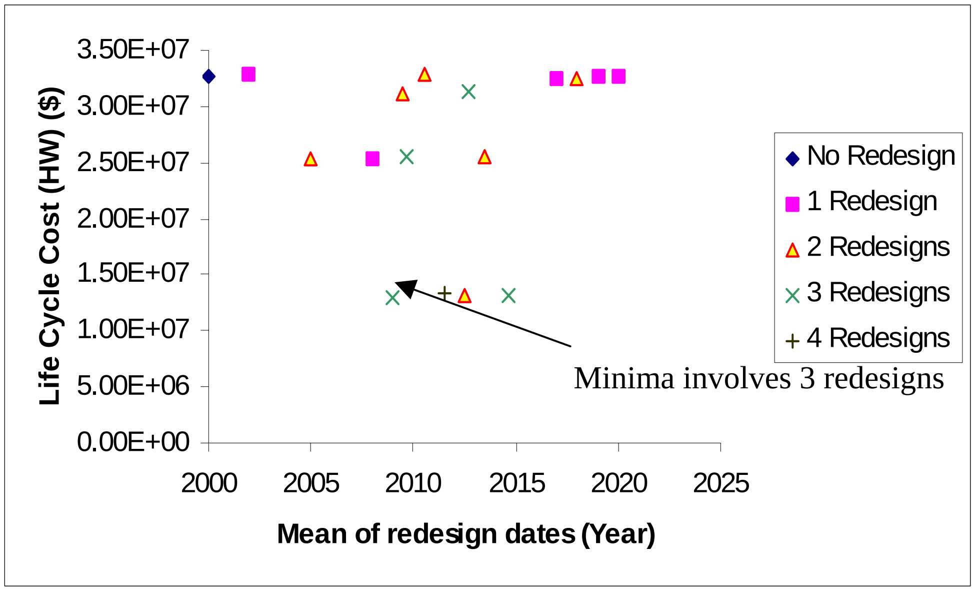


Figure 4.4: Life cycle cost accrued due to impact of part obsolescence on system hardware alone

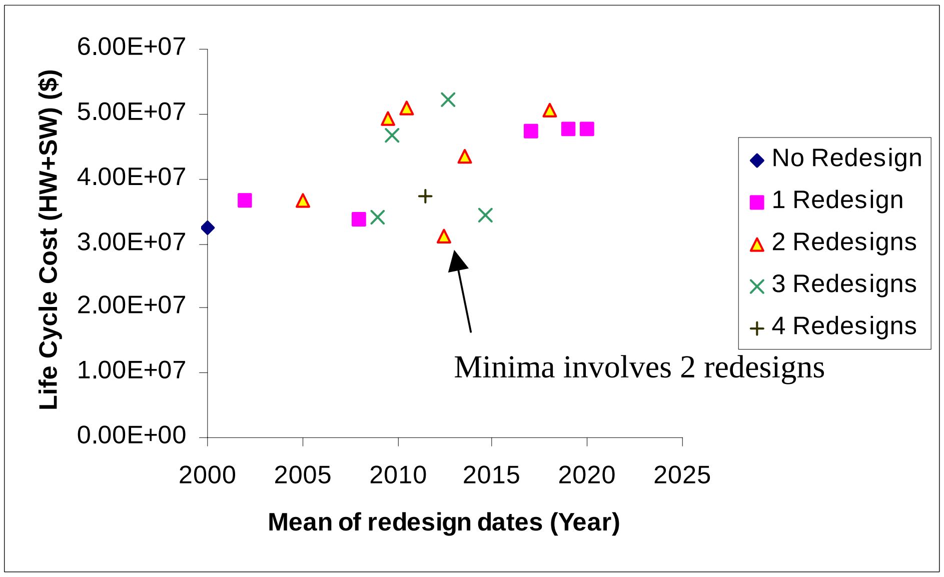


Figure 4.5: Life cycle cost accrued due to impact of part obsolescence on both system software and system hardware

4.2.2 Part Obsolescence impact on life cycle cost of AS900 system software alone

Figure 4.6 shows the result from an analysis that took into account the impact of part obsolescence on system software alone (i.e., it neglected the costs due to the impact on system hardware). It shows that the life cycle cost increases as the "number of redesigns in a design refresh plan" increases.

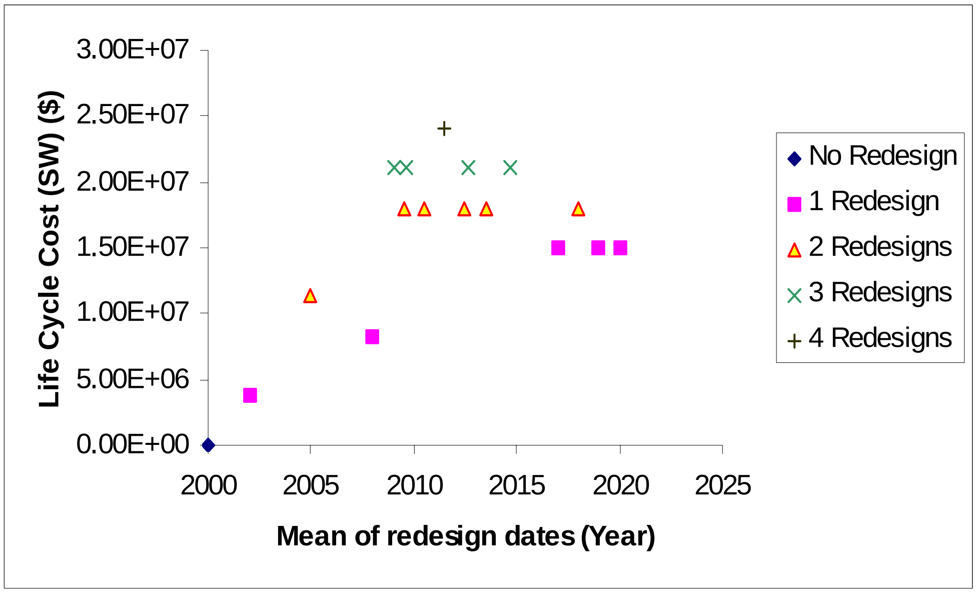


Figure 4.6: Life cycle cost accrued due to impact of part obsolescence on system software alone

This trend is because software redesign analysis, for the present case study, did not include recurring expenditures. Not doing so can make the analysis inaccurate because software maintenance, which *is* a recurring expenditure, does exist. However, it was not taken into account because the costs associated with it are insignificant compared to the hardware costs of the case study example. If we were to introduce a recurring cost into the system software redesign process, of the same order as hardware costs, then we would expect the above trend to change.

To that end, a hypothetical constraint was introduced. This constraint was: "For every five years that the system goes without software redesign there will be an expenditure of 20 million dollars for maintenance of system software." The results obtained by adding this constraint are depicted in Figure 4.7. As expected, they do show a departure from the trend of Figure 4.6.

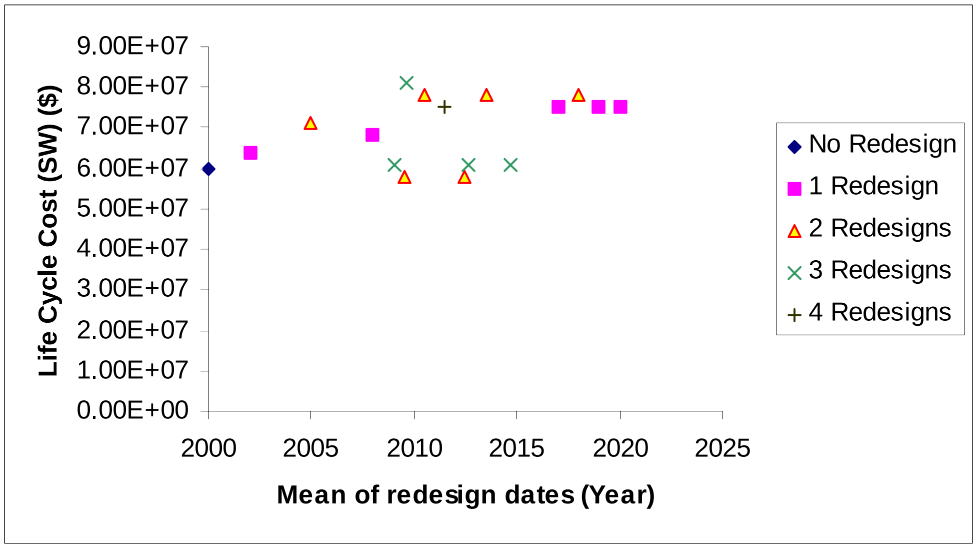


Figure 4.7: Life cycle cost accrued due to "impact of part obsolescence on system software" *and* "software maintenance"

4.2.3 Impact of "Design refresh look-ahead time"

At each design refresh activity MOCA looks ahead and addresses obsolescence related issues for all the parts expected to go obsolete within a user-specified range of time from the date of the design refresh activity in question. This range is called *"Design refresh look-ahead time"*. Figures 4.8 through 4.10 show the plots of life cycle cost of software obtained by varying the design refresh look-ahead time. Two salient points are:

- "Design refresh plans" with a fewer number of design refresh activities entail higher life cycle costs due to increased look-ahead time as compared to plans that have a larger number of design refresh activities scheduled – the latter remaining largely unaffected.
- "Design refresh plans" with a lower "mean of redesign dates" are affected more and they incur increased life cycle costs as the look-ahead time is increased.

It can be seen that as the design refresh look-ahead time is increased, the costs for design refresh plans having the same number of design refresh activities, tend to plateau out i.e., regardless of the mean of the redesign dates, all design refresh plans having "*n*" number of design refresh activities (for Figures 4.8-4.10; *n = 1,2,3 or 4*) entail the same life cycle cost as the look ahead time is increased. This combined with the fact that life cycle cost of software increases as the "number of redesigns in a design refresh plan" increases (as seen in Figure 4.6) implies that as the look-ahead time is increased, the number of redesigns for minimum cumulative life cycle cost (i.e., both software and hardware) has to either remain the same or decrease.

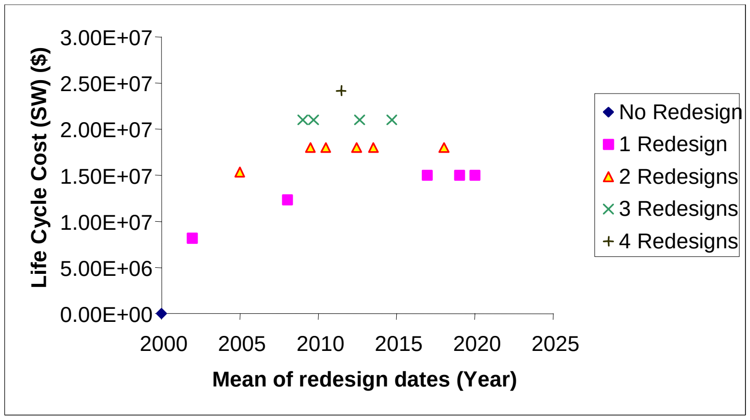


Figure 4.8: Look ahead time of 3 years

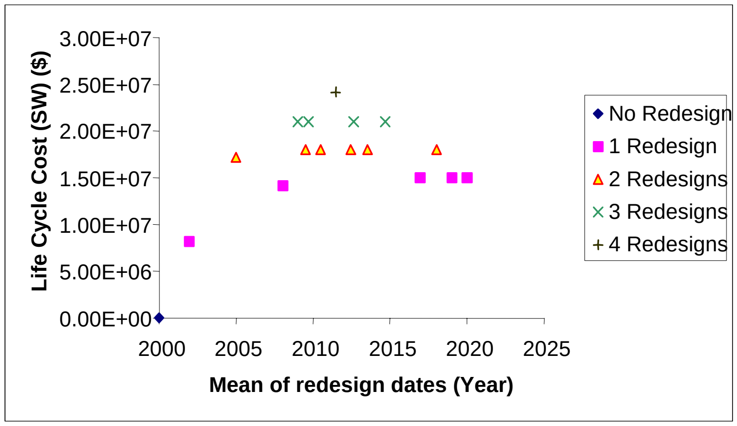


Figure 4.9: Look ahead time of 5 years

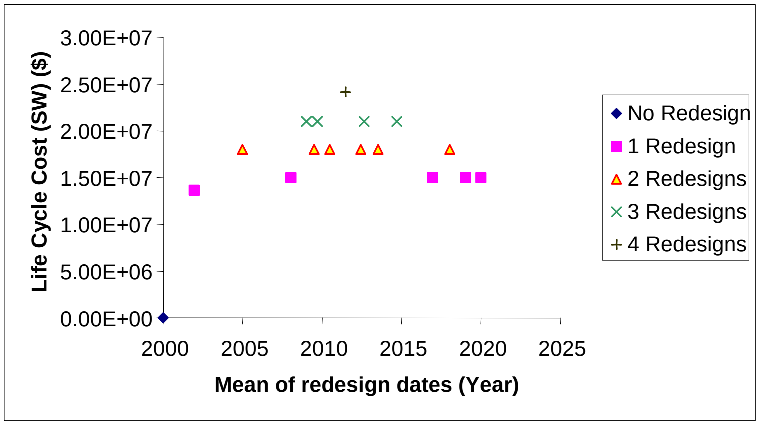


Figure 4.10: Look ahead time of 10 years

4.3 Validation Of the "Software Design Refresh" Analysis Methodology

The approach to quantifying the impact of hardware obsolescence on software redesign, as developed in this thesis, was implemented for a Navy test case (VH-60N Digital Cockpit Upgrade Program). This test case entailed costing software change arising because of a computer upgrade in a mission critical system. The objective of this case study was to compare the software change cost generated by MOCA with the actual cost borne by the Navy. Price Systems provided the data used for this validation.

The computer hardware upgrade is not performed only to manage obsolescence. However the reason for the upgrade does not matter for the purpose of validating the developed methodology because this methodology, in essence, calculates the cost of changing software precipitated by a hardware change. This hardware change could be due to part obsolescence, a general upgrade, or a combination of the two – as in this case.

The application of the methodology and validation will be discussed in the following sections.

4.3.1 Implementation of the methodology

- *Identification of elements at the "board-level" in the system hierarchy* – The following is a list of the boards that contained parts which were upgraded/redesigned in the system:
    a) 15641-VLF Transmit Terminal (Block I) - B Kit
    b) 15811-ARC-171C
    c) 15812-HPA w/ Mount
    d) 15813-Modem
    e) 15G11-Agilent Tech RF Test Set

f) 15H10-KG-33 REPL

g) 15I11-Raytheon Small Switch

h) 15I12-Transmux

i) 15I13-Comm Control Unit

j) 15I14-FD Crew Station

k) 15I15-Comm Crew Station

l) 15I16-Battlestaff Station/Phone

m) 17510-Legacy Interface Converter

n) 17611-Printer

o) 17614-RAID Storage

p) 17615-Network Encryptors

q) 17616-Ethernet Switches (2924XL)

r) 17618-IP Security Router (2621)

s) 17619-Network Protocol Processor

It should be noted that the parts-list was not available. Hence, fictitious bills of materials (parts lists) were created for these boards. Using the limited data that was available on the hardware to software mapping, i.e., which board participates in which functional block(s) and also how critical is the former for the functioning of the latter, the parts on these boards were assigned to the different blocks. The board specific part-lists for these boards were loaded into MOCA using comma-delimited files as described in Section 3.3.1.

- *Partitioning the system into functional blocks* – The main functions being executed by the system software in the case study system were identified and that was the basis for partitioning the system into the following functional blocks.

  a) Start Up & System Services (B1)

  b) Control (B2)

  c) Message Processing (B3)

  d) User Interface (B4)

  e) Communication Control (B5)

  f) Input Output Process (B6)

  g) Top Secret Services (B7)

  h) Test And Simulation (B8)

  i) Voice Functions (B9)

  j) Internal Flight Deck Communication (B10)

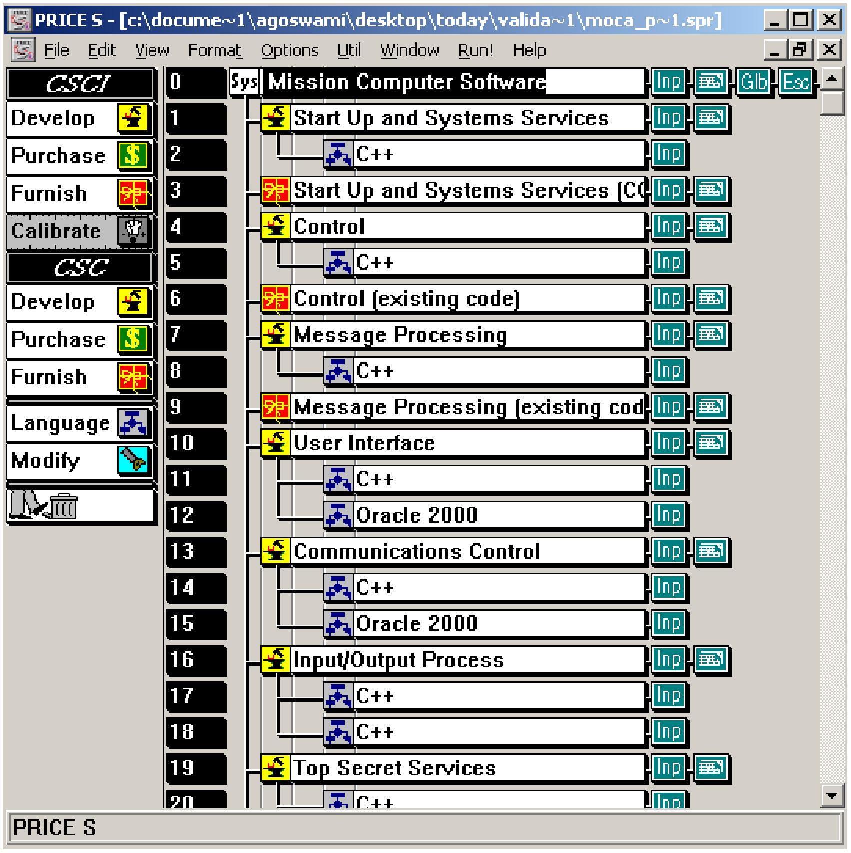
  k) VLF Transmit Terminal (B11)

The name in the parenthesis is the one used as the identifier inside MOCA and also, for convenience, at several places in the text of this thesis. The function point counting parameters were not available for this case. Therefore, the SLOC values provided for the functional blocks were used directly. The PRICE S attributes (APPL, SLOC, PROFAC, CPLXM, PLTFM, UTIL, NEWC, INTEGI, INTEGE and DSTART) - discussed in Chapter 3 – were determined for each of these functional blocks and entered into PRICE S database. The PRICE S EBS used for this purpose is shown in Figure 4.11. For all the functional blocks the values for PROFAC, CPLXM, PLTFM, UTIL, NEWC, INTEGI, INTEGE, and DSTART[*] were set to be 5.2, 1.0, 1.8, 0.5, 1.0, 0.5, 0.5 and 1004[9] respectively. Also, all the functional blocks used C++ as the implementation language. The SLOC and APPL values are tabulated in Table 4.1

Table 4.1: SLOC and APPL values for the functional blocks in the test case.

Name of functional block	APPL	SLOC
B1	9.99	74,688
B2	6.57	5,673
B3	2.44	10.056
B4	3.50	33,200
B5	6.47	2,492
B6	7.17	88,304
B7	6.16	816
B8	6.00	27,054
B9	5.95	13,03,68
B10	6.16	46,050
B11	8.23	46,675

[*] PRICE S requires this input to be in the MMYY format.


PRICE S - [c:\docume~1\agoswami\desktop\today\valida~1\moca_p~1.spr]

File Edit View Format Options Util Window Run! Help

CSCI	0	Mission Computer Software
Develop	1	Start Up and Systems Services
Purchase	2	C++
Furnish	3	Start Up and Systems Services (C
Calibrate	4	Control
CSC	5	C++
Develop	6	Control (existing code)
Purchase	7	Message Processing
Furnish	8	C++
	9	Message Processing (existing cod
Language	10	User Interface
Modify	11	C++
	12	Oracle 2000
	13	Communications Control
	14	C++
	15	Oracle 2000
	16	Input/Output Process
	17	C++
	18	C++
	19	Top Secret Services
	20	C++

PRICE S

Figure 4.11: PRICE S Work Breakdown Structure for the test case

- *Determining events to be placed on the system timeline* - A single reorder event was placed on the system timeline at the calendar date of 2004. Since our objective was to cost software change due to hardware change, the obsolescence dates and mitigation strategy for the parts were assigned such that they would be redesigned during the design refresh activity in 2004.

4.3.2 Validation

The MOCA tool was run after the steps outlined in Section 4.3.1 were completed. The resulting plot is shown in Figure 4.12.

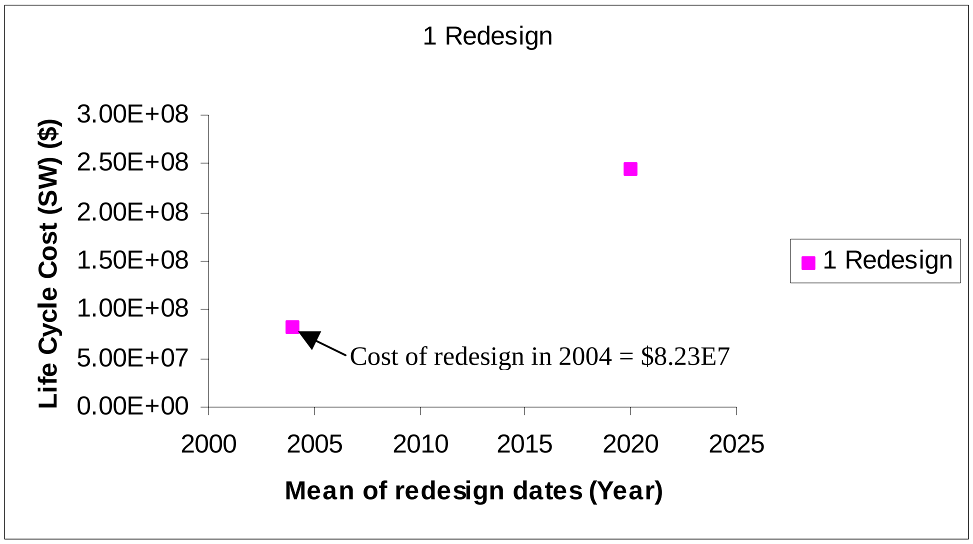


Figure 4.12: Life cycle cost accrued due to "impact of part obsolescence on system software" *and* "software maintenance"

It was determined from the plot that the cost entailed in developing new software due to redesigning the parts was $82.3 million. This agrees well with the actual cost of $90,742,245, which is tabulated in Table 4.2

Table 4.2: Actual cost incurred by the Navy *(Data provided by PRICE systems)*

Functional Block	**Cost**
Start Up & System Services	$11,226,435
Control	$343,695
Message Processing	$248,655
User Interface	$11,707,410
Communications Control	$1,408,770
Input & Output	$29,034,885
Top Secret Services	$42,570
Test and Simulation	$2,035,440
Voice Functions	$10,593,825
Internal Flight Deck Comm.	$3,667,620
VLF Transmit Terminal	$2,903,010
Sum by Component	$73,212,315
System Costs	$17,529,9309
Total Costs	$90,742,245

4.4 Remarks

Successful validation of the methodology developed in this thesis demonstrates that it has been structured correctly. A salient feature brought out by the case study of Section 4.2 is that as software life cycle costs tend to dominate hardware life cycle costs, it is more beneficial to have fewer design refreshes during the system's lifetime.

The fact that the impact of hardware part obsolescence on system software changes the optimum design refresh plan, as demonstrated in Section 4.2.1, clearly emphasizes the need to take software redesign analysis into account during life cycle management planning.

Chapter 5– Conclusions and Future Work

5.1 Summary

This thesis presents a methodology to determine the best points in time in the lifetime of a sustainment-dominated long field life system to schedule design refreshes when both software and hardware are considered. Life cycle cost of the system is the metric that has been used to determine the best solution. The methodology was applied to a test case provided by the Navy (VH-60N Digital Cockpit Upgrade Program).

5.2 Contributions

- This is one of the first attempts at studying the impact of part obsolescence on life cycle costs accrued by system software. There exists some literature on case-specific software change precipitated by hardware-change and the costs thereof. However, there are no general methodologies to be applied in such cases. This thesis presents the first attempt in that direction.

- This thesis demonstrates that revisions to software necessitated by the new parts introduced during the system's field life can involve significant costs, sometimes exceeding that required for hardware design and qualification.
- This thesis provides a general framework to tackle the problem of software obsolescence, which arises when a software vendor suddenly discontinues support or there is migration to different software platforms. Tackling the problem of software obsolescence often involves switching to a different implementation language. A different language implies a different number of lines of code even if

new functionality is not being developed, which may result in incurring costs over the software life cycle. MOCA can calculate the costs arising out of switching from one language to another.

5.3 Future Work

Future work in three directions is described below:

5.3.1 Spare Replenishment

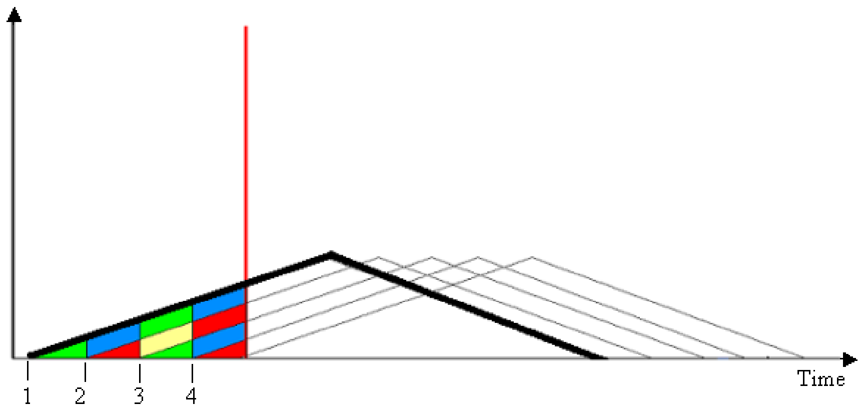


Figure 5.1: Model for spare replenishment

MOCA already has a spare replenishment model built within it. This model calculates the number of spare units that need to be produced to replace existing units that have failed. Each reorder event is treated as a candidate for spare replenishment. Generation of spares is important from the point of view of software redesign analysis even though the cost to develop new software is independent of the number of hardware units that it is going to run on.

This is because every time a new batch of hardware units is to be fielded (due to either spare replenishment or a reorder event), newly developed software has to be installed and perhaps also tested on all the units. Thus, a portion of the total cost involved in software redesign is dependent on the number of hardware units being fielded. A new model for doing so has been outlined below. It needs to be implemented.

Consider a system with a triangular distribution of time to failure. Let the origin of the time axis in Figure 5.1 be the time when it is put in the field. Let the points 1,2,3 and 4 on the timeline denote reorder events. In this model each reorder event is viewed as a candidate for replenishment

At each reorder event – the originally planned number of units ***plus*** the number of spare units to replace the ones that have failed prior to the reorder event have to be fielded. Each triangle in the figure represents the distribution of time to failure for the lot of units (spares plus originally planned) supplied at that reorder.

Thus, each lot of units (spares generated plus the originally planned units) at a reorder event follows a failure distribution identical to that of other lots but these distributions are offset from each other on the time axis. By keeping track of all of these lots, the spares to be generated can be calculated. The method for doing so is illustrated below:

For figure 5.1, let,

- Q1 units supplied at Reorder 1
  - A fraction "$\eta 1_{12}$" (green region between 1 and 2) of this lot fails before Reorder 2
  - A fraction "$\eta 1_{23}$" (red + blue region between 2 and 3) of this lot fails between Reorder 2 and Reorder 3…and so on.
- At Reorder 2, originally planned Q2 units + ($\eta 1_{12}$) Q1 units supplied

- A fraction "$\eta 2_{23}$" (red region between 2 and 3) of this lot fails between Reorder 2 and Reorder 3
- A fraction "$\eta 2_{34}$" (yellow + green region between 3 and 4) of this lot fails between Reorder 2 and Reorder 3…and so on.

These fractions "η" can be found out in MOCA by calculating the area under the curve between two reorders for the given probability distribution.

5.3.2 Software Design Refresh by Fault Tree Based Analysis of Functional Blocks

A methodology employing fault tree based analysis of functional blocks for software design refresh analysis was described in Section 2.3. This approach has not been implemented in MOCA yet. Doing so in the future will enable MOCA to take into account software reliability issues as well because fault trees are commonly used for studying software reliability. There is literature on combining Bayesian Belief Networks (BBN) with fault trees to improve software reliability analysis in complex systems [Pai, 2001]. Therefore, this might be a worthwile option to explore in the future.

5.3.3 Improving Efficiency

There is scope to come up with better techniques that quantify the impact of an affected part on the functional block(s) that it participates in. Also, the communication link between MOCA and PRICE S needs to be made more efficient by reducing the amount of data that is required to go back and forth at this point in time.

.

Appendix A - Function/Feature Point Counting

A.1 Overview

In this section, the factors used in Function/Feature Point Counting will be discussed. This will be followed by definitions that are relevant to these counting techniques. Lastly, the procedure to obtain the “counts” and size software using them will be explained.

A.1.1 Introduction to Function/Feature Point Counting and the factors used in these techniques

In the early stages of a software development process, the designer is concerned about the functionality to be delivered by the software, the development time, and the development cost. The ability to monitor these elements is influenced by a myriad of factors including the complexity of the language, the availability of skilled resources, and the techniques and methods used. An accurate “cost-per-unit-of-functionality” measure will produce results that are affected by all of these factors - but the unit of functionality itself cannot be altered by these factors; it must remain constant. It is for this reason that using source lines of code as a unit-of- functionality measure is problematic because the deliverable size varies based on language complexity (different languages produce different line-of-code counts for the same amount of functionality). Thus, a line of code measure will not provide a consistent unit-of- functionality measure across multiple languages.

A unit-of-functionality measure must be able to accurately quantify the functionality (value) being delivered. It was this rationale that led to the development of Function point counting.

In the late 1970's and early 1980's the software measurement technique, termed "Function Points" was introduced, [Albrecht, 1981]. Albrecht hypothesized that the observable functionality of software (in the context of this thesis - a functional block) could be enumerated accurately in terms of the following five items [Dreger, 1989]:

1. Outputs – Items of information processed by the functional block for the end user
2. Inputs – Items of data sent by the user to the functional block for processing and to add, change, or delete something.
3. Inquiries – Considered a simple output, they are direct inquiries into a database or master file that look for specific data, use simple keys, required immediate response, and perform no update functions
4. Logic Files – Data stored for a functional block, as viewed by the user
5. Interfaces – Data stored elsewhere by another functional block but used by the one under evaluation

These five functional elements are assessed based on their complexity and used to evaluate an unadjusted function point count. The next step in the methodology involves evaluating a series of general systems characteristics (GSCs), which include such things as performance, configuration, complexity, and reusability. The evaluation of the GSCs produces an adjustment factor, which is then applied to the unadjusted function point count for a final function point calculation. All this will be explained in greater detail in the following sub-sections.

The main strengths of function point metrics are [Jones, 2000]:

- Function points stay constant regardless of programming languages used.
- Function points are supported by many software cost estimating tools.
- Function points can be mathematically converted into number of logical code statements for many languages.

The weaknesses of function point metrics are [Jones, 2000]:

- Accurate counting requires certified function point specialists.
- Function point counting can be time-consuming and expensive.
- Function point counting automation is of unknown accuracy.
- Function point counts are erratic for applications or systems below 15 function points in size.
- Function point variations have no conversion rules to International Function Point User's Group (IFPUG) function points.
- Many function point variations have no "backfiring" conversion rules, making it difficult to convert from SLOC to a function point count.

Function point analysis is considered by many to be the most accurate and effective software metric ever developed. Counting accuracy by certified function point counters was found to have an accuracy of plus or minus 10% of actual software size in a study commissioned by IFPUG [Jones, 2000].

Function points are a good choice for the analysis of many different types of software projects and can provide information for different types of analyses, such as software-reuse analysis, object-oriented economic analysis, and even full-life cycle analysis. This versatility compared to other forms of software sizing makes function point counting a good choice for non-application-specific software measurement, which is desired in the context of this thesis.

- In 1986, Software Productivity Research (SPR) developed an experimental method for applying Function Point logic to system software such as operating systems, telephone switching systems, and the like [Jones, 1986]. The resulting metric was called the SPR Feature Point and is a superset of the IBM Function Point metric [Albrecht, 1981]. It introduces a new parameter- number of algorithms- in addition to the five standard Function Point parameters discussed earlier in this section. The motivation behind Feature Points was to be able to estimate software size for system software characterized by high algorithmic complexity.

The typical ratio of feature points to function points for an embedded real-time application is 1.35 to 1 [Jones, 1998].

Thus, the addition of the algorithm factor augmented the usage of the original functional metrics. Since Feature Points are driven by algorithmic complexity, a definition of “algorithm” is appropriate. An algorithm is defined in standard software engineering texts as the set of rules that must be completely expressed to solve a significant computational problem [Hetzel, 1993]. For Feature Point counting purposes, an algorithm can be defined in the following terms: “An algorithm is a bounded computational problem that is included within a specific computer program” [Jones, 1996]. When determining what algorithms are countable and significant, one must follow these supplemental rules [Jones, 1998]:

- The algorithm must deal with a solvable problem.

- The algorithm must deal with a bounded problem.
- The algorithm must deal with a definite problem.
- The algorithm must be finite and have an end.
- The algorithm must be precise and have no ambiguity.
- The algorithm must have an input or starting value.
- The algorithm must have output or produce a result.
- The algorithm must be implementable in that each step must be capable of execution on a computer.
- The algorithm can include or call upon subordinate algorithms.
- The algorithm must be capable of representation via the standard structured programming concepts of sequence, if-then-else, do-while, CASE, etc.

A.1.2 Definitions *[IFPUG, 1999]*

1. Application **-** This is a software package, such as a word processing, spreadsheet, or checkbook package etc.
2. Application User (simply referred to as "user") **-** A user is someone who needs a software application to perform his or her duties. For example, a user set might include data entry clerks, managers who need certain reports, customers who receive bills, system administrators who need to query the software's databases, et al. A user set does not normally refer to those whose role is software production such as programmers, database designers, or release managers; their role is to develop the software, not to use it after its market implementation.
3. Data Element Type (DET) **-** Usually a DET is a field of data. It can also be an element of control information, such as the “Enter” key when it is needed to

initiate the process of data input into an internal data file. In general, the more DETs in a function type (such as an external input), the higher its function point size. *"A unique, user-recognizable, non-recursive field. The number of DETs is used to determine the complexity of each function type and the function type's contribution to the unadjusted function point count."*

4. External Input (EI) **-** EI is the process of adding, changing, and/or deleting data from an internal database. An example would be entering check numbers and amounts into a checkbook software package. An EI has three, four, or six unadjusted function points depending on whether it is of low, average, or high size/complexity. The textbook definition includes *"... processes data or control information that comes from outside the application's boundary. The external input itself is an elementary process. The processed data maintains one or more internal logical files. The processed control information may or may not maintain an ILF."*
5. External Inquiry (EQ) **-** The process that allows the user to simply read or retrieve existing data from a database using certain criteria, much like an automated card catalog system in a public library. An EQ has three, four, or six unadjusted function points depending on whether it is of low, average, or high size and complexity. The textbook definition includes *"... an elementary process made up of an input-output combination that results in data retrieval. The output side contains no derived data. No ILF is maintained during processing."*
6. External Interface File (EIF) **-** A database maintained in another application, but accessed by the application being counted on a read-only basis. An EIF has five,

seven, or 10 unadjusted function points depending on whether it is of low, average, or high size/complexity. The textbook definition includes *"... a use-identifiable group of logically related data or control referenced by the application, but maintained within the boundary of another application. This means an EIF counted for an application must be an ILF in another application."*

7. External Output (EO) **-** The process that yields a completed report, output file, or any other type of message set, which is sent to users. The report often contains data in fields that require calculations to derive. Examples could include credit card bills, completed spreadsheet reports, or state tax refunds. An EO has four, five, or seven unadjusted function points depending on whether it is of low, average, or high size and complexity. The textbook definition includes *"... is an elementary process that generates data or control information sent outside the application's boundary."*
8. Function Point **-** One standard unit of delivered or finished software size, analogous to a gallon of milk, a case of beer, or a cord of wood. The size of a software package, from the viewpoint of a user, is its number of function points. A function point is unadjusted until it is weighted according to the overall application value adjustment factor. When using the term function point, it is usually understood that it refers to the adjusted or final function point. IFPUG describes it as *"A metric that describes a unit of work product suitable for quantifying application software."*

9. General Systems Characteristics (GSCs) - GSCs are 14 additional factors used to determine size/complexity of software. These will be discussed in greater detail in Section 2.1.3

10. Internal Logical File (ILF) - The ILF is a database that is inside the application. An ILF has seven, 10, or 15 unadjusted function points depending on whether it is of low, average, or high size/complexity. It is also defined as *" a user identifiable group of logically related data or control information maintained within the boundary of the application."*

11. Record Element Type (RET) - An RET is user recognizable sub group of data elements within a Logic File or an Interface file, also defined as *"User recognizable subgroups of data elements within an ILF or EIF"*

12. File Type Referenced (FTR) - Each major logical group of user data or control information maintained entirely within the application boundary. An FTR must also be a Logic File or an Interface file.

A.1.3 Counting Function Points and Sizing System Software

Each of these factors (Outputs, Inputs, Inquires, Logic Files, Interfaces and Algorithms) discussed above can be classified as "Low" complexity, "Average" complexity, "High" complexity depending on the number of Files and Data Elements referenced by each of them and then assigned "weights" corresponding to the complexity. Table A.1 through Table A.5 constitute the guideline for establishing the complexity levels and hence the weights (in brackets) for each of these factors [IFPUG, 1999]. Multiplying the number of instances of each type (low, average, high) of each parameter by the corresponding weight and then adding up the results for all the parameters will give an **Unadjusted Function Point** total.

Table A.1: Complexity values for Inputs

Files Referenced	**Data Elements Referenced**		
	1-4	5-15	Greater than 15
Less than 2	Low (3)	Low (3)	Average (4)
2	Low (3)	Average (4)	High (6)
Greater than 2	Average (4)	High (6)	High (6)

Table A.2: Complexity values for Outputs

Files Referenced	**Data Elements Referenced**		
	1-5	6-19	Greater than 19
Less than 2	Low (4)	Low (4)	Average (5)
2 or 3	Low (4)	Average (5)	High (7)
Greater than 3	Average (5)	High (7)	High (7)

Table A.3: Complexity values for Inquiries

Files Referenced	**Data Elements Referenced**		
	1-5	6-19	Greater than 19
Less than 2	Low (3)	Low (3)	Average (4)
2 or 3	Low (3)	Average (4)	High (6)
Greater than 3	Average (4)	High (6)	High (6)

Table A.4: Complexity values for logic files

Record Element Types (RET)	**Data Elements Referenced**		
	1 to 19	20 - 50	51 or More
1 RET	Low (7)	Low (7)	Average (10)
2 to 5 RET	Low (7)	Average (10)	High (15)
6 or More RET	Average (10)	High (15)	High (15)

Table A.5: Complexity values for logic files

Record Element Types (RET)	**Data Elements Referenced**		
	1 to 19	20 - 50	51 or More
1 RET	Low (5)	Low (5)	Average (7)
2 to 5 RET	Low (5)	Average (7)	High (10)
6 or More RET	Average (7)	High (10)	High (10)

The Unadjusted Function Point (UFP) total is then multiplied by a “value adjustment factor” in order to obtain an Adjusted Function Point Count. This “value adjustment factor (VAF)” is determined using the “total degrees of influence (TDI)” which in turn is calculated on the basis of rating the General System Characteristics (GSC) as described below.

These characteristics are production environment factors that influence the system as a whole and not just a particular function. The 14 GSCs are listed in Table 2.6

Table A.6: General System Characteristics [IFPUG, 1999]

General System Characteristic		**Brief Description**
1.	Data communications	How many communication facilities are there to aid in the transfer or exchange of information with the application or system?
2.	Distributed data processing	How are distributed data and processing functions handled?
3.	Performance	Did the user require response time or throughput?
4.	Heavily used configuration	How heavily used is the current hardware platform where the application will be executed?
5.	Transaction rate	How frequently are transactions executed - daily, weekly, monthly, etc.?
6.	On-Line data entry	What percentage of the information is entered On-Line?
7.	End-user efficiency	Was the application designed for end-user efficiency?
8.	On-Line update	How many Logic Files are updated by On-Line transaction?
9.	Complex processing	Does the application have extensive logical or mathematical processing?
10.	Reusability	Was the application developed to meet one or many user’s needs?
11.	Installation ease	How difficult is conversion and installation?
12.	Operational ease	How effective and/or automated are start-up, back up, and recovery procedures?
13.	Multiple sites	Was the application specifically designed, developed, and supported to be installed at multiple sites for multiple organizations?
14.	Facilitate change	Was the application specifically designed, developed, and supported to facilitate change?

The degrees of influence range on a scale of zero to five, from no influence to strong influence. Each characteristic is assigned the rating based upon detail descriptions provided by the International Function Point Users Group (IFPUG) 4.1 Manual, [IFPUG, 1999]. The ratings are:

0	Not present, or no influence
1	Incidental influence
2	Moderate influence
3	Average influence
4	Significant influence
5	Strong influence throughout

The sum of the scores of the fourteen characteristics in Table 2.6 gives the Total Degree of Influence (TDI).

The value adjustment factor is calculated using the formula

$$VAF = 0.65 + 0.01(TDI)$$

Thus, we have

$$FP = (VAF)(UFP)$$

The final function point (FP) calculation yields a single number that represents the total amount of functionality being delivered.

This function point can then be correlated to SLOC using Table 2.7

Language	Nominal Level	Source Statements Per Function Point		
		Low	Mean	High
Basic assembly	1	200	320	450
Macro assembly	1.5	130	213	300
C	2.5	60	128	170
FORTRAN	3	75	107	160
COBOL	3	65	107	150
PASCAL	3.5	50	91	125
PL/I	4	65	80	95
ADA 83	4.5	60	71	80
C++	6	30	53	125
ADA 95	6.5	28	49	110
Visual Basic	10	20	32	37
SMALLTALK	15	15	21	40
SQL	27	7	12	15

Table 2.7: Ratios of Logical Source Code Statements to Function Points for Selected Programming Languages Using Version 4.1 of the IFPUG Rules [Jones, 2001].

Appendix B - MOCA / PRICE S Inputs Pertinent to Software Redesign Analysis [Singh, 2001]

MOCA inputs are divided into 5 categories, which are:

- The part inputs characterize parts in the global parts database of MOCA. All the parts used in the system defined in MOCA are linked to this database and therefore any changes in this database of inputs would reflect everywhere the part(s) are present in the system.
- The board inputs characterize the sub-system level assembly of the system. A board is a sub-system that contains multiple parts. All parts that belong to the system must belong to at least one board. The system parts list is obtained by accumulating the part list from each board.
- The system inputs characterize the overall system on which MOCA runs its analysis. All the entities in this data affect the whole system, e.g., reorder would be at a system level requiring availability of all the piece parts etc.
- The solution control inputs characterize the options that may be set to study additions and variations in the analysis, e.g., "look ahead time" may be set here etc.

Part Inputs

1. Part Number – A unique entry in the global database to identify between any two distinct parts. This number could be the manufacturer part number or a company specific part number as long as they are unique. The link that binds parts used in

the system/board to the global database is the part number and therefore any ambiguity in its declaration is problematic.

2. Part Cost – Cost of a single instance of the part in dollars.
3. Obsolescence Date – The predicted date of part obsolescence. In this thesis the data for obsolescence is obtained from either TACTech [TACTech, 2000] analysis or from the CALCE Obsolescence Model [Solomon, 1999] (which uses trends in part sales data
4. Obsolescence Management Strategy –This input determines what is done on a part-specific basis at an obsolescence event. Even though re-design is handled separately by MOCA, it is also considered an obsolescence management strategy.
5. Replacement cost – The cost of replacing a part in a system whenever it needs to be replaced. This input is independent of the part cost, and is a measure of difficulty, complexity and time requirement to remove the part from the system and replace it with a new one.
6. Part Category – Every part belongs to a certain part category. This input indicates the type of part being used. Several part categories are available in MOCA. These are: Microcircuit, Diode, Transistor, Integrated circuit, Semiconductor, Assorted and Custom Defined. The "Assorted" part type is used to represent an aggregate of parts and their instances. All the parts, which do not have any obsolescence or maintenance issues (e.g., most passives and mechanical devices), are lumped together into a single part to reduce computation time. A "Custom Defined" part is a part type for which no single standard part type could be used. When a new part is synthesized as a result of a design refresh, the obsolescence date is reset

based on a default and a lifetime is obtained based on the part category of the modified part.

7. Cost Multiplier for Redesign – The cost multiplier is used to determine the cost of a part after a replacement of the part, i.e., if part “A” is replaced by a part “B” then the new part “B” will have a cost equal to the cost of part “A” multiplied by this redesign cost factor.
8. Block Participation - This indicates which functional block(s) the part participates in. In other words, it indicates which functional blocks need the hardware part in question to implement their functionality.
9. Block Role - As mentioned in item 8 above, “block participation”, indicates which functional block a hardware part participates in. The input “block role” contains information on the significance of the hardware part in that functional block, relative to the other hardware parts in it. MOCA allows four possible choices for “block role”. These are High, Average, Low and None signifying, as the names would suggest, the relative contribution of a hardware part in implementing the functionality of the functional block in question. Assigning a value to the input “block role” would draw on the system designer’s experience to judge the relative significance of each hardware part vis-à-vis every functional block that it participates in.

Board Inputs

1. Total Cost – Cumulative cost of the board including part costs, assembly cost, and any other integration cost. This field stores the user input for the initial cost of the board. The initial cost is the base cost of the board at the system field start. The

cost of the board may change at each obsolescence event due to change in the constituent part costs.

2. Disassembly Cost – Cost incurred to disassemble the board from the system for repair or replacement. This is obviously dependent upon placement of the board in the system and ease of access to it.
3. Assembly Cost – Cost incurred to reassemble the board after it is repaired or when it is being replaced back into the system. This cost if dependent upon placement of the board in the system.
4. Test Cost – Cost incurred to functionally test the board. Functional testing is performed on every board when it is manufactured and when it is repaired.
5. Number of Lumped Parts (also called “Assorted”) – Number of unique parts lumped to make a “Lumped part” for this particular board.
6. Number of Lumped Components – This is an extension of the above number. The difference being that it counts the total number of instances of parts lumped rather than only the number of unique parts.

System Inputs

1. Original Quantity – The quantity of units manufactured for the initial order.
2. Field Start Date – Expected date when the first system will be deployed in the field.
3. End of Product Support Date – Expected date when the system will be no longer be supportable by the manufacturer.
4. System events:

a. Reorder – A reorder date and expected quantity of systems reordered with uncertainties is required for this event.

b. Redesign – A date of expected redesign with uncertainty is required for this event. This is a “fixed” redesign (as opposed to a variable redesign that the MOCA software determines).

Solution Control Inputs

1. Log File (On/Off) – This option is provided to keep track of important analysis steps taking place while the design refresh optimization and life cycle cost analysis takes place.
2. Combine Reorders (On/Off) – This option is provided to enable combining of reorders within a specified time span from the start date specified in the system setup window.
3. Log File Name – This field is provided to change the log file name. If the analysis is a simple life cycle cost estimation, then the tool saves the log in name_rlc.log and if design optimization is running then the tool saves the log in name_rdo.log.
4. Output File Name – This field is provided to change the output file name. Same rules as the log file (above) are followed to name the output file.
5. System with (up-to/exactly, number of moving redesigns) – This field specifies the number of moving re-designs (as opposed to “fixed” redesigns) to be used for design refresh optimization analysis. If exactly option is chosen then MOCA uses only the specified number of moving re-designs for design refresh optimization algorithm. If up-to option is chosen then all the numbers of re-designs possible

up-to the number specified are used. Plotting of previous results follows from the options chosen here.

System Setup Inputs

1. Qualification at (System/Board) – This option is provided to facilitate re-qualification at either of the two levels: i) board, or ii) system. Depending on the option chosen, the interface adjusts and provides fields for necessary inputs in appropriate places, i.e., for board level re-qualification the inputs for re-qualification cost and other particulars should be board specific.
2. Set Part Category Lifetime – The button is provided to set the average mean lifetime for the part type categories. Any change made in this field it is automatically reflected in the complete system/global database.
3. Set Price Compatibility Options – This button is provided to open a window where the various Price Systems Inc. compatibility options can be specified. Some of the options are:
    a. Use Price Systems calculations for design refresh calculations or use MOCA three-tiered model for design refresh calculations.
    b. Use start date of design refresh and end dates for production of first prototype.
    c. Use the end of last prototype and start of production.
4. Economic Inflation Rate – This field is provided to specify the yearly inflation rate (average) to calculate the real value of the money in terms of the base year chosen. Currently the base year is a constant set to year 2000.

5. Look-ahead time – At each design refresh activity MOCA looks ahead and addresses obsolescence related issues for all the parts expected to go obsolete within a user-specified range of time from the date of the design refresh activity in question. This range is called *“Design refresh look-ahead time”*
6. Synthesis Obsolescence Index – This field is provided to specify the new assumed TACTech obsolescence index for the synthesized part when it is being replaced or redesigned at the design refresh activity. This index can vary between 1 and 5 (inclusive). A default of 2.0 is used.
7. Synthesis Obsolescence Confidence – This field is provided to specify the confidence in the value of TACTech risk indices.
8. Inventory – This field is not used in MOCA at this time, however at a later stage MOCA may consider the inventory carried over by the spares or manufactured systems.
9. Chip to Discrete Area Ratio – This field is provided to specify the approximate chip (IC) to discrete part (e.g., transistor, diode, etc.) area ratio. This is necessary to calculate the area units of the board required by the Price Systems tool.
10. Time Between Design Start and First Prototype – This field is provided to specify the time between the start date field and the first prototype date field in the Price H model. This field is only used if the Price compatibility option for re-design calculations in enabled.
11. Time Between First Prototype and Last Prototype - This field is used to specify the time between the first prototype date field and the last prototype date field in Price System H model. This field is only used if Price compatibility option for re-

design calculations in enabled. This field and the field explained above are used collectively to force the increase in re-design cost for sensitivity analysis.

12. Combine Reorders (yrs) – This field is provided to set the number of years (time span) to be used to combine reorders in the combine reorder algorithm. Combine Reorders Start Date – This field is provided to set the date from which the combine reorders algorithm starts combining reorders. Usually this coincides with the field start date field.

PRICE S Inputs

11. Application (APPL): a measure of the type (or types) of software, described by one of seven categories (mathematical, string manipulation, data storage and retrieval, on-line, real-time, interactive, or operating system).
12. Productivity Factor (PROFAC): A calibrated parameter, which relates the software program to the productivity, efficiency/inefficiencies, software development practices and management practices of the development organization.
13. Complexities (CPLXM, CPLX1, CPLX2): Three complexity parameters which relate the project to the expected completion time, based on organizational experience, personnel, development tools, hardware characteristics, and other complicating factors.
14. Platform (PLTFM): the operating environment, in terms of specification, structure and reliability requirements.

15. Utilization (UTIL): Percentage of hardware memory or processing speed utilized by the software.
16. New Design/New Code (NEWD/NEWC): Percentage of new design and new code.
17. Integration (Internal) (INTEGI): Effort to integrate various software components together to form an integrated and tested software module.
18. Integration (External) (INTEGE): Effort to integrate various software modules together to form an integrated and tested software system.
19. Schedule (DSTART/DEND): Software project start and/or end dates.
20. Optional Input Parameters: Financial factors, escalation, risk simulation.

BIBLIOGRAPHY

Albrecht, A., “Measuring Application Development Productivity,” Programming Productivity: Issues for the Eighties, IEEE Computer Society Press, Washington DC, 1981.

Baca, M., “TACTech Electronic Component Obsolescence Management”, Presentation to Boeing Electronic Component Management Users’ Forum, March 4, 1997.

Beland, S.C., and BonJour, B., “Functional Failure Path Analysis of Airborne Electronic Hardware”, DASC, Philadelphia, PA, Oct. 2000.

Boehm, B., Clark, B.K., Horowitz, E., Madachy, R., Selby, R.W., and Westland, C., “Cost Models for Future Software Processes: COCOMO 2.0,” Annals of Software Engineering, 1995.

Boehm, B., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Condra L. W., “Combating Electronic Component Obsolescence by Using Common Processes for Defense and Commercial Aerospace Electronics”, IECQ-CMC Avionics Working Group1, NDIA Paper document, September 1999.

Condra L. W., Presentation to Boeing Commercial Airplane Group Electronic Component Management Program Users' Forum II, March 1997.

DeBardelaben, J., “An Optimization-Based Approach for Cost-Effective Embedded DSP System Design”, Ph.D. dissertation, Georgia Institute of Technology, May 1998.

Dehlinger, J., and Lutz, R., "Software Fault Tree Analysis for Product Lines," 8th IEEE International Symposium on High Assurance Systems Engineering, Tampa, FL, 2004

Dreger, J. B., Function Point Analysis, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.

Hart, K., and Mitchell, T., “Aging Aircraft" Military Aerospace Technology, Apr 18, 2003.

Hetzel, B., Making Software Measurement Work, QED Publishing Group, Boston, 1993.

IFPUG, International Function Point Users Group: “Function Point Counting Practices Manual,” Release 4.1, 1999

JACG, Joint Aeronautical Commanders' Group: Flexible Sustainment Guide, Change 2, July 1999.

Jones, T.C., Applied Software Measurement, McGraw-Hill, Inc., New York, 1996.

Jones, T.C., Estimating Software Costs, McGraw-Hill, Inc., New York, 1998.

Jones, T.C., Programming Productivity, McGraw-Hill, Inc., New York, 1986.

Jones, T.C., Table of Programming Languages and Levels – Version 8.2; Software Productivity Research, Burlington, MA, 2001.

Leveson, N. G., "Safeware: System Safety and Computers." Addison-Wesley, Reading, MA, USA, 1995.

McArthur, C. J., and Snyder, H. M., "Life Cycle Cost – The Logistics Support Analysis Connection", Proceedings of the IEEE National Aerospace and Electronics Conference NAECON, Vol. 3, pp. 1206-1209, May 1989

MOCA user's guide (http://www.calce.umd.edu/contracts/MOCA/MOCA_Page.htm), June 2003

Pai, G.J., and Dugan, J.B., "Enhancing Software Reliability Estimation using Bayesian Networks and Fault Trees," Proceedings of the IEEE International Symposium on Software Reliability Engineering, Nov. 2001

Parametric Estimating Initiative (PEI) Parametric Estimating Handbook (http://www.ispa-cost.org/PEIWeb/newbook.htm), March 2004

Park, R. E., "The Central Equation of the PRICE Software Cost Model", 4th COCOMO User's Group meeting, Nov 1988

Sandborn, P.A., and Singh, P., "Electronic Part Obsolescence Driven Design Refresh Optimization," Proc. FAA/DoD/NASA Aging Aircraft Conference, San Francisco, CA, September 2002.

Singh, P., "Design refresh planning optimization driven by electronic part obsolescence", M.S. Thesis, Department of Mechanical Engineering, University of Maryland College Park, December 2001

Solomon R. "Life Cycle Mismatch Assessment and Obsolescence Management of Electronic Components," Ph.D. Dissertation, University of Maryland, College Park, MD, 1999

TACTech, data collected from TACTech in collaboration with Honeywell Inc., 2000

Wong, K., "On Inserting Program Understanding Technology into the Software Change Process", 4th International Workshop on Program Comprehension, March 1996

Wright M. B., Humphrey D., and McCluskey F. P., “Uprating Electronic Components for Use Outside their Temperature Specification Limits,” IEEE Transactions on Components, Packaging, and Manufacturing Technology, Part A, Vol. 20, No. 2, pp. 252-256, June 1997.