Executable Behavioral Modeling of System and Software Architecture Specifications to Inform Resourcing Decisions

Farah-Stapleton, Monica; Auguston, Mikhail; Giammarco, Kristin

Elsevier

Farah-Stapleton, Monica, Mikhail Auguston, and Kristin Giammarco. "Executable Behavioral Modeling of System and Software Architecture Specifications to Inform Resourcing Decisions." Procedia Computer Science 95 (2016): 48-57.
http://hdl.handle.net/10945/59372

Downloaded from NPS Archive: Calhoun
Abstract
The size, cost, and slow rate of change of DoD Information Technology (IT) systems in comparison with commercial IT makes introduction of a new DoD system or capability challenging. Making design decisions without consideration of the whole system and its environment may result in unintended behaviors that have operational and financial impacts, often not visible until later testing. The complexity of these system interactions isn’t cheap, impacting intellectual, programmatic, and organizational resources. Precise behavioral modeling offers a way to assess architectural design decisions prior to, during, and after implementation to mitigate the impacts of complexity, but in and of itself does not lead to estimates of the effort and the cost of those design decisions. This research introduces a methodology to extract Unadjusted Function Point (UFP) counts from architectural behavioral models utilizing a framework called Monterey Phoenix (MP), lightweight formal methods, and high level pseudocode for use in cost estimation models such as COCOMO II. Additionally, integration test estimates are informed by extracts of MP model event traces. These unambiguous, executable architecture models and their views can be inspected and revised, in order to facilitate communication with stakeholders, reduce the potential for software failure, and lower costs in implementation.

Keywords: architecture; lightweight formal methods; behaviors; event traces; function point analysis; COCOMO II, complexity.
1. Introduction

This paper presents a domain independent methodology, hereafter referred to as ThreeMetrics to extract Unadjusted Function Point (UFP) counts from discrete architectural behavioral models, created from the Monterey Phoenix (MP)\(^1\) modeling language and framework, for use in cost estimation models such as COnstructive COSt MOdel II (COCOMO\(^{®}\) II)\(^2,3\). The MP model itself is a rich source of additional information including scenarios (use cases) that can be extracted from the MP model to inform distinct integration test case development, as well as views of instances of the architecture model that can be inspected for accuracy and facilitate communication with stakeholders. The UFP count, event traces, and views were the inspiration for the name ThreeMetrics.

1.1. Background

Historically, there have been significant but often disconnected efforts to develop architectural descriptions of new and legacy systems. Architectural design and analysis are powerful mechanisms that allow the capture of design decisions early in the design process, so that it can be assessed and modified without incurring unnecessary costs of incorrect implementations. Unfortunately, architectural design decisions are often captured on a system by system basis, using a spectrum of representations from natural language to formal notations. These inconsistent systems architectures are then analyzed through manually intensive methods such as inspections and reviews. System and software architecture and development efforts are often unrelated, incomplete, or duplicative, with a technically and programmatically unsustainable result. This is an unfortunate state of affairs because architectures matter. “Every system has an architecture, whether or not it is documented and understood\(^4\).” Not only is the architecture of a software system complex, but so are the programmatic, organizational, and resourcing constructs that interact with each other throughout the software lifecycle. All these architectures deserve the attention of technical and programmatic decision makers because if constructed properly, they can not only capture design decisions but also inform resourcing decisions and reduce the complexity of sociotechnical implementation. The ThreeMetrics methodology applies elements of the Function Point counting process to MP architecture models, in order to extract an Unadjusted Function Point count from MP models, and inform technical and programmatic decision making.

1.2. Monterey Phoenix (MP)

MP is a behavioral model for system and software architecture specification based on event traces, and supports several architecture composition operations and views. As an executable architecture model, it can be used to automatically generate examples of the behaviors (e.g. use cases) for early system architecture testing. This software and system modeling framework can also be used to capture design decisions such as precedence, inclusion, concurrency, and ordering (dependency relation between activities).\(^5,6,7\) MP’s foundation is in lightweight formal methods, which plays a key role in assessing the complex behaviors of a software intensive system, and in the development of formal specifications for the system and the environment. Formal methods are essential to behavioral modeling of complex systems, because they remove ambiguity from architectural modeling. As with all assessments, lightweight formal methods based architectural assessments are assisted by visual representations and automated tools. Such tools provide immediate feedback, assist in identifying errors once an early architecture draft is constructed, and allow the user to reason about the model. There are many tools supporting lightweight formal methods based analysis, including the MP Analyzer on Firebird\(^8\), Eagle6\(^9\), and Alloy Analyzer\(^10\). Firebird and Eagle6 are implementations of the MP Framework. Eagle6 is a commercial tool, which has been graciously made available for select research purposes. Firebird is an NPS implementation that is publicly available, and was ultimately selected for this work.

1.3. Function Point Counting

The ThreeMetrics methodology leverages the International Function Point User Group (IFPUG) Function Point counting method defined in Function Point Counting Practices Manual Release 4.3.1. “A Function Point is a normalized metric used to evaluate software deliverables and to measure size based on well-defined functional characteristics of the software system.”\(^11\) The unit of functional size for this method is called a Function Point (FP).
Function Point counting is intended to provide a way to measure software development and maintenance projects, independent of the technology used for implementation. It is viewed from the perspective of the functionality requested by and provided to the User, either a human or another application, with the expectation that the software can be quantified and measured consistently across the enterprise. FP descriptions can also be considered as ways to view a system, its sub-components, and the environment through interactions and behaviors, in order to address the concerns of specific stakeholders. As such, FP Analysis (FPA) is an initial step to describing the architecture of a system.

One of the earliest activities in the FPA process is identifying the counting scope and application boundaries. FP transactional functions, i.e. interactions, can be viewed as markers of these boundaries. Transactional functions include External Input (EI), external Output (EO), and External Inquiry (EQ). EIs are input data that is entering the system (application being counted), maintaining (i.e. adding, changing, deleting) an Internal Logical File (ILF) and/or altering the behavior of the system. An EO is derived data or algorithms leaving the system (application being counted), sending data or control information outside the application’s boundary. The primary intent of an EO is to present information to a User through processing logic other than or in addition to the retrieval of data or control information. The processing logic must contain at least one mathematical formula or calculation, create derived data, maintain one or more ILFs, and/or alter the behavior of the system. An EQ retrieves data only, and presents information to a user through the retrieval of data or control information. Data functions associated with FPA can be represented as interactions, and also viewed as markers of internal and external boundaries. An Internal Logical File (ILF) addresses data that is processed and stored (maintained) within the application boundary. The External Interface File (EIF) addresses data that is maintained by applications outside the application boundary, but are necessary to satisfy a particular process requirement. ThreeMetrics methodology employing MP assists in unambiguously identifying the boundaries and interactions of the application, the data, and environment (anything not the application being counted), through descriptions in the MP model schema.

2. ThreeMetrics Methodology

The ThreeMetrics methodology is illustrated in Figure 1 and described in the following 9 steps.

Step 1: Determine Stakeholder Questions To Be Answered and Gather Existing Documentation. The first step is to understand why the model is being developed and what existing documentation is available to assist in understanding the software system and the environment (everything not the system) with which it interacts. Practitioners of the FP counting methodology recommend using any documentation or architectural artifacts that may be available when performing a functional size measurement, such as: Requirements document(s); Logical data models; Data flow diagrams; Use Cases; and anything else that provides insights into what the application is intended to do. This is consistent with system and software architecture and engineering approaches.

Stakeholders should have complementary interests, but due to incomplete or insufficient architectural representations, their interests are often in conflict. The practical requirements of multi-stakeholder challenges can be satisfied by early and consistent behavioral modeling and the extraction of statistics from those executable architecture models, to inform high level design and cost. Typical stakeholder questions include:

- Are user, technical, cost, and management expectations being met?
- Does this system do what the users expected? Does it fulfill prioritized requirements?
- What implementation option(s) should be considered to meet user performance expectations?
- What are optimal instrumentation points? What statistics should be gathered? What is the correct level of architectural abstraction?
- What is the cost of the system from requirements elicitation thru software evolution?

These questions can be informed by an architecture model, serving as a bridge between requirements and high level design.
Step 2: Identify scope and application boundary. Step 2 utilizes the information from Step 1 to determine the scope of the count, i.e. is it a Development Project function point count, an Enhancement Project function point count, or an Application function point count. Most importantly, Step 2 utilizes this information to identify the boundary of the application to be counted, a critical step in any software or system engineering analysis, when trying to distinguish the system under analysis and the environment with which it interacts. This boundary is a conceptual interface between the software application and its users\textsuperscript{11}, and includes the ILF(s) maintained by the application. The simple box and arrow type of architectural representation of Figure 2 can usually be recovered from disparate

---

\textsuperscript{11} The exact definition and implications of the boundary can vary depending on the methodology and context of the analysis. In the context of the ThreeMetrics Methodology, the boundary is defined as the interface between the software application and its users, including the ILF(s) maintained by the application.
or incomplete system artifacts. The red dotted line represents the boundary of the application being counted. Since
the natural language of the functional requirements may still be ambiguous, the box and arrow representation assists
in confirming: The boundary of the application to be counted; All known data functions (ILF, EIF); All transactional
functions (EI, EO, EQ); and The User (human or another application). The box and arrow view also illustrates the
decomposition of the application to be counted into the ILF(s) and an Internal Abstracted Application (IAA) which
represents everything except the ILF(s). The IAA was needed to represent the internal interactions with the ILF and
the external interactions with the EIF, in order to capture the UFP count for data functions. These interactions
represent operations on the data. Without the IAA, the interactions between the User and the ILFs and EIFs would
have accounted for the contributions of the transactional functions to the UFP, but not those of the data functions.

While this high level architecture view does not adequately represent the software system behaviors needed to
extract the UFP count, it is a practitioner’s tool to set the conditions to develop an MP model, and can be expanded to
assist in assigning organizational responsibility in a complex system of systems (SoS). This view also resonates with
non-technical stakeholders, but can quickly become unwieldy, containing so much information that it defeats its
original purpose of simplification. Representing the model in MP is much more efficient and as an executable
architecture model, can benefit from the use of automated tools.

Step 3: Develop MP Model. Leveraging techniques from FP counting, functional requirements are assessed in
order to shape what the application is intended to do, from the perspective of the User. The User, the ILFs, the EIFs
and the IAA are represented as Actors (ROOTs). The interactions between the User, ILFs, EIFs and IAA, i.e. the
transactional functions EQ, EO, EI, are specified through MP composition operations COORDINATE and SHARE
ALL. When the PRECEDES relationship is added, the MP COORDINATE effectively says “do something, and then
something else happens” in pseudocode. SHARE ALL is the simplified expression of interaction, when who initiates
the interaction and who is the recipient is not relevant, or insufficient detail about the interaction is available.
COORDINATE requires two events and is used to represent the interactions associated with transactional functions.
SHARE ALL requires a single shared event and is used to represent interactions between the IAA and the ILFs and
EIFs. If sufficient information about the data functions is available, then it would be possible to use a nested
COORDINATE to obtain a more precise UFP count for the data functions.

UFPs represent interaction abstractions and can be extracted from COORDINATE and SHARE ALL MP
constructs. “Hidden” within the COORDINATE are FP descriptions that contribute to functional size, such as Data
Element Types (DETs). File Types Referenced (FTR), and Record Element Types (RETs) which also contribute to
functional size are represented in the schema, based on source information available. The number of DETs and FTRs
associated with a transactional function determine the functional complexity rating and the functional size (i.e. UFP
count) of a transactional function. The number of DETs and RETs associated with a data function determine the
functional complexity rating and the functional size (i.e. UFP count) of a data function.

COORDINATE is used to represent the high level interactions (EI, EO, EQ) of the transactional functions, and
then nested interactions (nested COORDINATE) to represent the DETs that determine the functional complexity
rating. This functional complexity rating corresponds to an UFP size in the IFPUG tables. Those numbers are then
used as a “weight” associated with the COORDINATE, resulting in the same number of UFPs a traditional UFP count
would produce for that transactional function. The overall initial UFP count extracted from the MP model and the
traditional UFP count are very close, if not identical. For counts performed on applications whose functional
requirements are still maturing, an initial complexity rating of Average can be used, and then refined.

The MP Schema in Figure 3 is derived from the It’s TeeTime\textsuperscript{16} FP counting example, graciously provided by Q/P
Management Group Inc., for this research. It’s TeeTime (hereafter referred to as Tee Time) is a golfing application
that includes natural language descriptions of functional requirements, prototyped display screens, detailed
descriptions of ILFs, EIFs, and their content, and descriptions of User and application interactions that result in EIs,
EQs, and EOs. For an EQ named Display State Drop Down, the User interacts with the application and queries for
information that is resident in the ILF named Golf Courses (GC_ILF). The specific behaviors include “click on state
arrow” and “state list display returned”, meaning the User clicks on the arrow of a drop down menu, and a list of states is then displayed back to the user, on the screen. From the source data, one FTR (GC_ILF), and 2 DETs (arrow click, state field) are identified using the FP counting practice. The naming convention of the ROOTs is to assist with managing the complexity of the descriptions of ROOT behaviors and the interactions between the ROOTs. The structure of events visible in an MP model provides the source for assigning weights, with some interactions being “heavier”, i.e. more complex, than others. The nested COORDINATE will have composite events, and the number of composite events will affect the weight, and ultimately the UFP count.

For this example, the IAA is named TT. For transactional functions, ROOT TT_GC_ILF represents the abstracted combination of the GC_ILF and the IAA TT, both of which are internal to the Tee Time application boundary. The User interacts with the abstracted combination ROOT TT_GC_ILF in order to inquire on data that is in the GC_ILF. This sample schema specifically represents the behaviors of one EQ, State Drop Down, which describes the interactions between actors ROOT User and ROOT TT_GC_ILF in lines 02-05. The MP composition operation COORDINATE is represented in lines 06-17. The DETs are represented in Lines 09-16, nested in the COORDINATE of lines 06-17.

Fig. 3 MP schema for EQ_State_Drop Down transactional function

Figure 4 continues with the MP schema extracted from the TeeTime example, focusing on one data function, GC_ILF. SHARE ALL is used for the internal interactions between the IAA ROOT TT and the Golf Courses ILF ROOT GC_ILF.

Fig. 4 MP schema for GC_ILF data function

Since an MP representation is an executable model, it should be iteratively tested and debugged before extracting information from it.
Step 4: Extract Data Functions count from MP model. The number of SHARE ALLs can be extracted from the MP schema, in this case there is one SHARE ALL in Fig. 4. The SHARE ALL corresponds to counting one data function, in this case for GC_ILF. Once the number of SHARE ALLs are extracted from the schema, the IFPUG tables provide the functional complexity rating and corresponding UFP size that are then applied as weights.

Step 5: Extract Transactional Function count from MP model. Similarly, by inspecting the schema for the EQ State Drop Down transactional function in Fig. 3, one COORDINATE is counted with the nested operations of ADDs, each ADD representing one DET for a total of 2 DETs. The IFPUG tables provide the functional complexity rating and corresponding UFP size that are then applied as weights.

Step 6: Extract integration test cases and views from MP model. MP is identified as an executable architecture model because event traces are generated from the event grammar rules and then adjusted and filtered according to the composition operations (COORDINATE and SHARE ALL) in the schema.

As an executable architecture model, MP provides a rich source of information that informs effort. Additionally, extracting event traces from specifications of behaviors and interactions not only sets the conditions for early system architecture testing and verification with tools, but also for using the resulting scenarios and use cases to support test case construction. As mentioned in Step 3, an MP representation is an executable model, and should be tested and debugged before extracting information from it. Once the model is considered correct, then there is a greater degree of confidence that all scenarios and use cases generated by the model are also correct. These scenarios and use cases can then each inform a test case, for implementation.

Fig.5 MP event trace

MP can be used to automatically generate event traces, which represent examples of behaviors (e.g. scenarios, or use cases if the environment is included). Recall that an event trace represents an example of a particular execution of the system that is derived from the architecture specified by a schema. In the case of MP models, it is possible to automatically generate all event traces within a given limit, i.e. scope. Jackson introduced the Small Scope Hypothesis, which states that “Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference”, effectively, most errors can be demonstrated on relatively small counterexamples. Auguston stated that “such a limit (scope) may be set by the maximum total number of events within the trace, or by the upper limit on the number of iterations in grammar rules.” Firebird MP Analyzer is an implementation of the MP event trace generator that utilizes the Small Scope Hypothesis. The automatically generated event traces are made small by simulating only a small number of iterations on all loops, typically up to 3 (i.e. Scope 3). For some MP models, Scope 1 is sufficient because
increasing the scope will result in a large number of event traces that may not show anything new or notable, and will not improve chances of exposing errors in testing. The executable model may take too long to run, resulting in a poor return on investment of time and effort. Auguston also states that “in the case of MP models it is possible to automatically generate all event traces within the given scope (exhaustive testing).” The process of generating and inspecting event traces from an MP schema is similar to the traditional software testing process of generating and inspecting test cases. It is easier to evaluate an example of behavior (a particular event trace) than the generic description of all behaviors (the schema). Tools such as MP Analyzer assist in these evaluations by generating an exhaustive set of event traces for a scope that can then be inspected and used to inform integration test cases. As an example, the MP schema for EQ State Drop Down was executed using MP Analyzer on Firebird, with an event trace illustrated in Fig. 5 that was extracted from the model. For the complete Tee Time example, assuming an Average functional complexity, 864 event traces (number of potential test cases informed) were generated using a Scope of 1.

TutorialsPoint synopsizes the definition of a software test case as “A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.” A test case includes test steps, preconditions, test data that supports what the test case needs to achieve, expected results, post conditions, and information about the environment. An integration test case addresses the interface and data flow between modules or systems, focusing on what happen at the boundary. The creation of integration test cases takes effort. The event traces generated from an MP model provide solid detailed blueprints, which can be viewed as guidelines for the creation of the integration test cases. The event trace in Fig. 5 illustrates the behaviors of User and TT_GF_ILF and the interactions between them, which can be used to identify the steps in a test case. For this event trace, the User input results in two events: The User click’s the arrow for state drop down (Click_state_arrow_dropdown); and User should receive state list display (Receive_state_list_display). Receive state list display is a description of the expected system’s output. Brooks states that he has “successfully used the following rule of thumb for scheduling a software task: 1/3 planning, 1/6 coding, 1/4 component test and early system test, 1/4 system test, all components in hand.”

\[ .25 \times \text{Total effort} = \text{Estimate for integration testing} \]

As discussed by Wolff, approximately six integration tests per day can be executed for a large application, such as an electronic commerce system. This doesn’t include the amount of time required to create the test case.

\[ (\text{Integration test case creation} + \text{execution}) = \text{Estimate for integration testing} \]

The amount of time for integration test case construction varies by the complexity of the interface being tested and the identification of test data. The time to create integration test cases ranges from several hours for a simple test case to several days for a more complicated one. Estimates are not only numbers; they provide useful information for informed decision making regarding the planning, implementation overall and management of a real software project. If 500 integration test cases are needed to ensure all behaviors of a system are covered, but an organization is resource constrained or schedule constrained and can only execute 50 integration test cases, which integration test cases should be selected? The process and criteria for selecting a subset of test cases is a topic for next steps.

Step 7: Determine the Unadjusted Function Point (UFP) count. For the transactional function EQ State Drop Down, 1 COORDINATE and 2 ADDs are identified through manual inspection of Fig. 3. TeeTime source information indicates that for EQ State Drop Down there is 1 FTR. Based on the IFPUG tables, 1 FTR and 2 DETs correspond to a functional complexity rating of Low. For an EQ, Low corresponds to a functional size of 3 UFPs. Therefore, EQ State Drop Down = (1 COORDINATE) \* 3 UFP/COORDINATE = 3 UFPs. For the GC_ILF data function in Fig. 4, the TeeTime source information indicated that there is one RET and 1-19 DETs, so based on the IFPUG tables, the functional complexity rating is Low. For an ILF, Low corresponds to a functional size of 7 UFPs. Often, information such as the precise number of RETs and DETs, or FTRs and DETs is not known, so the interactions between the ROOTs can initially be assigned a functional complexity rating of Average from the IFPUG functional complexity and size tables, until the requirements are refined.
Step 8: Calculate effort estimate. The COCOMO II capability for estimation assesses effort by utilizing UFP counts or software lines of code. UFP counts can be transformed into lines of code based on the software implementation language used. These calculations can be done manually leveraging equations\textsuperscript{2,3}, or by using automated tools that relate the MP model to the COCOMO II, such as http://csse.usc.edu/tools/MP_COCOMO\textsuperscript{21}. This tool has been extended by Dr. Ray Madachy, to not only accept an UFP direct input, but to also extract an UFP count from an uploaded .mp file. For the complete TeeTime example, 88 UFPs were counted. That UFP count is input in the COCOMO II model illustrated in Figure 6. The output of the model, for nominal inputs, Java language, and a cost per person-month of $20,000/month resulted in 16 Person-months of effort, 9.2 months for schedule, and cost of $319,750.

![COCOMO II model input](image)

Step 9: Finalize analysis and provide results to stakeholders. A software engineer may expect behaviors to be represented by UML sequence diagrams, Agile user stories, high level pseudo code, or implemented code. A system of systems engineer may want to search for the conditions that result in emergent behavior. Cost analysts will review the resourcing implications for each instance of architecture, independent of the spectrum of estimation strategies from Excel through parametric models addressing individual system costs. The user just wants the system to work from his perspective, independent of the healthy tensions between cost and design\textsuperscript{22}. Each step in this methodology results in a view or set of consistent views that can be used to inform multiple stakeholders.

3. Summary and follow-on work

The ThreeMetrics methodology addresses the relationship of Function Point Analysis, COCOMO II cost modeling, and executable behavioral modeling of system and software architecture specifications leveraging lightweight formal methods and pseudocode. The methodology extracts UFP counts from MP behavioral models for use in COCOMO II. The results can be visualized in multiple views, in order to communicate with a spectrum of stakeholders. Follow-on work includes: Identifying how to select the most relevant integration test cases from the blueprint provided by MP model scenarios, if resourcing limits the number of test cases that can be created; Exploring the range of effort estimates for a given number of UFPs, language, and range of input options; and Identifying a precise convention for the translation of natural language into MP pseudo code to represent FP transactional and data functions.
References

1. M. Auguston and C. Whitcomb, “Behavior Models and Composition for Software and Systems Architecture”, Proceedings of the 24th ICSSEA Conference (International Conference on Software and Systems Engineering and their Applications), Paris, France, October 23-25 2012.
2. COCOMO II Model Definition Manual, Version 2000.3, © 1995 – 2000 Center for Software Engineering, USC http://csse.usc.edu/csse/affiliate/private/COCOMOII_2000_3/COCOMO_II_2000_3.html
3. USC Center for Systems and Software Engineering (UCSSE) Website http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html
4. N. Rozanski, E. Woods, Software Systems Architecture, Addison-Wesley, 2012
5. M. Auguston, C. Whitcomb, “System Architecture Specification Based on Behavior Models”, in the Proceedings of the 15th International Command and Control Research and Technology Symposium (ICCRTS), June 22–24, 2010, in Santa Monica, CA (USA). http://dodccrp.org/html4/events_15.html
6. M. Auguston, Monterey Phoenix System and Software Architecture Modeling Language v.4, May 20, 2012, Naval Postgraduate School
7. M. Auguston, Software Architecture Built from Behavior Models, ACM SIGSOFT Volume 34, Number 5, September 2009, pp. 1-15
8. Firebird Analyzer, http://firebird.nps.edu/
9. Eagle6, http://www.eagle6.com/index.php
10. Alloy Analyzer 4.1.10 MIT, Accessed May 8, 2009 http://alloy.mit.edu/community/software
11. International Function Point User Group (IFPUG), Function Point Counting Practices Manual, Release 4.3.1 © 2010 IFPUG. ISBN 978-0-9753783-4-2
12. D. Garmus, J. Russac, R. Edwards, Certified Function Point Specialist Examination Guide, CRC Press, 2010
13. D. Longstreet, Function Points Analysis Training Course, David@SoftwareMetrics.Com, www.SoftwareMetrics.Com
14. N. Fenton, J. Bieman, Software Metrics, 3rd Edition, CRC, 2015, pp.252-254
15. Ruben. G. Mathew, Progressive Function Point Analysis, Advanced Estimation Techniques for IT Projects, 2014, rubengerad@gmail.com, https://sourceforge.net/projects/functionpoints/
16. It’s TeeTime Example, courtesy of Q/P Management Group, © Copyright 2010. Q/P Management Group, Inc. All Rights Reserved, and Subject Matter Expertise provided by Ms. Lori Holmes-Limbacher
17. D. Jackson, Software Abstractions Reference Jackson, Daniel, 2006, Software Abstractions: Logic, Language, and Analysis. Cambridge, Massachusetts: The MIT Press
18. TutorialPoint, http://www.tutorialspoint.com/software_testing_dictionary/test_case.htm
19. F. Brooks, The Mythical Man Month, Addison Wesley, 1995
20. E.Wolff, Microservices, Flexible Software Architectures, Leanpub, 2016
21. COCOMO II MP, extension including direct input of MP file http://csse.usc.edu/tools/MP_COCOMO/, Dr. Ray Madachy
22. R. Hamming, Richard W., The Art of Doing SCIENCE and Engineering: Learning to Learn, Gordon and Breach Science Publishers, Amsterdam B.V, The Netherlands, 1997, and associated NPS Lectures.