Fall 2013

Control-Theoretic Decision Support for Mitigation of Modeled Software Project Cost Overruns

Scott David Miller
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_dissertations
Part of the Computer Sciences Commons

Recommended Citation
Miller, Scott David, "Control-Theoretic Decision Support for Mitigation of Modeled Software Project Cost Overruns" (2013). Open Access Dissertations. 100.
https://docs.lib.purdue.edu/open_access_dissertations/100

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
This is to certify that the thesis/dissertation prepared

By Scott D. Miller

Entitled
Control-Theoretic Decision Support for Mitigation of Modeled Software Project Cost Overruns

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Aditya Mathur (co-chair) 
Raymond DeCarlo (co-chair)
Ananth Grama
Xiangyu Zhang

To the best of my knowledge and as understood by the student in the Research Integrity and Copyright Disclaimer (Graduate School Form 20), this thesis/dissertation adheres to the provisions of Purdue University’s “Policy on Integrity in Research” and the use of copyrighted material.

Approved by Major Professor(s): Aditya Mathur
Raymond DeCarlo

Approved by: Sunil Prabhakar / William Gorman 09/03/2013
Head of the Graduate Program Date
CONTROL-THEORETIC DECISION SUPPORT FOR MITIGATION
OF MODELED SOFTWARE PROJECT COST OVERRUNS

A Dissertation
Submitted to the Faculty
of
Purdue University
by
Scott D. Miller

In Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

December 2013
Purdue University
West Lafayette, Indiana
ACKNOWLEDGMENTS

First, and foremost, I would like to express my deepest gratitude to my advisors, Professors Raymond DeCarlo (ECE) and Aditya Mathur (CS), who provided more encouragement and support over more years than any student has a right to expect.

I would like to acknowledge the wisdom of Dr. William Gorman, in regard to taking full-time employment during graduate school, who advised, “Don’t do it!” This sage advice should be heeded by all who do not wish to include the following in their dissertation: To the graduate study committee, who tolerated my reduced progress, and deemed to grant my final petition for an extension to the required timeline, I am deeply thankful. Correspondingly, to James Gallagher, for his support of my periodic need to drop everything in order to meet an academic deadline, I wish to express my sincere appreciation.

Lastly, to my wife Michelle, unwavering stanchion of our family, I cannot express sufficient appreciation or gratitude. It is now officially your turn.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER 1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Synopsis</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER 2. HISTORY AND CONTEXT</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Early Lessons: Pre-1970</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Issues in Determining Progress</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Issues in Estimating Schedule/Budget Adherence</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Issues in Determining Corrective Action</td>
<td>13</td>
</tr>
<tr>
<td>2.1.4 Early Descriptive Models</td>
<td>14</td>
</tr>
<tr>
<td>2.1.5 Early Prescriptive Models</td>
<td>15</td>
</tr>
<tr>
<td>2.1.6 Outliers</td>
<td>15</td>
</tr>
<tr>
<td>2.1.7 A Brief Aside on the Art of Software Development</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Age of Software Cost Estimation: 1970–1985</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1 Sizing and Complexity</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2 Productivity</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Parametric Models</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Recent Practice: 1985–Present</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1 Dynamical Models</td>
<td>24</td>
</tr>
<tr>
<td>2.3.2 Model-Based Decision Support Methodologies</td>
<td>25</td>
</tr>
<tr>
<td>2.3.3 Statistical Process Control</td>
<td>27</td>
</tr>
<tr>
<td>2.3.4 Feedback Control</td>
<td>28</td>
</tr>
<tr>
<td>CHAPTER 3. MODELING SOFTWARE DEVELOPMENT</td>
<td>29</td>
</tr>
<tr>
<td>3.1 Terminology: Task and Work</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Overview of Software Development</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Productivity</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1 Mathematical Development of Productivity Concepts</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2 State Model of Productivity</td>
<td>40</td>
</tr>
<tr>
<td>3.4 Backlog</td>
<td>41</td>
</tr>
<tr>
<td>3.4.1 Mathematical Development of Backlog Concepts</td>
<td>41</td>
</tr>
<tr>
<td>Chapter/Section</td>
<td>Title</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>3.4.2</td>
<td>State Model of a Backlog Queue</td>
</tr>
<tr>
<td>3.5</td>
<td>Activation Constraints</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Mathematical Development of Activation Constraint Concepts</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Algebraic Model of an Activation Controller</td>
</tr>
<tr>
<td>3.6</td>
<td>A Generic Development Activity Model</td>
</tr>
<tr>
<td>3.7</td>
<td>Side-Effect Generation</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Defect Population</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Defect Detection</td>
</tr>
<tr>
<td>3.8</td>
<td>Special Case: Failure Analysis</td>
</tr>
<tr>
<td>3.9</td>
<td>Model Solution</td>
</tr>
<tr>
<td>3.10</td>
<td>Simulation Study</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Example Scenario</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Simulation Outcome</td>
</tr>
<tr>
<td>3.11</td>
<td>Modeling Summary</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview</td>
</tr>
<tr>
<td>4.2</td>
<td>Definitions and Convention</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Model Equations, Inputs, and Outputs</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Components of Interest in the Model Output</td>
</tr>
<tr>
<td>4.3</td>
<td>Objective 1: Minimize the Impact of Schedule Deviation</td>
</tr>
<tr>
<td>4.3.1</td>
<td>A Notion of Schedule</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Aligning the Schedule with the Development Activities</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Determining the Status of Schedule Adherence</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Scoring Corrective Actions for Schedule Adherence</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Extensions of Schedule Tracking</td>
</tr>
<tr>
<td>4.4</td>
<td>Objective 2: Minimize the Overall Project Duration</td>
</tr>
<tr>
<td>4.4.1</td>
<td>A Notion of Project Completion</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Scoring Corrective Actions for Duration Minimization</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Accommodating Fixed Workforce</td>
</tr>
<tr>
<td>4.5</td>
<td>Defining the Control Law</td>
</tr>
<tr>
<td>4.6</td>
<td>A Justification of Quadratic Costs</td>
</tr>
<tr>
<td>4.7</td>
<td>Sub-Optimality</td>
</tr>
<tr>
<td>4.8</td>
<td>Controller Implementation</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Specification of Control Variables and Bounds</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Selection of Weights</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Establishment of Control Partitions</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Formulation of Equality and Inequality Constraints</td>
</tr>
<tr>
<td>4.8.5</td>
<td>Evaluation of the Control Law</td>
</tr>
<tr>
<td>4.8.6</td>
<td>Interpretation of Results</td>
</tr>
<tr>
<td>4.9</td>
<td>Decision Support/Control Summary</td>
</tr>
</tbody>
</table>
# CHAPTER 5. CONTROLLER EVALUATION

5.1 Scenario 1: Mis-prediction of Required Level of Effort .......................... 105
   5.1.1 Scenario Narrative ............................................ 105
   5.1.2 Analysis and Controller Construction .......................... 107
   5.1.3 Control Results .................................................. 110
   5.1.4 Details of Scenario Construction .............................. 123
5.2 Scenario 2: Allocating a Fixed Workforce ................................. 123
   5.2.1 Scenario Narrative ............................................ 123
   5.2.2 Analysis and Controller Construction .......................... 124
   5.2.3 Control Results .................................................. 126
5.3 Summary of Controller Evaluation ........................................ 133

# CHAPTER 6. APPLICATION IN PRACTICE ........................................ 136
6.1 Calibrating from Historical Data ....................................... 136
   6.1.1 Calibrating the Productivity Model .......................... 137
   6.1.2 Defect Detection .................................................. 137
6.2 Managing the Impact of Activation Constraints ....................... 138

# CHAPTER 7. SUMMARY .................................................. 139
7.1 Modeling .................................................. 139
   7.1.1 Submodels .................................................. 139
   7.1.2 Interconnection .................................................. 140
7.2 Decision Support .................................................. 140
7.3 Evaluation .................................................. 141
7.4 Future Work .................................................. 142

# APPENDICES

Appendix A. Example Model Parameters ..................................... 143
   A.1 Submodel Interconnections ...................................... 143
   A.2 Incorporation of External Inputs ................................ 147
   A.3 Model Parameters .................................................. 149
   A.4 Initial State .................................................. 150
Appendix B. Two Methods of Minimization .................................. 151
   B.1 In-line Model Solution .......................................... 151
   B.2 Model Equations as Equality Constraints ....................... 153

# LIST OF REFERENCES .................................................. 156

# VITA .................................................. 161
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relative effort by development activity in early publications.</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Correspondence between process modeling quantities, and their counterparts in the analogy with physics.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Scenario 1 weights penalizing schedule deviation.</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>Scenario 1 weights penalizing total input magnitude.</td>
<td>108</td>
</tr>
<tr>
<td>5.3</td>
<td>Weights relating the relative cost of work between activities.</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td><strong>Appendix Table</strong></td>
<td></td>
</tr>
<tr>
<td>A.1</td>
<td>Identifiers for model component types.</td>
<td>143</td>
</tr>
<tr>
<td>A.2</td>
<td>Development activity/side-effect identifiers.</td>
<td>143</td>
</tr>
<tr>
<td>A.3</td>
<td>Submodel interconnections made via $L_{11}$.</td>
<td>144</td>
</tr>
<tr>
<td>A.4</td>
<td>Incorporation of external inputs via $L_{11}$.</td>
<td>148</td>
</tr>
<tr>
<td>A.5</td>
<td>Queue parameters for simulation examples.</td>
<td>149</td>
</tr>
<tr>
<td>A.6</td>
<td>Productivity model parameters for simulation examples.</td>
<td>149</td>
</tr>
<tr>
<td>A.7</td>
<td>Defect population model parameters for simulation examples.</td>
<td>149</td>
</tr>
<tr>
<td>A.8</td>
<td>Defect detection model parameters for simulation examples.</td>
<td>149</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>U.S. revenue from custom software development, 1999–2008.</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Timeline of early events in the literature of software engineering.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Early reported proportion of project effort by development activity.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Timeline of events in the era of parametric cost estimation.</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Linearly interconnected system of state-based and algebraic submodels.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>A typical model of the software implementation phase.</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Example Gaussian model of Csikszentmihalyi’s “flow” anecdote.</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Decision logic for queue out-flow.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Asymmetry in arguments of sat-based min.</td>
<td>42</td>
</tr>
<tr>
<td>3.6</td>
<td>Illustration of constraint implementation through thresholds.</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Decision logic for the activation constraint regulator.</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>Internal feedback within the activity model.</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Specialized activity model for failure analysis.</td>
<td>58</td>
</tr>
<tr>
<td>3.10</td>
<td>Direct computation order for example implementation-phase process.</td>
<td>61</td>
</tr>
<tr>
<td>3.11</td>
<td>I/O traces for feature coding productivity.</td>
<td>64</td>
</tr>
<tr>
<td>3.12</td>
<td>I/O traces for the feature coding backlog.</td>
<td>64</td>
</tr>
<tr>
<td>3.13</td>
<td>I/O traces for test case coding productivity.</td>
<td>65</td>
</tr>
<tr>
<td>3.14</td>
<td>I/O traces for the test case coding backlog.</td>
<td>66</td>
</tr>
<tr>
<td>3.15</td>
<td>I/O traces for new test execution productivity.</td>
<td>67</td>
</tr>
<tr>
<td>3.16</td>
<td>I/O traces for the new test execution backlog.</td>
<td>68</td>
</tr>
<tr>
<td>3.17</td>
<td>Interaction of activation constraint and new test execution activity.</td>
<td>68</td>
</tr>
<tr>
<td>3.18</td>
<td>Area graph of new test execution in-flow rate components.</td>
<td>69</td>
</tr>
<tr>
<td>3.19</td>
<td>I/O traces for new regression test execution productivity.</td>
<td>70</td>
</tr>
<tr>
<td>3.20</td>
<td>I/O traces for the regression test execution backlog.</td>
<td>71</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.21</td>
<td>Estimated feature defect introduction, detection, and removal.</td>
<td>72</td>
</tr>
<tr>
<td>3.22</td>
<td>Estimated test case defect introduction, detection, and removal.</td>
<td>73</td>
</tr>
<tr>
<td>3.23</td>
<td>I/O traces for failure analysis productivity.</td>
<td>74</td>
</tr>
<tr>
<td>3.24</td>
<td>I/O traces for the failure analysis backlog.</td>
<td>75</td>
</tr>
<tr>
<td>3.25</td>
<td>Proportional splitting component inputs.</td>
<td>76</td>
</tr>
<tr>
<td>3.26</td>
<td>Proportional splitting component performance.</td>
<td>76</td>
</tr>
<tr>
<td>3.27</td>
<td>I/O traces for feature correction productivity.</td>
<td>77</td>
</tr>
<tr>
<td>3.28</td>
<td>I/O traces for the feature coding backlog.</td>
<td>78</td>
</tr>
<tr>
<td>3.29</td>
<td>I/O traces for test case correction productivity.</td>
<td>79</td>
</tr>
<tr>
<td>3.30</td>
<td>I/O traces for the test case coding backlog.</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>Process of applying decision support.</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>Observed deviation in feature coding progress, scenario 1.</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Re-calibrated estimate of feature coding progress, scenario 1.</td>
<td>105</td>
</tr>
<tr>
<td>5.3</td>
<td>Impact of delayed internal release on new test execution, scenario 1.</td>
<td>106</td>
</tr>
<tr>
<td>5.4</td>
<td>Impact of delayed internal release on failure analysis, scenario 1.</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>Computed reference trajectory, scenario 1.</td>
<td>111</td>
</tr>
<tr>
<td>5.6</td>
<td>Result of control, feature coding activity, scenario 1.</td>
<td>111</td>
</tr>
<tr>
<td>5.7</td>
<td>Result of control, remaining development activities, scenario 1.</td>
<td>113</td>
</tr>
<tr>
<td>5.8</td>
<td>Nominal and controlled workforce size, feature coding, scenario 1.</td>
<td>115</td>
</tr>
<tr>
<td>5.9</td>
<td>Nominal and controlled workforce size, remaining activities, scenario 1.</td>
<td>116</td>
</tr>
<tr>
<td>5.10</td>
<td>Total nominal and controlled workforce size, all activities, scenario 1.</td>
<td>118</td>
</tr>
<tr>
<td>5.11</td>
<td>Nominal and controlled process quality, feature coding, scenario 1.</td>
<td>119</td>
</tr>
<tr>
<td>5.12</td>
<td>Nominal and controlled process quality, remaining activities, scenario 1.</td>
<td>121</td>
</tr>
<tr>
<td>5.13</td>
<td>Total nominal and controlled process quality, all activities, scenario 1.</td>
<td>122</td>
</tr>
<tr>
<td>5.14</td>
<td>Predictions, controller-selected versus a priori allocation, scenario 2.</td>
<td>127</td>
</tr>
<tr>
<td>5.15</td>
<td>In-flow traces for scenario 2 controller diagnostics.</td>
<td>128</td>
</tr>
<tr>
<td>5.16</td>
<td>Nominal and controlled workforce size, scenario 2.</td>
<td>130</td>
</tr>
<tr>
<td>5.17</td>
<td>Incomplete failure analysis work, scenario 2.</td>
<td>131</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>5.18</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>5.19</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>5.20</td>
<td>135</td>
<td></td>
</tr>
</tbody>
</table>

Appendix Figure

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>152</td>
</tr>
</tbody>
</table>
ABSTRACT


Despite sixty years of practice, the production of software remains an endeavor that is difficult to manage according to a schedule. Control theory studies the ability to influence the dynamical behavior of systems to achieve desired behaviors or eliminate undesired behaviors. In this work, the management problem of software project schedule adherence is re-cast as a problem in control theory.

Below, a modeling framework is proposed for capturing the constraints and dependencies found in the arbitrary organization-specific work-flows underlying software production. Combined with proposed models for productivity, defect introduction, and defect detection, the framework completes a method for producing models of progress in software development using the techniques dynamical systems modeling. Management objectives are then formalized in terms of behaviors to be elicited or eliminated from the models. Finally, the techniques of control theory are applied to determine changes of the variable inputs of the model that maximize achievement of the management objectives. Simulation results are analyzed and reveal that the control technique succeeds in modifying the behavior of such models to improve adherence to specified management objectives.

The control loop is intended to be closed by the software project manager, who translates the suggested modifications of the model inputs into candidate changes to be optionally applied to the real-world software development process. As the candidate process changes enter into the decision-making process of the project manager, this work constitutes a management decision support tool.
CHAPTER 1. INTRODUCTION

Software development is an industry with clear demand. Over the period from 1999 through 2008, custom software development for U.S. firms represented more than 55 billion U.S. dollars per year in revenue with an average upward trend of more than 1.8 billion dollars year over year [1] despite the widespread economic impact of the attacks on the World Trade Center in 2001. Figure 1.1 plots this revenue data and trend.

Yet despite sixty years of large-scale software development, software projects tend to overrun their projected cost by more than 30 percent [2]. Such overruns impact business decision making and affect competitiveness. To understand how, consider the economic theory expressed as, “If a firm in class $k$ is acting in the best interest of the stockholders at the time of the decision, it will exploit an investment opportunity if

Figure 1.1. U.S. revenue from custom software development, 1999–2008.
and only if the rate of return on the investment, say \( p^* \), is as large as or larger than \( p_k \)" [3], where \( p_k \) gives the average cost of acquiring capital for firms with similar return on investment (as characterized by the equivalence class \( k \)). In short, the theory espouses a view that projects should only be undertaken if the expected rate of return on investment exceeds an economically defined threshold. A project overrun diminishes the profits, thereby adversely affecting the rate of return, potentially affecting the desirability of the project as a business investment. To hedge against the potential of such overruns, development organizations must increase bid prices to provide a margin of safety, but in so doing, reduce the competitiveness of their bids.

Thus, methods have been sought to reduce cost overruns. In the late 1960s, the software community undertook the task of determining best practices for large-scale software development [4,5]. Cost estimation was subsequently undertaken as a primary area of research interest [6–10] as evidenced in the quotation, “One of the basic goals of software engineering is the establishment of useful models and equations to predict the cost of any given programming project” [9].

Perhaps the most well-known cost estimation model is COCOMO\(^1\) II (the Constructive Cost Model, revision 2) [11]. COCOMO II is a parametric cost estimation model that has been demonstrated to achieve prediction “within 30 percent of the actuals 75 percent of the time” under Bayesian calibration to an organization’s historical data—the highest accuracy achieved of a predictive model at the time (1999) [12]. Thus, the observations from practice and the accuracy of predictive models indicate that \textit{a priori} cost estimation is a task subject to non-negligible levels of uncertainty.

This work considers the possibility that there exists a component of project cost that is unpredictable at practical levels of modeling effort but that the in-place project plan is reasonably accurate, at least within the COCOMO II limits. Under such an assumption, the research focus is shifted away from the task of explaining variability in the software development process, and is redirected to the complementary task of compensating for unanticipated deviations from the project plan. That is, the prob-\(^1\)COCOMO is a registered trademark of the University of Southern California.
lem of software project cost overruns need not be solved solely in terms of heightened predictive accuracy; the solution may also incorporate intervention to mitigate the impact of the manifestations of uncertainty as they arise. Consider an industry anecdote regarding software development: “We have learned by trial and error that no cost predictions can be fulfilled unless the mechanism for management control is solved in advance [6].”

It is proposed that such intervention can be facilitated through quantitative feedback to project management personnel regarding the magnitude and timing of interventions to be applied within a given project plan.

1.1 Synopsis

The central objective of this work is the development and demonstration of a predictive capability and associated management decision support mechanism suitable to the task of guiding management intervention in pursuit of achieving cost control and schedule adherence for software projects. This objective is distinct from that of the prescriptive taxonomies of best practices and lessons learned found in the literature, focusing instead on the task of modeling the dynamics of existing processes, and on adapting to their evolution through recalibration. Instead of a focus on long-term predictive accuracy, the modeling aspect of the work seeks to underlie a process optimization that quantifies intervention strategies relative to management objectives such as cost control, schedule adherence, duration reduction, or efficiency improvement. Generally, the intent is to fit a quantitative model’s parameters to project data, and evaluate how the predicted future project evolution tracks predetermined management objectives. To optimize expected tracking performance, quantitative management intervention strategies are proposed as modifications to the resource allocation plan.

The modeling work is carried out within a system-of-interconnected-subsystems framework, comprising linear and non-linear, state-based and algebraic submodels.
Individual development activities may be captured through instances of a generic
development activity submodel, or customized submodels defined at need. These
are interconnected with models of other intrinsic processes (such as defect introduc-
tion and detection) in a manner so as to reflect the workflows of an organization,
and thereby to capture the dominant dynamics of software development within that
organization. Further, the modeling framework includes support for modeling depend-
dencies between development tasks. Such dependencies may be intrinsic inter-task
dependencies, or extrinsic constraints—such as those imposed under an incremental
development lifecycle to define separate increments.

The decision support capability is formulated as an application in Control The-
ory. Given a model within the preceding framework, then for initial applications in
schedule management as the dominant cost driver, management objectives are for-
malized as performance indices. In the present context, performance indices can be
understood as functions that assign a numeric score to tuples consisting of a repre-
sentation of tracking errors between management objectives and the estimated future
evolution of a software project, and a representation of the resource allocation plan
that yields such. Thus, a performance index considers both the result to be achieved,
and the mechanism that achieves it. By their nature, performance indices establish a
partial order on potential management intervention strategies, and this partial order
forms the basis for a search, (specifically a minimization of the performance index)
over variables within a project manager’s sphere of influence, to determine an efficient
intervention strategy.

As the minimization occurs over a representation of management interventions,
a translation is necessary to interpret the outcome, which is specified as a repre-
sentation of a selected set of interventions. The expertise of the project manager is
required to identify real-world interventions that correspond to the numerical repre-
sentation returned by the minimization. Thus the decision support capability can be
seen to aid in the selection of magnitude and timing of interventions to be applied.
Ultimately, the project manager closes the control loop by translating and applying a corresponding real-world intervention, at his/her sole discretion.

1.2 Contributions

The major contributions of this work are

1. a framework for constructing quantitative models of software development processes
   (a) within the formalism of state-variable modeling with continuous state-evolution dynamics (so as to remain amenable to established techniques in control theory),
   (b) while capturing the non-trivial interdependencies between software development tasks (i.e., preconditions on when work may be performed), and
   (c) preserving an intuitive and qualitative mapping between model elements and the processes they represent (i.e., the overall structure is an interconnection of sub-models for each development activity);

2. a feedback mechanism for management decision support
   (a) based on identifying resource changes that achieve manager-specified behavioral changes (as represented by a performance index),
   (b) while respecting manager-specified constraints
   (c) from any model constructed within the framework; and

3. examples of the translation from business-language statements of desired behavioral changes into performance indices suitable for characterizing the feedback (intervention) mechanisms.
1.3 Organization

The remainder of the work is organized as follows. Chapter II gives a survey of the related work and establishes the context for the present work. Chapter III develops the modeling framework, and presents an example representation of the implementation-phase of software development. Chapter IV contains the mathematical development of the decision support capability. Chapter V develops examples of performance indices from narrative scenarios and presents the results of applying the resulting decision support process. Chapter VI addresses issues encountered in practical application of the proposed technique. Lastly, Chapter VII gives a summary and sets the direction for future work.
CHAPTER 2. HISTORY AND CONTEXT

In 1968, the NATO Science Committee held a conference titled “Software Engineering” [5] to bring together practitioners and theoreticians to address the crisis generated by the imbalance between the increasing need for large scale software development in industry, and the industry’s ability to deliver it reliably, in accordance with budget, schedule, and quality objectives. It is reported [13] that the title of this conference was intended to be controversial, as the discipline of software development at the time was widely acknowledged to lack the maturity of a typical engineering discipline.

Nearly 50 years later, one finds many changes in the software industry. The state of practice in “automatic programming” has advanced such that third generation programming languages and optimizing compilers are produced alongside new microprocessors, and hence form the default choice for project managers and their engineering teams. With compiler support taken for granted, high performance microprocessors have reached commodity status. Degree programs in computer science now exist at a majority of U.S. universities\(^1\) and there exist 156 advanced degree programs in computer science in the U.S.\(^2\) providing ample manpower where formerly recruitment or training of qualified individuals required a significant investment. Yet, the industry still struggles to produce software according to schedule, budget, and quality objectives.

A brief history of large-scale software development is presented to document the early emergence of the issues in software engineering. Commentary from the early practitioners is interjected to provide color for the subsequent literature survey.

\(^{1}\)Confirmed by undergraduate degree-program search at campusexplorer.com, retrieved 2013-08-19, yielding 1225 bachelor’s degree programs in computer science, out of 2204 4-year institutions in the database.

\(^{2}\)Based on 2010 U.S. News and World Report rankings of graduate programs in computer science.
2.1 Early Lessons: Pre-1970

The literature addressing software development as the subject of academic inquiry begins shortly after the advent of the first solid-state computers (mid-1950s). An early example is found in the publications describing the software development effort for the SAGE air defense system [14]. In the intervening period, software has come to play an increasingly important role in the design and development of complex systems. Consequently, the need for reliable production of high-quality software within schedule and budget constraints has continued to motivate the study of software development.

Figure 2.1 gives a timeline of major events of the early history of software development. The early literature tends to focus on two problems:

1. Estimating the size/effort/cost of a development effort—in order to produce an appropriate bid (the “estimation problem”); and

2. Managing a project to adhere to a fixed schedule and budget—once a bid is accepted (the “management problem”).

Such a focus is to be expected, given the context in an industrial setting where failure to deliver on contract deadlines can invoke significant penalties, yet including too much of a safety margin in a bid may price one out of competition.

The management problem is first addressed in the form of lessons learned and best practices; prescriptions for how to structure a development effort are given alongside

In addressing the management problem, three subdivisions are identified:

1. determining the level of progress

2. estimating the degree of future adherence to schedule and budget objectives

3. determining what corrective action to take, if any, given schedule/objective lags

2.1.1 Issues in Determining Progress

“When projects tended to make rapid progress toward completion when work first began, it took an inordinately long time to get from 90 per cent to 100 per cent complete. [19]” This quotation reflects a phenomenon well-known today in software development organizations, as the “90% complete syndrome”. The phrase denotes the tendency of a software developer to self-report progress on a task as being 90% complete for a period far longer than 10% of the overall task duration. In response to this phenomenon, Haine and Lob [19] resort to assigning a boolean completion status to a fine grained task decomposition/work breakdown:

Our experience with [monthly progress reporting] indicated that the shortcomings of the concept “per cent complete” were sufficiently great to negate its value. [...] The “per cent complete” limitation has been circumvented by having the project engineer subdivide his project into the milestones of technical accomplishment by which progress can be evaluated.
Similarly, Kornreich provides an early reference (1962) to the problem of tracking the progress of a software development effort. His paper [21] describes a quantitative technique for generalizing past performance data for the purposes of

1. evaluating new proposals for their compliance to the norms of past successful projects, and

2. determining whether a contract effort is falling behind, thereby foreshadowing a budget and/or schedule overrun.

Roberts [24] finds similar issues in industry on gauging progress:

For example, the author has examined one large company’s survey of several hundred engineering progress reports, in which the project leaders were so often confounded as to what progress had actually been achieved that during about 80 percent of the actual time spent on a project, the supervisors reported the job to be about 90 percent completed.

Pearlman [22] opines that

The planning of technical objectives and the accompanying measurement of success is perhaps the most nebulous aspect of program management.

Wolverton [6] echoes Pearlman’s frustration regarding the ability to quantify the level of progress in software projects:

Because software is during most of its development cycle (and even afterward) a basically intangible product, it is even harder to control than a complex hardware system.

2.1.2 Issues in Estimating Schedule/Budget Adherence

Progress estimation is only the first step in managing software development. Data collection from real processes is an application of measurement and measurement is
typically noisy (in the stochastic sense). A project manager must interpret noisy data to determine whether project progress and quality are appropriate given the observed schedule and budget consumption—or whether corrective action is necessary. “Control charts” (alternatively, “Shewhart charts”) are among the earliest and most wide-spread technique for interpreting noisy process data. First described by Shewhart [15], they provide, as Juran succinctly recalls [25], “a perpetual test of significance,” allowing the reader to distinguish quickly between events that are likely to represent real deviations from the project plan/schedule, and those that are likely to arise as artifacts of noise. Pearlman [22] suggests monitoring actual cost expenditure over time, or feature completion as a function of cost, as alternate representative measures of status:

One method of measurement is based upon the forecasted expenditures which are re-planned by the project engineer every odd month. This value is compared to the difference between the authorized funds and the expended funds. Obvious actions will result from differences greater than 5 per cent. A more valuable tool is an intuitive comparison between schedule success and cost.

Kornreich [21] espouses a similar use of data, though in his work the values are normalized into percentages, and are categorized into “types” of success (i.e., drawings delivered, prototypes produced, etc.) which are tracked against total project budget expenditure, or total project schedule elapsed. Kornreich notes that control limits have been established for many of the success categories, but does not note their values or form.

A number of authors give a breakdown of the nominal project schedule into the proportions allocated to each development activity. These divisions are summarized in Figure 2.2 and Table 2.1. From the figure alone one can see that the value of such breakdowns is found with respect to the organization from which they originate. In the absence of such context, it would be difficult to make use of such coarse
Figure 2.2. Early reported proportion of project effort by development activity.

Table 2.1
Relative effort by development activity in early publications.

<table>
<thead>
<tr>
<th></th>
<th>Requirements</th>
<th>Design</th>
<th>Implementation</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wolverton [6]</td>
<td>12.1%</td>
<td>34.1%</td>
<td>20%</td>
<td>33.8%</td>
</tr>
<tr>
<td>Benington [14]</td>
<td>30%</td>
<td>23%</td>
<td>8.1%</td>
<td>38.9%</td>
</tr>
<tr>
<td>Albrecht [26]</td>
<td>7.9%</td>
<td>12.1%</td>
<td>31%</td>
<td>49%</td>
</tr>
<tr>
<td>Aron [27]</td>
<td>0%</td>
<td>30%</td>
<td>40%</td>
<td>30%</td>
</tr>
</tbody>
</table>
and widely varying reports to establish a test for whether a development effort is proceeding according to plan.

2.1.3 Issues in Determining Corrective Action

On the importance of maintaining control of a software project, Wolverton [6] recounts his experience:

We have learned by trial and error that no cost predictions can be fulfilled unless the mechanism for management control is solved in advance.

A number of anecdotes are offered to describe the sources of schedule and budget deviation. Parkinson’s Law [16] is a well-known management axiom stating that “work expands to fill the time available for its completion.” The following references support Parkinson’s Law for software development, yielding one area for management oversight and control: Hosier [23] states

Programmers differ little from engineers, in general, in their reluctance to stop tinkering with and improving their creations. This is a laudable trait; but as delivery dates approach and time grows short, it has to be restrained.

Boehm [28] notes that a software engineer is presented with “an unlimited number of temptations” to add “gold-plating”, which he defines as

features which make the job bigger and disproportionately more expensive, but which turn out to provide little help to the user or maintainer when put into practice.

In Boehm’s context, “features” refers both to system functionality, and non-functional qualities such as performance, size, and precision. Albrecht [26] relays the importance of change control perceived at IBM, which hints at the gold-plating phenomenon in its emphasis of the cost/value trade-off:
Above all we control change—not to prevent change, but to make sure each party understands the value and cost, and approves the change, before it is implemented.

Wolverton [6] offers an opinion on software developers that may account for the tendency toward gold-plating:

The nature of programmers is such that interesting work gets done at the expense of dull work [...]. Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost.

2.1.4 Early Descriptive Models

**Definition 2.1.1** A “descriptive model” captures the method by which a process is, or has been, carried out. Descriptive models typically form the starting point for introspective or retrospective analysis [29].

In the late 1950s, Benington provides a high-level overview [30] of the production of large software systems. Within this overview, a diagram is given illustrating the paths of information flow between activities (some labeled for the artifacts they produce). The major sequence dictated by the proposed information flow is

\[ \text{Plans} \rightarrow \text{Specifications} \rightarrow \text{Code} \rightarrow \text{Test/Evaluation Results}. \]

Four years later, Hosier gives a detailed description of the activities undertaken during development of a digital real-time system, the producer/consumer relations among them, and the types of data/artifacts produced [23]. This descriptive model explicitly documents the feedback cycle for hardware change, but fails to capture the tendency for rework in software activities (well-understood in retrospect). Much of the discussion surrounding the model addresses the need for the system specifications to be fixed before coding begins as a remedy for the tendency of programmers to study/experiment/simulate endlessly, rather than undertake the construction of the
production code. This prescriptive sentiment is fully developed a decade later with the emergence of the venerable waterfall model [31].

2.1.5 Early Prescriptive Models

**Definition 2.1.2** A “prescriptive model” sets forth a method for carrying out a process. Such models typically seek to achieve desirable outcomes by advancing a method that explicitly mitigates commonly observed poor practices [29].

Near the beginning of the 1970s, the software development community took up the task of determining best practices for developing large software systems. Surrounding the NATO conference the managers of various large-scale software developments began to share their collected anecdotes for how to drive software development to acceptable levels of quality while respecting schedule and budget bounds.

One such early account is that of Royce in 1970 [31], in which is laid out a general plan for large-scale software development—the rudiments of which are now commonly known as the venerable, if not deprecated, “Waterfall Model”. (Contemporary readers who peruse the aforementioned work may be surprised by the elaborateness of Royce’s model, given the simplification of it in common use. Also, while the waterfall model is commonly attributed to Royce, it is clearly formalized in the descriptive model of Nash presented at the NATO conference on Software Engineering two years earlier. Other authors also note potential caveats in the ascription of the waterfall model to Royce [32]).

2.1.6 Outliers

Three contributions of the 1960s take decidedly different directions than the contemporaneous literature:

In *Industrial Dynamics*, Forrester [33] initiates an area of modeling well beyond the scope of what had been undertaken previously (and perhaps well ahead of the
computational power required for its practical adoption). The theory of industrial dynamics espouses an approach of modeling complete economic systems, be that at the level of a corporation, industry, or a complete economy [33]; such a model is proposed for introspection, rather than prediction, to understand the impacts of decisions, and the causal relations underlying events. Later, Forrester generalized the concepts of industrial dynamics to other modeling domains. Under the name “system dynamics” this later modeling paradigm is found three decades later to dominate the literature on new modeling work in software engineering. (See Section 2.3 for coverage of system dynamics modeling).

Next, PERT [22] and CPM [20] are concurrently developed as methods for project analysis based on a graph-theoretic representation of project tasking/work breakdown. Graph-based representations are used when planning to identify personnel-loading assumptions, and to gauge the amount of slack or rigidity (and thereby, risk) that a given schedule incorporates along its “critical path”. PERT and CPM are interesting for their construction of formal representations of the intangible aspects of software development projects, and subsequent analysis of such representations to expose additional assumptions or requirements. CPM is distinguished from PERT by providing a framework for assigning cost functions, toward the goal of computing optimal schedule solutions.

Lastly, Kornreich provides one of the earliest published attempts [21] to extrapolate rigorously from past performance data to form a basis for evaluating new plans and for tracking expected completion of new contracts. From 1958–1962, the U.S. Air Force developed a normalization scheme for past R&D acquisition programs, in which historical data was partitioned into categories (1) by the function, type, and quality of system to be produced; and (2) such that a high degree of correlation is achieved between all pairs of partition members with respect to the normalized time-series of percent completion over percent of contract duration expended for all phases of the development effort (e.g., detailed specification, breadboard work, etc.) This conversion of historical project data into “percent complete” over “percent of contract
time” form motivates the given name, “normalized curves”. The requirement that a set within the partition of past projects must exhibit high mutual-correlation is made to support the conjecture that the historical data is representative of all projects of the category/type. Also normalized (to “percent expended”) are the planned contract costs, planned labor hours, and planned subcontract expenditures. These additions allow a schedule-independent view of project progress by re-plotting the normalized curves over “[percentage of] contract funds expended” rather than calendar time—Kornreich notes that this view is often more aptly diagnostic when using normalized curves for tracking progress. A significant portion of the paper is dedicated to the method by which one might adjust such normalized curves to take into account the specific properties of a new project so that the resulting modified curves form a predictive structure from which to track whether the project is proceeding “normally.” For a project deviating outside of established acceptable variational limits on such adjusted normalized curves, a predictive capability is described, using the curves, to estimate the likely resulting cost and schedule overruns.

2.1.7 A Brief Aside on the Art of Software Development

   In the early literature, it is common to find quotations of management personnel stating that software production is a different kind of work, more of an art or craft than an engineering process. For example, d’Agapeyeff is quoted in 1968 stating, from a cost-management perspective, “Programming is still too much of an artistic endeavor” [4]. The analogy with art may be more apt than first seems. In a study of arts patronage contracts executed during the Renaissance period, O’Malley describes the evolution of contract language to include more strict specification of the responsibilities assigned to patrons and sponsored artists. Later contracts begin to require the artist to absorb the cost overruns incurred by poor planning, rework, and maintenance (at times, even when rework is demanded a matter of aesthetic disagreement [34]—the ultimate untestable requirement). Quality is also brought to
light as an important aspect of commissioned work, echoing much modern sentiment on software quality:

a member of a civic council in Prato expressed [...] during a meeting about a new painting for the town hall: “if the said work is not excellent...it will bring more dishonor than honor, and, even if obtained with a low cost, it will be completely useless.” [34]

At the time, the lavishness of one’s possessions, hence the quality of one’s commissioned works, was used as an indication of credit-worthiness, and was thus critically important to the wealthy patron class [34]. Master artists maintained workshops where portions of commissioned works were frequently delegated to skilled apprentices. There are numerous documents indicating that even the acknowledged master artists occasionally struggled with cost estimation. Most famously, Leonardo Da Vinci was forced to plead for additional funds to complete the Virgin of the Rocks, purportedly for underestimating the cost of gilding the woodwork [34]. Hence, O’Malley notes:

the most successful Renaissance master painters might more appropriately be called painter managers. Indeed, around the end of the fifteenth century management skills begin specifically to be recognized in some painters’ contracts as central to the production of a work of quality. [34]

Thus, the problems encountered in planning, estimating, and managing creative processes to achieve high-quality work on-schedule and within-budget from a team of highly skilled workers have been recognized since at least the turn of the sixteenth century.


In his 1992 treatment of software cost estimation models, Heemstra notes the rapid growth of software cost estimation models during the 1970s. Figure 2.3 gives a
1972 • Halstead initiates “software science” [35, 36].
1974 • The “TRW” model is published [6].
1976 • McCabe’s complexity metric is published [37].
1977 • The “Boeing” model is published [38].
1977 • The “Doty” model is published [39].
1978 • The SLIM model is published [40].
1979 • IBM publishes the concept of “function points” [26].
1981 • Industrial models recognized as non-portable [9].
1981 • Objective measures of “complexity” gain prominence [41–43].

Figure 2.3. Timeline of events in the era of parametric cost estimation.
timeline for the model development during this period. The cost estimation problem is commonly divided into two parts: sizing, and productivity estimation. These parts are sometimes addressed as research areas in themselves, and are sometimes treated as components of a comprehensive method of cost estimation.

2.2.1 Sizing and Complexity

Albrecht [26] describes a technique for estimating project effort that was developed at IBM. This technique is based on measuring a language-independent metric of the amount work to be performed called “function points”. Function points are computed as a sum of weighted counts of external characteristics of the target software specification, adjusted through expert-estimated qualitative scaling factors. Using function points as an objective representation of effort reduces subjectivity in estimation, and provides a uniform representation upon which past data can more easily be applied across teams, subject areas, and languages.

Halstead’s controversial work defining “software science” occurs during this period. Independent of its eventual acceptance or rejection, Halstead sought to derive a unit of work for cognitive tasks from first principles. His difficulty metric was intended to scale an information-theoretic notion of program size (called “volume”) to create a representation of the total effort required to implement the program. In this regard, it is similar in objective to contemporaneous work, which seeks to derive a measure of needed effort from characteristics of the tasks to be accomplished [44].

McCabe [37] proposed a metric called “cyclomatic complexity” that is based on the control-flow graph of the target program. Relative to this metric, McCabe finds that structured programs yield the lowest complexity—in particular, for a structured program with a single entry and exit point, the cyclomatic number is shown to be equal to one plus the number of decision points in the program. Later work illustrates a strong correlation between McCabe’s cyclomatic complexity, Halstead’s difficulty metric, and the size of a program in source-lines of code [45].
2.2.2 Productivity

Upon estimation of the amount of total effort required for a project, the next task is modeling the production of worker effort over time—productivity. Excepting Halstead, work on productivity modeling is primarily undertaken during this period as an empirical component of parametric cost and schedule estimation models.

Halstead [7,35] pioneers the adaptation of psychological results directly to software project schedule estimation. The fundamental unit studied is the “mental discrimination” that must be made by a programmer in coding an algorithm. Halstead and collaborators provide a number of studies that support the underlying hypothesis that coding time is proportional to the number of mental discriminations a programmer must make. Later, Halstead and Funami [46] extend the idea to model defect introduction as a function of mental discriminations. Ottenstein [8] extends this work to an explicit theory for predicting the number of defects present in a software module based on quantitative results in psychology, and Halstead’s software science.

2.2.3 Parametric Models

Aron [27] describes a quantitative estimation scheme wherein the design of a software system is refined down to “units”, defined as the level at which the designer turns over control for detailed design to the implementing programmer. Because historical data indicates that a given designer is relatively consistent in the level of detail where he marks this transition, a unit-level design can be translated to the expected number of resulting instructions by calibrating to historical implementations built to the designer’s specifications. Once such a size estimate is obtained, it is translated to a duration estimate using a table whose indices are qualitative estimates of the software to be constructed, such as novelty, difficulty, and duration (to account for learning). Such a table is therefore organization-specific, and is calibrated to historical data.
The Software Lifecycle Management (SLIM) model is a sizing and scheduling model due to Putnam [40]. SLIM derives from observations by Norden on the tendency of measurements of software completion to evolve according to Rayleigh curves. The model attempts to calibrate such Rayleigh curves to past and present project data to provide a predictive capability.

The so-called “Doty model” [39], resulted from a U.S. Air Force contract to Doty Associates for the purpose of studying the cost and schedule drivers of large-scale software development. The resulting model is based on applying a linear cost-per-unit-effort over an estimated effort produced as $x = a(size)^b$. Here, $x$ is effort measured in man-months, size is a measure of the expected program size in thousands of source instructions (with alternate parameters identified for measuring size in words of object code), $b$ indicates economies ($b < 1$) or diseconomies ($b > 1$) of scale within the development effort, and $a$ gives a linear scaling of the result to produce an estimate of the needed effort. Both $a$ and $b$ are derived empirically by non-linear regression. For smaller programs (< 10,000 source instructions) 14 effort multipliers are selected by answering yes-or-no questions, and applied to scale the resulting effort estimate in man-months. The effort multipliers range in magnitude from 0.90—programmer has unlimited access to the computer, to 1.92—is the first software developed for the CPU. Such regression models are characteristic of the period.

Also during this period, the COCOMO [28] model was formulated. The roots of COCOMO can be seen clearly in the Doty model; its effort model takes the same form of $x = a(size)^b$, for size estimated in thousands-of-lines-of-code. The effort model in (basic) COCOMO is distinguished from that in the Doty model by providing alternate sets of the model parameters, $a$ and $b$, depending on a qualitative classification of the subject development effort as “organic”, “semi-detached”, or “embedded”. This classification is assigned based on team size and the rigidity of the constraints that define the effort. Additionally, COCOMO adds models for translating the effort estimate into a project duration (in calendar time) and into a suggested workforce size. COCOMO is the predecessor to the well known COCOMO II model, still in widespread
use contemporary with this writing; the latter enhances the former by providing parametric models for estimating the parameters $a$ and $b$ based on qualitative metrics of the project, development team, and development organization.

Similar to the Doty model, the “Boeing model” [38], results from a U.S. Air Force contract intending to understand the impact of modern programming practices. The Boeing model consists of a linear cost estimate based on estimated program size, wherein a cost-scaling factor is selected according to the type of application to be developed. This initial estimation is followed by a cost-refinement phase in which qualitative aspects, such as similarity to past projects, are taken into account.

Highlighting the tendency of the period, the IBM-FSD model, due to Walston and Felix, is described by Boehm [28] as a pair of parametric models taking the form $\text{eff} = 5.2 (\text{size})^{0.91}$; $\text{duration} = 2.47 (\text{effort})^{0.35}$ where eff is a prediction of the effort needed to complete the project (in man-months), and size is an estimate, in thousands of lines of source code, of the work-product to be produced. The constants 5.2, 0.91, 2.47, and 0.35 are derived from past project data via regression.

Many others propose parametric models based on a simple algebraic form with calibrations to past project data through linear or non-linear regression. Boehm gives a survey and comparison of the parameters marked as important by the various parametric models [10]. Heemstra, reporting later [47], cites a report by German analysts detailing the prolific parametric model production of this period: “A study by Noth and Kretzschmar [6] found that more than 1200 different [cost and schedule] drivers were mentioned.”

2.3 Recent Practice: 1985–Present

A colorful introduction to the history of process control is given by NIST [48] in which the economic forces of a competitive capitalism provided early informal “bang-bang” quality control systems. Here, manufacturers attempt to retain and grow their market share in response to competitor product advances. NIST also cites the
historical precedent for recognizing the value of “process” as the central determining factor of quality, evidenced as far back as the medieval trade guilds.

2.3.1 Dynamical Models

Forrester provides an early suggestion [17, 33] that the modeling methods of engineering could provide a rigorous foundation for management decision making. He notes that the contemporaneous work in management science tends to take the modeling methodology of physics and biology; an unfortunate choice, by his reckoning, due to the relative simplicity of the subjects:

Economics and management, like engineering, deal with aggregate systems above the level of the individual elementary events that are the subject of many physical science models. [...] engineering and social systems have a continuous gradation (from the obviously important, through the doubtful, into the negligible) in influences that affect each action and decision; by contrast, the physical science systems [often have] a substantial gap in importance between the few factors that must be included in a model and nearly insignificant ones that can be omitted. (Section 4.2, p.53) [33].

Instead, Forrester lays out a modeling methodology called “industrial dynamics”, wherein a modeler specifies the relationships between several quantities (“levels”) via update functions (“rate equations/decision equations”) where rates and levels must alternate along any flow. The models operate over six types of data: information, material, orders, money, personnel, and capital equipment. Ideally, one produces a closed model describing the entire industry including customer motivations, supplier behaviors, and internal corporate decision making. The stated purpose of such models is to understand the dynamics of the organization—to determine how sensitive the overall process may be to changes or uncertainty in some variable, or to interpret (in full context) data such as inventory levels relative to customer orders. The potential predictive aspect of such dynamical models is left unexplored in Forrester’s work,
though he gives examples clearly intended to demotivate such pursuit. Later, after recognizing the lack of acceptance of industrial dynamics [49], Forrester generalizes the modeling framework to arbitrary systems, seeding contemporary interest in the discipline of “system dynamics” modeling [50].

2.3.2 Model-Based Decision Support Methodologies

Assume a software project manager has a decision support objective defined as a set of characteristics to be elicited or eliminated in the future evolution of a software development effort, while regulating the resource expenditure required to effect such a change. For example, a manager may specify a desired cumulative work-completion profile over time, or a desired reduction in the defect introduction rate, but may also be constrained in terms of workforce or budget. Next, consider two methods of deriving decision support information from a given calibrated model of the software development process: Let the phrase “interrogative method” denote a Monte Carlo search over a domain defined by a subset of the model’s variable inputs, for which the intent is to find a set of input values that best achieves the decision support objective. As counterpoint, let “automatic method” denote a method of searching, again over a domain defined by a subset of the model’s variable inputs, but guided autonomously by evaluation and analysis of a “performance index”—a function of the model’s variable inputs that produces a numerical scoring of the degree to which the decision support objective is achieved.

Monte Carlo methods gained practical feasibility as a direct consequence of the availability of the first electronic computer in 1945 [51]. In Industrial Dynamics [33] Forrester describes Monte Carlo analyses designed to aid in management decision making (specifically to evaluate the potential impacts of management action or policy change), and to reveal the sensitivity-to-change embodied in a given industry-model. Malcolm et al. [18] give another early example of Monte Carlo analysis for decision support using PERT: They analyze the overall schedule impact of randomly
perturbing individual tasks (as a representation of specification changes or additions) to determine the schedule risk, from which an appropriate level of risk-mitigation can be planned.

Monte Carlo analyses for management decision support need not be simulation based, however. In general, Monte Carlo analyses arise from any predictive system whose inputs may be randomly varied. For example, consider the COCOMO software project effort prediction model; though it has a closed algebraic form, it can serve as the basis of a Monte Carlo decision support method. Consider a manager contemplating a feature addition: he can compute the COCOMO effort prediction, augmenting the estimated project size by a range of estimated sizes for the new feature, and thereby determine if the effort increase (and corresponding cost increase) are likely to produce a favorable cost/benefit tradeoff for implementing the contemplated feature.

The automatic method is far less prevalent in the literature, but appears in an early instance in the discussion of the Critical Path Method [20], wherein a piecewise linear objective function is defined to score potential renderings of a project schedule based on a notion of “slack” and the impact of slack on risk and cost given the hard interdependencies of a plan and the direct cost of labor. Here, slack (in contrast to “rigidity”) denotes the amount of time a task may be delayed within a schedule before its non-completion directly causes the delay of subsequent scheduled tasks, potentially rippling delays throughout the schedule. Parametric linear programming is applied to allocate the available slack on all non-critical paths through the task network to produce a schedule with an “optimal” balance of cost and risk (relative to the piecewise-linear performance index). The early applications of the automatic method are criticized by Needham and Aron [52] who believe that computer scientists’ predilection for elegance tends to cause them to oversimplify problems to the point that their optimal solutions have little to do with the solution to the real-world problem. Abdel-Hamid and Madnick [53] produce a model of software development by re-casting the Waterfall lifecycle phases within the framework of industrial dynamics—to which they attempt to apply state variable control [53].
Lastly, the work of Cangussu et al. [54] produced and validated a piecewise linear state-variable model of the software testing process, and provided an automatic control technique based on minimally adjusting the eigenvalues of the resulting linear system to achieve the desired quality level by a scheduled deadline. A second automatic control technique based on model predictive control, rather than eigenvalue specification, was developed for Cangussu’s model [55].

The present work falls into this last category: causal dynamical modeling for the purpose of achieving automatic decision support. The modeling framework proposed herein constructs development activity models as a combination of three concerns: productivity modeling, work-backlog tracking, and management of constraints on work availability. Development activity models are interconnected via conservative flows, alongside other submodels representing “side effect” processes such as defect introduction and defect detection, to compose an overall interconnected system of subsystems.

Rather than adopting Forrester’s approach of ad hoc interconnected dynamical components, the present work follows DeCarlo and Saeks, who develop a theory of interconnected dynamical systems [56] that trades the freedom of unrestricted model construction in exchange for useful results in control theory regarding the controllability of the resulting models. Their modeling restrictions permit arbitrary nonlinear state-based and nonlinear algebraic submodels, but limit the interconnections to be linear in nature, as are most conservation laws in science and engineering.

2.3.3 Statistical Process Control

The origin of statistical process control is traced to a 1924 memorandum issued by Shewhart of Bell Laboratories [48]. Shewhart’s innovation was a chart that provided, in Juran’s words, “a perpetual test of significance”, allowing one to distinguish at a glance whether a disturbance was likely noise, or whether it was causal. Such charts
are now known as control charts, or Shewhart charts. Shewhart’s subsequent book [15] laid the groundwork for wide-spread adoption of statistical process control.

Note: statistical process control is focused on the task of detecting significant disturbance; not the task of compensating for such disturbances. A more apt name may have been statistical process monitoring.

2.3.4 Feedback Control

Juran, describing AT&T’s management of the logistics of building and maintaining a nationwide telephone network in the mid-20s [25], gives one of the earliest references to feedback in quality control systems: “[AT&T] established a system of data feedback on quality of service and on field quality failures.”

Haine and Lob [19] prescribe a large-scale corporate organizational structure and reporting process inspired by closed-loop feedback-control servomechanisms. In such an organization, the feedback circuit is based on maintaining and updating a project schedule chart augmented with quantitative metrics to reflect the expected progress and cost-to-completion for the software project. In cases of corporate emergencies, having such data allows the command chain to be effectively used to estimate long term future impacts of short-term task cancellation and postponement. The major focus of the prescriptive process is placed on breaking the work into pieces sufficiently small so as to be accurately estimable, while publishing updates to the plan and schedule in a regular fashion. Such a scheme may work well at a large organization, due to the inherent averaging that will occur among the numerous projects executing concurrently. Haine and Lob, however, give no statistics on their project success rate using the system.

The work in feedback control, as applied to software development, by Abdel-Hamid and Madnick, and later by Cangussu is covered in Section 2.3.2, but is referenced again here for completeness.
CHAPTER 3. MODELING SOFTWARE DEVELOPMENT

This chapter sets forth a framework for constructing quantitative behavioral models of work-completion in the implementation phase of software development. The resulting models serve as the object of control activities in subsequent chapters. The framework takes a system-of-systems approach to modeling. Below, several modeling

![Linearly interconnected system of state-based and algebraic submodels.](image)

Figure 3.1. Linearly interconnected system of state-based and algebraic submodels.
components are described independently; when combined via linear interconnections a
broad range of development activity models can be represented. Figure 3.1 illustrates
the interconnection concept [56]. In the figure, several state-based models are shown,
each comprising a state-evolution function $f(\cdot)$ and an output generation function $g(\cdot)$. In addition, several algebraic models are depicted, consisting only of an output
generation function. The individual subsystem outputs are related to the subsystem
inputs as described in Equation 3.1,

$$\begin{bmatrix}
a \\
y
\end{bmatrix} =
\begin{bmatrix}
L_{11} & L_{12} \\
L_{21} & L_{22}
\end{bmatrix}
\begin{bmatrix}
b \\
u
\end{bmatrix}$$

(3.1)

where $u$ gives the vector of external inputs to the system, $y$ is the vector of external
outputs, and the vectors

$$a = [a_1, a_2, \ldots, a_n]^T$$

(3.2)

$$b = [b_1, b_2, \ldots, b_n]^T$$

(3.3)

$$x = [x_1, x_2, \ldots, x_m]^T$$

(3.4)

are related by

$$b = g(x, a) = [g_1(x_1, a_1), \ldots, g_m(x_m, a_m), g_{m+1}(a_{m+1}), \ldots, g_n(a_n)]^T$$

(3.5)

$$\dot{x} = f(x, a) = [f_1(x_1, a_1), \ldots, f_m(x_m, a_m)]^T$$

(3.6)

That is, there are $m$ state models and $n - m$ algebraic models in the interconnected
system. The matrix partition $L_{11}$ defines the direct interconnections from subsystem
outputs to subsystem inputs. $L_{12}$ describes how the external inputs $u$ enter into the
subsystem inputs. The lower partitions, $L_{21}$ and $L_{22}$ describe the construction of the
external outputs from the subsystem outputs, and feed-through from the external
inputs, respectively.
In the sections that follow, the concepts of productivity, outstanding work, and task-scheduling constraints are developed. The resulting sub-models are combined into a generic software development activity model. Next, models of unintentional “side-effect” processes are proposed. Such side-effect models are interconnected with development activity models to provide a method for representing project- and organization-specific software development processes. Following this, a numerical algorithm is given for solving the initial value problem (IVP) specified by such interconnection models. Solution of the model IVP, for a given parameterization and set of inputs, yields an estimate of the expected evolution of work completion over time. Lastly, a concrete example model is constructed based on Figure 3.2, and is evaluated to provide a basis for observing the modeled behavior.

3.1 Terminology: Task and Work

Recall from Section 2.2.1 that objective notions of software complexity became a subject of academic inquiry during 1970–1985 as focus shifted toward the development of project-, team-, and organization-independent models. In present practice, multiple complexity metrics are employed, including still those of Halstead [35], McCabe [37], and the venerable KSLOC—thousands of source lines of code [45]. The present work takes a flexible approach to complexity: conceptually, it as a factor that scales a uniformly counted “number of assignments” to a heterogeneous “level of effort required to complete each assignment”; practically, it is permitted to be any metric that satisfies this representational concept. Without loss of generality, we assume a complexity computation implemented as a convex combination of relevant, established, external complexity metrics per the method of Cangussu [54].

Let “task” denote an assignment within the context of software development. That is, an objective that a given set of individuals must achieve, and thereby complete the task. Contrast this with “work”, defined as the amount of mental effort required
to accomplish a task; a quantity directly proportional to the complexity of a task. Each task therefore represents an amount of work, and all work derives from tasks.

The distinction is required due to the desire to understand productivity, which is a function of work, within a process specified primarily in terms of tasks.

3.2 Overview of Software Development

Software development is a social, collaborative undertaking in knowledge-work. The types of tasks involved range from creative problem solving to rote book-keeping, and depend on the project as well the host development organization. Due to the this flexible nature, modeling efforts for software development tend to classify types of tasks into abstract categories based on the nature of inputs consumed and work-product produced in completing the task. The resulting notion of “development activities” is intended to partition the workforce into functional teams—groups of people performing similar tasks—which can then be modeled in aggregate.

Under such a partition, work is defined relative to the development activity, in terms of the effort required to process the types of tasks destined for the activity. For example, the activity capturing production of project source code, may define its measure of work in terms of function points, or number of uniform requirements implemented, whereas the activity of failure analysis my define its unit of work in terms of the complexity of the features and/or subsystems in which failures must be traced to their root-causes.

The resulting conception is that a development activity constitutes a process, driven by a workforce, for consuming tasks and generating work product. Activities do not typically operate independently, however. In some cases, the resulting work product directly provides the tasks consumed by a subsequent activity, thereby establishing task chains, or workflows. In other cases, side-effects of task completion by other activities interact to generate tasks for an activity; e.g., the accidental introduction of defects during coding, and their subsequent probabilistic revelation through
Figure 3.2. A typical model of the software implementation phase.

testing, as the source of tasks for the failure analysis activity. As a result, software development can be understood as an interconnection of development activities and side-effect processes: a directed graph. Figure 3.2 gives the directed graph for a typical model of the implementation phase in the software development lifecycle. In the figure, side-effect processes are represented as rectangles, with ellipses representing development activities. Solid edges carry work to consumer activities, dashed edges note parametric dependencies.
Definition 3.2.1 Let “productivity” denote the rate of work completion for a development activity.

As with the early cost estimation models, productivity is a quantity of primary interest. Where the rate of work arrival outpaces productivity, an accumulation occurs—this is the familiar concept from Queueing Theory, adapted to continuous-time semantics.

Definition 3.2.2 Let “backlog” denote the collection of outstanding work to be completed by a development activity.

Note, however, that the presence of work in a backlog does not imply that such work is ready for processing. Consider a workflow from an activity that produces new test cases to an activity that executes them: If the features under test have not yet been completed, then the test cases cannot be meaningfully executed. In this example, an additional constraint is intrinsic in the nature of the tasks being performed; there are extrinsic sources of such constraints as well. For example, in some phased contracting situations, tasks are partitioned and a constraint is imposed that tasks within each partition may not be initiated before external approval is given, even when all intrinsic dependencies have been satisfied. Hence, satisfaction of all dependencies required to begin a collection of work is a more complex consideration than simple arrival of the work in the backlog of its consuming activity.

Definition 3.2.3 Let “activation constraints” refer to the complete set of constraints determining whether work can be processed by an activity.

The notions of productivity, backlog, and activation underlie the causal relations to be captured in the modeling work of this chapter.

3.3 Productivity

It is common in practice to estimate productivity empirically by measuring work-product production over time. The approach to productivity modeling developed
below is rooted in empirical calibration, but adjusts the calibrated value dynamically as a function of development-process factors. Productivity of a development activity is defined as the activity’s rate of work completion over time; as of yet the representation of neither work nor time has been addressed.

3.3.1 Mathematical Development of Productivity Concepts

Representing Time

All subsequent modeling work is invariant under arbitrary scaling of the time axis, assuming calibrated parameters are scaled appropriately. Hence the selection of a timescale is arbitrary and is left the the choice of the modeler. Without loss of generality, the subsequent discussion shall assume a timescale in calendar weeks.

Representing Work

There are many metrics proposed in the literature that seek to translate from a set of diverse tasks to uniform notion of work. It is desirable to accommodate diverse representations of work. Hence, rather than prescribing a representation for work, the assumptions regarding the representation are listed. The modeler is free to use any representation that satisfies the assumptions. Indeed, it is anticipated that distinct representations of work may be defined for each development activity.

Assumption 3.3.1 Measurement of work is presumed to satisfy linearity.

That is, when measuring the amount work embodied in a composite task, the total work should be equal to the sum of work measured from the component sub-tasks. That is, effects on productivity that arise from the overall amount of work that an individual is assigned to process should be accounted for in the productivity model, not the measurement of work.

Assumption 3.3.2 The representation of work is assumed to be time invariant.
That is, a task that requires 6 units of work at time $t$, should require 6 units of work if it is delayed until time $t'$.

**Force Balance**

The productivity model is based on an analogy with Newton’s Second Law. Such physics-based productivity models have been shown to predict the behavior of the system-test phase of the software development lifecycle adequately [54]. Table 3.1 gives the correspondences in the physical analogy, which are explained below.

In the context of software development, let mass $m$ represent task complexity, and distance $d$ represent the total task completion count. Note that the number of tasks completed is somewhat meaningless without an understanding of the relative complexity of the tasks. Thus $md$ is the meaningful measure of progress, as it is proportional to the work accomplished. Velocity $v$ is the rate of task completion (events per unit time), and momentum $\rho$ is proportional to productivity. The net force acting on a system is equal to the derivative of momentum,

$$ F_{\text{net}} = \dot{\rho} $$

(3.7)

That is, changes in productivity are the result of the net force acting within the development process. Likewise, constant productivity results from a system at

<table>
<thead>
<tr>
<th>Process Modeling</th>
<th>Symbol</th>
<th>Physics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Completion Count</td>
<td>$d$</td>
<td>Distance</td>
</tr>
<tr>
<td>Complexity</td>
<td>$m$</td>
<td>Mass</td>
</tr>
<tr>
<td>Progress</td>
<td>$md$</td>
<td>(no meaningful analogue)</td>
</tr>
<tr>
<td>Task Completion Rate</td>
<td>$v = \dot{d}$</td>
<td>Velocity</td>
</tr>
<tr>
<td>Productivity</td>
<td>$\rho = mv$</td>
<td>Momentum</td>
</tr>
<tr>
<td>Rate of Productivity Change</td>
<td>$F = \dot{\rho}$</td>
<td>Force</td>
</tr>
</tbody>
</table>
equilibrium. As productivity is the subject of interest, it is necessary to understand the opposing forces driving momentum in software development.

The net unbalanced force in a physical system is available to effect change. In the discussion of software development productivity, net force is given as

$$F_{\text{net}} = F_{\text{prod}} - F_{\text{res}}$$  \hspace{1cm} (3.8)

where $F_{\text{prod}}$ is the productive force generated by the workforce, and $F_{\text{res}}$ is the resistive force opposing it.

Resistive Force

Resistive forces can be modeled using a dashpot-like term [54]. A dashpot provides a velocity-dependent force opposing motion.

In software development, the dashpot reflects the difficulty of dealing with complex tasks rapidly. Let the resistive force be modeled as the dashpot-like damping equation

$$F_{\text{res}} = \frac{\xi}{\gamma} \rho$$  \hspace{1cm} (3.9)

where $\rho$ is the instantaneous momentum, $\xi \geq 1$ is a calibrated parameter capturing the resistive force per unit momentum experienced in the development activity, and $0 < \gamma \leq 1$, called “process quality”, captures the effect of personnel, management, and environmental attributes to magnify the resistive force. Low values of $\gamma$ are indicated by sources of distraction and interruption, lack of tools or adequate tool training [57], required multitasking [58], etc.

Combining Equation 3.8 and Equation 3.9, the equilibrium state is given by $F_{\text{net}} = 0 = F_{\text{prod}} - \frac{\xi}{\gamma} \rho$, which, solved for $\rho$, is

$$\rho = \frac{F_{\text{prod}} \gamma}{\xi}$$  \hspace{1cm} (3.10)
This reveals that momentum tracks a fraction of the magnitude of the productive force, and that steady-state productivity is maximized as $\gamma$ and $\xi$ approach 1. Before proceeding to the productive force, a brief aside is necessary into the psychological phenomenon of “flow”.

Flow and Productivity

The basic premise of flow [59] is that people find joy in accomplishing challenging tasks for which they have the required skills. As a result, people achieve maximal productivity on such tasks. However, a mismatch between personal capability and the level of challenge offered by a task—in either direction—can adversely affect productivity.

In the absence of a specific equation, the Gaussian function provides a ready match to this anecdotal description. By applying an affine map to the free variable, the Gaussian function can be centered over any empirical measure of challenge, and it can be scaled to fit an observed relationship between productivity and measured level of challenge. Figure 3.3 gives one such affine map. In the context of productivity in

![Scaled shifted Gaussian function](image)

Figure 3.3. Example Gaussian model of Csikszentmihalyi’s “flow” anecdote.
software development activities, the parameters of the affine map should be trained to historical data from the workforce assigned to the activity.

The following quotation is interesting when considering how flow may manifest in software development: “The nature of programmers is such that interesting work gets done at the expense of dull work, and documentation is dull work. Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost [6].” An interpretation of this quotation in the context of flow can be given as: “Certain tasks satisfy the optimal challenge level, and there is joy to be had in working on them. Alternatively, there are many tasks which do not, and these are avoided.” Note the potential impact of a work backlog: developers may choose the order of tasks they undertake. There is a potential to prioritize interesting tasks first, maintaining high productivity so long as the backlog is large enough to contain interesting tasks. As the backlog dwindles, however, the developer is forced to undertake the long remaining sequence of undesired tasks, diminishing productivity.

**Productive Force**

Productive force is the result of the effort of the workforce to overcome the resistive force and increase productivity, per the force-balance described in Equation 3.8. It is given as

\[ F_{\text{prod}} = e^{-\theta \beta} F_{\text{cap}} \omega \]  

(3.11)

where \( e \) is Euler’s number, \( \theta \) represents the backlog size, \( \omega \) is the workforce size, and \( \alpha \), \( \beta \), and \( F_{\text{cap}} \) are calibrated parameters. \( F_{\text{cap}} \) represents the maximum force generated by the average FTE of workforce. Its combination with \( \omega \) serves to estimate the total maximum force that can be delivered by the workforce. The Gaussian term can be seen to regulate this force as a function of the backlog size; this construction is a representation of the anecdotal description of “flow” addressed in Section 3.3.1.
Combining Equations 3.7–3.11 yields a differential equation describing productivity as a balance of forces,
\[ \dot{p} = e^{-(\alpha+\theta\beta)^2}F_{\text{cap}} - \frac{\xi}{\gamma}p \]  
(3.12)

### 3.3.2 State Model of Productivity

Let \( a_p, b_p, \) and \( x_p \) define the inputs, outputs, and state vector for a productivity model, respectively, as

\[
a_p = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \text{Backlog size} \\ \text{Workforce size} \\ \text{Process quality} \end{bmatrix}, \tag{3.13}
\]

\[
b_p = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} \text{Potential Production} \\ \text{Potential Productivity} \end{bmatrix}, \tag{3.14}
\]

\[
x_p = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \text{Potential Production} \\ \text{Potential Productivity} \end{bmatrix} \tag{3.15}
\]

with dynamics

\[
\dot{x}_p = f_p(x_p, a_p) = \begin{bmatrix} x_2 \\ e^{-(\alpha+a_1\beta)^2}F_{\text{cap}}a_2 - \frac{\xi}{a_3}x_2 \end{bmatrix}, \tag{3.16}
\]

\[
b_p = g_p(x_p, a_p) = Ix_p + 0a_p \tag{3.17}
\]

Here, “potential production” is the integral of potential productivity—a representation of the amount of work the team can have accomplished in the absence of activation constraints. Equation 3.16 gives the state evolution equation, where the instantaneous derivative of potential production is defined as the instantaneous productivity, and the derivative of potential productivity—i.e., net unbalanced force—takes the form of Equation 3.12. Equation 3.17 indicates that the output vector for the productivity state-model is a pass-through of the state vector.
3.4 Backlog

Recall that the backlog for a development activity is a collection of work to be completed. To track the backlog, a non-standard queuing model is defined. Like most queue models, it tracks a queue level as the difference between total input and total output. In the present case, the input and output are continuous, and are described by rates. Where this model is non-standard is that the out-flow rate is not directly specified. When the queue is empty, the out-flow rate targets the lesser of the in-flow rate and the commanded out-flow rate. The intuition is that the queue will provide as much of the commanded output as it can, directly transferring work from the in-flow to the out-flow, if necessary. Figure 3.4 gives the decision logic as a flowchart. To avoid the potential for discontinuities, the out-flow rate tracks the target rate rather than directly setting the out-flow rate.

3.4.1 Mathematical Development of Backlog Concepts

In this section, a saturation function is defined as a building block from which to build a smoothed approximation of the min function, and ultimately the decision-making capability described in Figure 3.4. The method of tracking that guides the actual out-flow rate is also developed.
A Saturation Function

Define

\[
\text{sat}(x) = \begin{cases} 
0 & \text{if } x < 0 \\
 x & \text{if } 0 \leq x < 1 \\
1 & \text{if } 1 \leq x 
\end{cases}
\]  

(3.18)

as a piecewise continuous saturation function. When used in a product, sat functions as a switch. Applying an affine map to the input parameter results in a translation of the saturation point, and a scaling of the interval over which the transition occurs. Such modifications allow one to position the switch at an arbitrary point in the input domain, while also governing the transition interval.

A sat-Based Min Function

Using sat, an approximation to the \( \min(\cdot, \cdot) \) function is constructed as

\[
\text{sm}(a, b) = a + \text{sat}(a - b)(b - a)
\]

(3.19)

Figure 3.5 plots this function with each argument fixed, in turn, to zero, revealing

\[\text{Asymmetry in sm}(x)\]

Figure 3.5. Asymmetry in arguments of sat-based min.
an asymmetry in the parameters. The asymmetry is introduced by the fact that sat performs its transition strictly after the threshold has been reached, rather than centering the transition at the threshold.

Determining the Target Out-Flow Rate

The preceding development of sat and sm provide the basic building blocks required to construct the decision logic for selecting the target out-flow rate. Let $r_c$ denote the commanded out-flow rate, and $r_i$ denote the in-flow rate. Then the lower decision in Figure 3.4 can be approximated by $\text{sm}(r_i, r_c)$. To address the upper decision, set the result of the lower decision as the default value, then apply sat to correct this value if the queue level is positive. Let $l_q$ denote the queue level; then the target out-flow rate can be expressed as

$$r_t \approx \text{sm}(r_i, r_c) + \text{sat}(l_q)(r_c - \text{sm}(r_i, r_c))$$ (3.20)

Tracking within a Continuous State Model

The method of tracking the target out-flow rate is formulated as an application of proportional control. Let $r_o$ give the out-flow rate of the queue; then $r_t - r_o$ gives the instantaneous tracking error. Let $\tau$ be a time constant such that $\tau^{-1}$ defines the desired proportional gain of the tracking controller. The resulting control law is given as

$$\dot{l}_q = \tau^{-1}(r_t - r_o)$$ (3.21)
3.4.2 State Model of a Backlog Queue

Let $a_q$, $b_q$, and $x_q$ define the inputs, outputs, and state vector for a queue model, respectively, as

\[ a_q = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \text{In-flow rate} \\ \text{Commanded out-flow rate} \end{bmatrix}, \quad (3.22) \]

\[ b_q = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} \text{Cumulative out-flow} \\ \text{Queue level} \\ \text{Out-flow rate} \end{bmatrix}, \quad (3.23) \]

\[ x_q = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \text{Cumulative out-flow} \\ \text{Queue level} \\ \text{Out-flow rate} \end{bmatrix} \quad (3.24) \]

with dynamics

\[ \dot{x}_q = f_q(x_q, a_q) = \begin{bmatrix} x_3 \\ a_1 - x_3 \\ \tau^{-1}(\text{sm}(a_1, a_2) + \text{sat}(x_2)(a_2 - \text{sm}(a_1, a_2)) - x_3) \end{bmatrix}, \quad (3.25) \]

\[ b_q = g_q(x_q, a_q) = I x_q + 0 a_q \quad (3.26) \]

where the third row of Equation 3.25 arises from the composition of Equation 3.20 and Equation 3.21. Thus, the derivative of the out-flow rate is defined to achieve reference tracking to the target out-flow rate. The middle row defines the accumulation of the queue by setting the derivative of the queue level to the difference between the in-flow and out-flow rates.

3.5 Activation Constraints

As discussed earlier, the presence of work in a backlog is not sufficient to indicate that the work may be performed. Instead, presence in a backlog queue merely
indicates that the work is known. Various intrinsic and extrinsic constraints may be imposed to declare criteria that must be satisfied before specific portions of work may be undertaken.

One such source of constraints is found in the intent to manage a software project under an incremental development lifecycle. Here, under several cycles of development, features are designed and potentially delayed until subsequent iterations. In such scenarios, the testing activities must be coordinated with feature implementation as well as test construction. It may occur that the features for a second planned internal release are being implemented concurrent with the execution of tests exercising a first internal release, or, depending on progress, feature and test implementation may proceed concurrently, with test execution proceeding where it can.

Consider the possible interleavings of the feature coding activity, test case coding activity, and the test execution activity with respect to two test cases, one for each of two features. The objective is to understand when each of the three development activities may process their respective tasks. For example, if the test team is ready to execute the test before either the feature to be tested is complete, or the test case is ready, they must wait. To analyze the remainder of the cases, define 5 events: completion of each feature, completion of each test case, and the event of reaching a point where the test team has the manpower available to execute a test case. Given 5 events, there are 5! = 120 possible orderings in which the events may occur. The factorial-order growth in the number of events to be considered motivates a need to express constraints directly and evaluate them dynamically, rather than attempting to predict their impact \textit{a priori}.

In Section 3.4.1 a rudimentary switch is developed. A simple approach to representing the activation constraints would seem to be possible through replicating the development activities for each partition of the work to be completed, and using the sat function to gate progress on the copies. This approach does not solve the issue of constraints, but instead translates the problem of representing constraints into a problem of representing the allocation of workforce over several copies of a develop-
Figure 3.6. Illustration of constraint implementation through thresholds.

ment activity. (However, it may be interesting in future work to consider methods for allocating the workforce under such a representation of constrained tasks.)

Instead, constraints are represented by establishing thresholds on the amount of progress that the activities may make. For example, to represent two phases of development for the feature coding activity, feature coding progress is initially constrained to a threshold based on the work underlying the first-phase features. Once the first phase is complete, and any other dependencies are satisfied, the threshold is raised to permit progress on the work underlying the second-phase tasks. Figure 3.6 gives an illustration of this example.

Thus, the amount of work that is ready to be processed by a development activity is given implicitly as the minimum of the queue level for the process, and the difference between the activity’s cumulative progress and the threshold on progress imposed by the activation constraints. As a result, the modeling effort must take the generation and maintenance of such thresholds as a primary concern.

Once the thresholds are established, a mechanism is required for enforcing them. Recall that the progress of an activity is represented as the cumulative out-flow from its associated queue. A second objective of this section is therefore to synthesize a
controller to regulate the commanded out-flow value supplied to an activity’s queue, in order to cause the cumulative out-flow of the queue to honor its threshold.

3.5.1 Mathematical Development of Activation Constraint Concepts

Before examining the methods of representing constraints, it is first necessary to characterize the nature of the constraints to be represented.

**Assumption 3.5.1** *It is assumed that the satisfaction of an individual constraint may be expressed in terms of the cumulative progress of the development activities.*

**Assumption 3.5.2** *It is assumed that threshold functions are time-differentiable everywhere.*

For example, in an incremental development effort, the criteria for beginning feature development on the second increment may be specified as the completion of 95% of the work for the first increment. Even constraints imposed by external independent parties can be fit into this framework if the negotiation with the external parties is modeled as a development activity that is constrained to start based on the progress of the traditional development activities.

The motivation underlying Assumption 3.5.1 is the eventual need to evaluate a derivative of the constraint function. By restricting the definition to functions described in terms of cumulative out-flow elements, any derivative terms required by the chain rule are trivially available as the corresponding out-flow rate members.
Threshold Functions

Consider the following piecewise constant step function, with transitions smoothed using cubic splines:

\[
\begin{align*}
    c(x) = \begin{cases} 
        0 & \text{if } x < 9 \\ 
        -140x^3 + 210x^2 & \text{if } 9 \leq x < 10 \\ 
        70 & \text{if } 10 \leq x < 29 \\ 
        260x^3 + 390x^2 + 70 & \text{if } 29 \leq x < 30 \\ 
        200 & \text{if } x \geq 30 
    \end{cases}
\end{align*}
\]  

(3.27)

such a function can be appropriate to represent the statically-known work to be executed by an activity, as a function of the progress of another activity. For example, assume that for two internal releases of a software project, the project plan anticipates 70 and 130 units of test execution work, respectively. The effort to produce the features-under-test is expected to require 10 units, and 20 units, respectively, of feature coding progress. Then \( c(\cdot) \) provides an appropriate representation of the threshold for the test execution activity.

Per the example, \( c(\cdot) \) only specifies the statically known work to be executed by the test execution activity. If there are additional sources of test execution work, e.g., tests re-executed upon feature correction work, then the definition of \( c(\cdot) \) would need to be extended to permit this dynamically generated work; perhaps by including an addend defined in the cumulative output of the feature correction activity.

By the continuously differentiable nature of cubic splines, and the chain rule, \( c(\cdot) \) is clearly time differentiable everywhere,

\[
\frac{d}{dt} c(x) = \frac{dc}{dx} \frac{dx}{dt}
\]
where $dx/dt$ is a queue out-flow rate, when $x$ is defined as a cumulative out-flow. In the general case, the threshold function takes a vector of arguments. Hence, a more general presentation is

$$\frac{d}{dt}c(x) = \nabla c \cdot \begin{bmatrix} \frac{d}{dt}x_1, & \frac{d}{dt}x_2, & \cdots, & \frac{d}{dt}x_n \end{bmatrix}^T$$  \hspace{1cm} (3.28)$$

Regulation of Progress

The purpose of the constraint functions is to provide a bound which can restrict the cumulative out-flow of a given queue—where cumulative out-flow represents progress. Given the preceding development of the queue, it is clear that actuation must be achieved in terms of the commanded out-flow rate. Thus a controller is to be specified which monitors progress relative to a threshold function, and adjusts the commanded out-flow rate so that the cumulative queue out-flow remains below the threshold.

There are a few cases to consider in achieving this goal. If the cumulative out-flow of the subject queue is below the threshold, then no regulation is required. When the subject queue’s cumulative out-flow is at the threshold, then another determination is needed. Assumption 3.5.2 requires that the threshold function is differentiable everywhere. If the derivative of the threshold function is greater than the commanded out-flow rate, then no regulation is required. Only when the subject queue’s cumulative out-flow is equal to the current value of the threshold, and the derivative of the threshold function is less than the commanded out-flow rate is regulation necessary. Figure 3.7 gives a flow-chart for this decision logic.

Let $p$ be a vector of cumulative out-flow members of the model output, and let $\dot{p}$ denote a vector of the corresponding out-flow rate members. Let $c(\cdot)$ be a time-differentiable threshold function for which $p$ is a suitable input. Then given a queue to be controlled, let $q_c$ denote its cumulative out-flow.
Figure 3.7. Decision logic for the activation constraint regulator.
Using a construction similar to Equation 3.20, the decision logic can be approximated as follows to produce a regulated commanded-rate as

\[ r'_c = \text{sm}(\nabla c|_p \cdot \dot{p}, r_c) + \text{sat}(c|_p - q_c)(r_c - \text{sm}(\nabla c|_p \cdot \dot{p}, r_c)) \]  

(3.29)

where \( r_c \) gives the a priori commanded rate, and the notation \( \alpha|\beta \) denotes \( \alpha \) evaluated using \( \beta \) as the value of its free variable. Here, the left-most term on the RHS gives (approximately) the minimum between the time-derivative of the threshold function, computed in-line, and the a priori commanded rate; it therefore implements the lower decision in the flowchart in Figure 3.7. The sat term is non-zero only when the cumulative out-flow of the queue is below the threshold; thus it implements the upper decision by either i) conditionally canceling the right term, leaving the left term to define the overall value, or ii) activating the right term, which cancels the left, and causes the a priori commanded rate to define the overall value.

### 3.5.2 Algebraic Model of an Activation Controller

Given a threshold function \( c(\cdot) \) defined in terms of development activity progress, define the inputs and outputs for an algebraic activation controller, respectively, as

\[
    a_c = \begin{bmatrix}
        a_1 \\
        a_2 \\
        a_3 \\
        a_4
    \end{bmatrix} = 
    \begin{bmatrix}
        \text{A priori commanded out-flow rate} \\
        \text{Dependent queue cumulative out-flow} \\
        \text{Progress measure(s) for computing } c(\cdot) \\
        \text{Productivity measure(s) for computing } \dot{c}(\cdot)
    \end{bmatrix}
\]

(3.30)

\[
    b_c = \begin{bmatrix}
        b_1
    \end{bmatrix} = 
    \begin{bmatrix}
        \text{Regulated commanded out-flow rate}
    \end{bmatrix}
\]

(3.31)

with output

\[
    b_c = g_c(a_c) = \text{sm}(\nabla c|_{a_3} \cdot a_4, a_1) + \text{sat}(c|_{a_3} - a_2)(a_1 - \text{sm}(\nabla c|_{a_3} \cdot a_4, a_1))
\]

(3.32)
Being algebraic, there is neither a state vector, nor a state-evolution function. That is, the entirety of the component is described as an algebraic combination of the inputs.

3.6 A Generic Development Activity Model

Having developed the basic components of a development model in the preceding sections, this section describes their interconnection into a generic model of a development activity. Figure 3.8 shows the interconnections among the submodels comprising the development activity model. The productivity model describes a capability to do work, which is regulated by the activation controller before being applied to consume items from the backlog queue. The queue level, representing the known outstanding work, feeds into the productivity model as a factor affecting productivity. The cumulative out-flow of the queue feeds back to the activation controller as a factor in

![Figure 3.8. Internal feedback within the activity model.](image_url)
determining whether to regulate the commanded out-flow, or pass through the full available productivity.

All inputs not participating in internal interconnections are exposed as inputs of the combined development activity model. All queue outputs are exposed as the outputs of the combined development activity model. Note that not all development activities require an activation controller, hence a simplified development activity model can be defined consisting of only the productivity model, directly connected to the backlog queue model.

3.7 Side-Effect Generation

Let “side effects” denote processes whose execution is unknown, unintended, or otherwise implicit within the activities of software development. The process of defect introduction within the feature coding activity is one such side-effect: it is certainly not intended, and is not explicitly staffed, yet it contributes a non-negligible sum to the overall project cost.

3.7.1 Defect Population

Defect introduction is the process of addition to the pool of defects present within a body of work product. The defect population is the result of defect introduction less the effect of defect removal. Toward understanding the dynamics of defect detection, it is necessary to gain an estimate of the defect population.

COQUALMO [60] is a parametric defect population model that can be calibrated to assume linear, superlinear, or sublinear defect introduction as a function of work-product size. COQUALMO takes the form

$$d_p = \sum_{j=1}^{3} A_j \text{Size}^B_j \prod_{i=1}^{21} (DL_{\text{driver}_{ij}})$$  \hspace{1cm} (3.33)
where \( d_p \) gives the estimated defect population size, and "\( j \) iterates over three categories of development artifact types, and \( i \) iterates over the impact of 21 different process factors describing the process used to create each type of development artifact" [60]. The exponents \( B_j \) determine whether COQUALMO is linear, super-linear, or sub-linear, and absent calibration, are set to 1, indicating linearity. The values \( DI_{\text{driver},ij} \) capture the impact, as a scale factor, of the \( i \)-th COQUALMO process factor, as it is assessed based on the \( j \)-th artifact type; they are selected from a table (which may be tailored to the organization) based on the results of a nominal-scale qualitative survey.

A Proposed Defect Introduction Side-Effect Process

Consider a dynamic evaluation of the parametric COQUALMO model: given an estimate of the work-product size as a function of time, the form of COQUALMO gives a method of estimating the growth of the defect population over time.

COQUALMO’s three development artifact types \{requirements, design, code\}, represent separate development activities, and are therefore treated separately, thus the outer summation can be eliminated. The \( DI_{\text{driver},ij} \)s can be combined into a composite process-impact factor \( D \), yielding the form

\[
d_p = A (\text{Size})^b D
\]  

(3.34)

Given Equation 3.34, an algebraic defect population model can be defined for a given development process by replicating Equation 3.34 for each artifact-production development activity.
Algebraic Model of Defect Population

Given inputs and outputs, respectively, as

\[
a_{dp} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \text{Total production} \\ \text{Total defects removed} \end{bmatrix}
\]  \hspace{1cm} (3.35)

\[
b_{dp} = \begin{bmatrix} b_1 \end{bmatrix} = \begin{bmatrix} \text{Estimated defects present} \end{bmatrix}
\]  \hspace{1cm} (3.36)

the outputs are produced as

\[
b_{dp} = g_{dp}(a_{dp}) = \begin{bmatrix} A(a_1)^b D - a_2 \end{bmatrix}
\]  \hspace{1cm} (3.37)

3.7.2 Defect Detection

Defect detection occurs when a test is executed over work-product containing a defect, and as a result of the defect, the test criteria are not satisfied. Note that this definition implies an interaction between the test criteria and the nature of a defect. This interaction has been described using an analogy with the Lotka-Volterra [61,62] predator-prey population dynamics model in the context of software testing [54].

Under this analogy, the defects present in a software target constitute a population of prey. The workforce assigned to testing activities constitute the predators. By enumerating the possible encounters and assigning a probability of occurrence, such a model predicts average defect detection rates in case studies [54].

The approach proposed here assumes that the rate of test execution in the software test process is proportional to the size of the testing workforce, and therefore modifies the analogy to identify test cases as the predators. As test cases are temporal, the new analogy uses the test execution rate as a representation of the number of predators hunting in an area per unit time. The prey are then the defects present at that time. This modification is motivated by the potential for the testing activities to be
prevented by activation constraints, even though a workforce remains allocated to the process. Clearly a workforce that is executing no tests will encounter no test failures.

The general form of a defect encounter model is given as

\[ r_f = \mu r_t p_d \]  

(3.38)

where \( r_f \) gives the test failure rate, \( \mu \) gives the decline rate—the rate by which potential encounters are actualized, \( r_t \) is the rate of test case execution, and \( p_d \) gives the estimated defect population. Thus the product \( r_tp_d \) gives the potential encounters, and the calibrated parameter \( \mu \) translates potential encounters into test failures.

It has been noted that \( \mu \) is likely time-varying [54], but that a piecewise constant treatment—through periodic recalibration—provides a suitable approximation.

Algebraic Model of Defect Detection

Given inputs and outputs, respectively, as

\[ a_{dd} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \text{Test case execution rate} \\ \text{Estimated defect population} \end{bmatrix} \]

(3.39)

\[ b_{dd} = \begin{bmatrix} b_1 \end{bmatrix} = \begin{bmatrix} \text{Test failure rate} \end{bmatrix} \]

(3.40)

the outputs are produced as

\[ b_{dd} = g_{dd}(a_{dd}) = \begin{bmatrix} \mu a_1 a_2 \end{bmatrix} \]

(3.41)

3.8 Special Case: Failure Analysis

Failure analysis is the activity responsible for performing the root-cause analysis on test failures. Due to the source of its in-flows, special handling is required. It is developed here as an example of how the generic development activity model can be tailored to accommodate process details.
The unique feature exhibited by the failure analysis activity is the uncertainty in
the destination of the corrective work it generates. Test failures arise from three main
sources:

1. feature defects,

2. test case defects, and

3. operator error.

The failure analysis activity requires extra book-keeping to predict the proportions
of its progress that will generate feature correction work and test correction work. To
accomplish this, it is augmented with a second queue and an algebraic component to
manage a proportional division of the commanded out-flow rate between the queues.
All other aspects of the generic activity model sum the queue outputs before use,
effectively treating them as a single queue. The dual queues are intended to consume
the output of test failures estimated by the defect detection models based on the pop-
ulation of feature defects, and test case defects, respectively, given the test execution
rate. Figure 3.9 illustrates this construction.

The algebraic proportional splitting component is defined as follows: Given inputs
and outputs, respectively, as

\[
a_{ps} = \begin{bmatrix}
  a_1 \\
  a_2 \\
  a_3 
\end{bmatrix} = \begin{bmatrix}
  \text{Reference 1} \\
  \text{Reference 2} \\
  \text{Quantity to be split}
\end{bmatrix}
\]  

(3.42)

\[
b_{ps} = \begin{bmatrix}
  b_1 \\
  b_2 
\end{bmatrix} = \begin{bmatrix}
  \text{Output 1} \\
  \text{Output 2}
\end{bmatrix}
\]  

(3.43)
Figure 3.9. Specialized activity model for failure analysis.
the outputs are produced as follows, where the piecewise form aids numerical stability for small values of the references.

\[ b_{ps} = g_{ps}(a_{ps}) = \begin{cases} \frac{(a_3a_1)}{(a_1 + a_2)} & \text{if } a_1 > a_2 \\ a_3 - \frac{(a_3a_1)}{(a_1 + a_2)} & \text{if } a_2 > a_1 \\ \frac{(a_3a_1)}{(a_1 + a_2)} & \text{if } a_1 = a_2 \end{cases} \]

\[(3.44)\]

### 3.9 Model Solution

Interconnected dynamical systems of the form described in Figure 3.1 and Equations 3.1–3.6 constitute an initial value problem when augmented with an initial state \(x_0\), and a specification of time-varying inputs \(u\). Listing 3.1 gives a modification of the improved Euler predictor-corrector method for approximating the solution of these initial value problems. Let \(h\) denote the timestep for the predictor-corrector. The first steps begin as usual: the derivative is taken from the current state, and an Euler step is taken to predict a future state \(\hat{x}_{t+h}\) (Steps 01–02). Step 03 introduces the modification: for an interconnected dynamical system, the subsystem input vector is required to evaluate the state evolution equation in order to compute the derivative. Hence, a value \(\hat{a}_{t+h}\) must be solved for, given the prediction \(\hat{x}_{t+h}\) and the expected
external system input \( u_{t+h} \). Once \( \hat{a}_{t+h} \) is acquired, Steps 04–06 compute and apply the correction before making a step in the usual way.

Step 03 requires an implicit solution in the general case of interconnected dynamical systems. The conditions on the existence of such a solution are given by the Implicit Function Theorem. However, for models built within the framework, Step 03 may have a direct solution. The following method gives a graph-theoretic test sufficient for the existence of a direct solution.

Create a vertex for each vector \( a_i, b_j, x_k, \) and \( \dot{x}_l \) described in Equations 3.1–3.6. For each subsystem input vector \( a_z \), connect an incoming edge from all subsystem output vectors \( b_w \) with elements contributing to any element of \( a_z \) through \( L_{11} \). For each subsystem output vector \( b_z \), connect an incoming edge from the subsystem state and subsystem input vectors \( x_z \) and \( a_z \) such that \( b_z = g_z(x_z, a_z) \). For each subsystem state derivative, \( \dot{x}_z \), connect an incoming edge from the subsystem state and subsystem input vectors \( x_z \) and \( a_z \) such that \( \dot{x}_z = f_z(x_z, a_z) \).

This graph gives the complete forward instantaneous data dependencies for a direct computation of the solution of Step 03 in Listing 3.1. Note that the edges relating \( \dot{x}_z \) to \( x_z \) are absent. As these edges represent integration, there is no amount of change to \( \dot{x}_z \) that can impact the value of \( x_z \), hence there is no instantaneous data dependency. If the resulting graph is acyclic, then there exists a direct solution. Proof: any topological ordering of the graph yields a valid direct computation ordering for all vectors. A topological ordering can be constructed if and only if the graph is acyclic. A direct computation ordering gives a method of computing \( \hat{a}_{t+h} \) given \( \hat{x}_{t+h} \) and \( u_{t+h} \). Figure 3.10 demonstrates this analysis at a coarser level, using categories of vector, showing a direct computation order for the implementation-phase process model illustrated in Figure 3.2.
Figure 3.10. Direct computation order for example implementation-phase process.
3.10 Simulation Study

To demonstrate the model behavior, the implementation-phase model of Figure 3.2 has been constructed. Parameter selection for the example is guided by the following scenario.

3.10.1 Example Scenario

Consider the implementation phase of a software development effort, structured according to Figure 3.2, in which the features to be completed are divided into two internal releases. Effort estimation for these two feature sets yields an initial estimate of 10 units of work for the first internal release, and an additional 20 units of work for the second release.

The test plan for the development effort calls for new test cases totaling 200 units of work. (Note that each development activity uses a different scale for work). The test case coding work is divided into two portions, of 70 and 130 work units, respectively, dependent on the internal feature coding releases.

The staffing plan for the effort calls for 29 FTEs of workforce, with 5 FTEs allocated to each activity, except for feature correction and test case correction; these activities receive only 2 FTEs of workforce. The development organization is reasonably efficient, thus a nominal process quality of 0.8 is assumed as a baseline when calibrating parameters.

Recall the piecewise constant function with transitions smoothed by cubic splines from Equation 3.27. The example implementation phase model includes re-work channels from the correction activities back to the new test execution activity. Thus the threshold function must account for this potential of unplanned rework. Let $c_{fcr}$ denote the total out-flow of rework flowing from the feature correction activity back to the test execution activity. Likewise, let $c_{tcr}$ denote the total out-flow of rework flowing to the new test execution activity from the test case correction activity. Finally, let $c_{fc}$ denote the cumulative out-flow of the feature coding activity. Then the complete
threshold function for the new test execution, and regression test execution, activities is given as \( T(c_{fc}, c_{fcr}, c_{tcr}) = c(c_{fc}) + c_{fcr} + c_{tcr} \).

The specification of the remaining parameters, and the specific interconnections in the \( L \) matrix are left for Appendix A. Without loss of generality, it is assumed that time units in the example are given in weeks.

3.10.2 Simulation Outcome

In the following discussion, approximately 80 input and output traces are plotted and meaningful correspondences are described. To reduce the number figures, a convention is enacted wherein dual y-axes are used. Magnitude plots are plotted against the left y-axis. Rate plots are plotted against the right y-axis. Note that productivity is a rate, and hence is plotted against the right axis. All other rates include the word “rate” in the plot legend.

Feature Coding

Figure 3.11 gives the inputs and outputs for the productivity model of the feature coding activity. A peak potential productivity is reached just prior to week 5, after a rapid ramp-up. All feature coding work is complete near week 13, yet it can be seen from the non-vanishing potential productivity that there remains a workforce assigned to feature coding. Figure 3.12 describes the behavior of the feature coding backlog during the simulation. The commanded out-flow rate, which represents the ability of the workforce to complete work, is driven directly by the productivity model and the out-flow rate tracks the commanded rate until approximately week 12. At this point, the queue nears an empty state, causing the out-flow to slacken.

Note that the cumulative out-flow—representing the total production of the feature coding activity levels off near week 14. Compared to the potential production from Figure 3.11 one sees that there is potential for production that is not being
Figure 3.11. I/O traces for feature coding productivity.

Figure 3.12. I/O traces for the feature coding backlog.
effectively utilized. This may indicate the need for additional tasking, schedule rear-
rangement, or staff reallocation.

Note also that the backlog size of the productivity model is precisely the queue-
level of the backlog model, in accordance with Figure 3.8.

Test Case Coding

The productivity model traces for test case coding, given in Figure 3.13, largely
follow the behavioral theme of the feature coding productivity model. A rapid ramp-
up leads to completion of test-case development after 26 weeks.

The test case coding queue (Figure 3.14), however, exhibits a new feature: a non-zero in-flow of additional work beginning just prior to week 4. Below, it is shown that this non-zero incoming stream of work results from corrective activities through a feedback channel capturing the intent that corrective actions may motivate a revision of the testing strategy.
Figure 3.14. I/O traces for the test case coding backlog.
New Test Execution

The new test case coding activity represents the execution of test cases of unknown quality—they have not been executed before—and hence are a higher source of test case defects than the other regression tests. In Figure 3.15, a more gradual growth of productivity is observed than for either of the preceding coding activities. This is to be expected as the backlog for new test case execution is dynamically generated, yielding a smaller backlog. With the present calibration, the work arrival rate is insufficient to drive the activity to its peak productivity.

There are a number of interesting characteristics found in the backlog traces of the new test execution activity, as plotted in Figure 3.16. First, note that the new test execution activity experiences a 4 week delay before work begins (as indicated by zero cumulative out-flow from from the backlog queue). During this period, the backlog simply accumulates until, at week 4, the commanded out-flow rate rises, and with it, the actual queue out-flow rate. The delayed start is due to the activation
Figure 3.16. I/O traces for the new test execution backlog.

Figure 3.17. Interaction of activation constraint and new test execution activity.
constraints on the new test execution queue—testing may not begin until the first release is available. The threshold is plotted in Figure 3.17; one may observe that the feature coding cumulative out-flow reaches the first-release value of 10 units of work at week 4. In response, the threshold for new test execution is raised to 70.

Next, one observes a sharp drop in the commanded rate near week 12. Note that this is precisely as the cumulative out-flow for new test execution reaches the threshold value of 70. Fortunately, the feature coding activity finishes the second internal release at this time, raising the threshold to 200, and thereby causing the activation controller to cease regulating the commanded rate.

The remainder of the simulation is uneventful until week 26, at which point new test execution activities cease.

The in-flow rate in Figure 3.16 has an unusual shape; Figure 3.18 gives a deconstruction of the input into its contributions from various development activities. As discussed earlier, it consists primarily of the testing work generated by the test

![New Test Execution In-Flow Composition](image)

Figure 3.18. Area graph of new test execution in-flow rate components.
case coding activity, but also includes components of re-work arriving from the two corrective activities.

Regression Testing

Figure 3.19 gives the traces for the regression test execution activity. A notable feature of the productivity plot is the flat segment of the backlog size during weeks 11–12. Productivity is non-zero so it must be the result of activation constraints. Figure 3.20 confirms the suspicion, showing that the commanded out-flow rate vanished during weeks 11–12. The regression test execution activity is configured with the same threshold function as the new-test execution activity, however the regression test activity exhibits a higher productivity. Thus, the threshold was reached sooner, causing the effect of the activation controller to become quite pronounced. As with the new test execution case, the activation constraint for the second release is satisfied near week 12, and the commanded out-flow rate rises in response.
Figure 3.20. I/O traces for the regression test execution backlog.
Defect Introduction and Defect Detection

Figure 3.21 plots the modeled number of defects introduced during the feature coding activity, and also plots the total progress of defect detection. Defect removal is plotted as a function of the work accomplished by the feature correction activity below. The difference between defect introduction and defect removal is plotted as the estimated feature defect population present in the work-product.

By the nature of the defect detection model, the defect population is asymptotic to zero under test execution in the absence of additional defect introduction.

In Figure 3.22, the test case defect introduction, detection, and removal traces are given, revealing a much lower detection rate for test case defects, ultimately leaving 22 test case defects in the test case work-product.
Failure Analysis

Figure 3.23 gives the rather unremarkable plot of the productivity model traces for the failure analysis activity. Note that the backlog size remains near zero, as all work arriving in the queue is quickly dispatched. This yields an interesting case in the backlog model: In Figure 3.24 one observes that the queue level is nearly zero at all times. Thus the out-flow rate closely tracks the in-flow rate, per the decision logic of Figure 3.4. Note also that the plots in Figure 3.24 plot the sum of the two queues used for the failure analysis activity (per the diagram in Figure 3.9).

The deviation in the in-flow rate, visible across weeks 11–12 is the direct impact of the cessation of regression testing due to the activation constraints described above.

As illustrated in Figure 3.9, the productivity and activation controller submodels of the failure analysis activity model combine the output of the dual failure analysis queues. Thus to provision the resulting out-flow rate appropriately between the queues, the proportional splitting component is applied. Figure 3.25 gives the source

Figure 3.22. Estimated test case defect introduction, detection, and removal.
Figure 3.23. I/O traces for failure analysis productivity.
rate, to be split, and the two reference rates that govern the split. The jitter present in the R1 trace is a consequence of the step size, small queue levels, and having a commanded out-flow rate that is larger than the in-flow rate. Figure 3.26 gives the outputs of the splitter, which sum to the source rate, and represent the same proportion as the references.

Feature Correction

The feature correction productivity and queue submodels (Figures 3.27–3.28) are quite similar in behavior to submodels discussed earlier, except in one aspect: while the queue is mostly empty, the in-flow rate crosses the commanded rate at a few points, causing the queue out-flow tracking logic to switch what it tracks. This is most notable from weeks 13–14, where the actual out-flow rate does not track the high peak of the in-flow rate, but instead transitions to the lower commanded out-flow rate. The tracking is imperfect due to the fact that the sm function underlying the

Figure 3.24. I/O traces for the failure analysis backlog.
Figure 3.25. Proportional splitting component inputs.

Figure 3.26. Proportional splitting component performance.
controller is approximate, experiencing its largest error when the difference separating the arguments is small.

Test Case Correction

The behavior evident in the test case correction traces (Figures 3.29–3.30) is largely covered in the preceding descriptions. They are included here for completeness. The small deviation in the in-flow rate of Figure 3.30 is an artifact of the brief impact of the activation controller on the new test execution activity.

3.11 Modeling Summary

In establishing Industrial Dynamics, Forrester expressed a desire to maintain a close correspondence between the concepts of the business domain and their representations in the models: "I hope to show, however, that mathematical notation

![Figure 3.27. I/O traces for feature correction productivity.](image-url)
Figure 3.28. I/O traces for the feature coding backlog.
can be kept close to the vocabulary of business; that each variable and constant in an equation has individual meaning to the practicing manager; that the successful manager of the future can understand, in fact will help originate, the relationships described by the equations” (p.9, section I.5) [33]. The preceding modeling work and its specific submodels, through their development in analogy and anecdote, take a middle ground between Forrester’s proposal of direct representation of business quantities and the empirical approach taken by the parametric models of Section 2.2.3.

It is anticipated that the productivity model, the defect introduction model, the defect detection model, and perhaps the entire specialized development activity for failure analysis may be augmented and enhanced in future work. So long as the resulting models continue to manage queues of the form described in Section 3.4.2 individual submodels may be independently replaced.

The model development presented in this chapter establishes a linearly interconnected system-of-subsystems [56] approach to modeling organization specific software development processes. This modeling method is suitable for the application of

Figure 3.29. I/O traces for test case correction productivity.
Figure 3.30. I/O traces for the test case coding backlog.
control-theoretic methods. This chapter has established that such models need not
sacrifice richness of behavior to accommodate the application of control.
CHAPTER 4. A METHOD OF CONTROL

The preceding chapter has established a framework for modeling the behavior of a software development effort based on prior project data. The present chapter leverages models built within the framework to construct a “decision support” capability as an application of automatic control.

Figure 4.1 illustrates the overall process of decision support described in this chapter. First, a determination is made as to whether the project progress is on-track with expectations. If not, an analysis is undertaken to understand whether external factors have caused a temporary disturbance, or whether something intrinsic in the

Figure 4.1. Process of applying decision support.
process has changed. The result of this analysis determines whether the predictive modeling capability must be re-calibrated. In either case, an updated forecast of future progress is made and is used to determine whether intervention is necessary. If necessary, the control computation is performed, as described below, to search for a set of model inputs that yields an improved forecast. Once a set of inputs that yields an acceptable forecast is found, the inputs are translated using management expertise into candidate process changes—to be applied at management discretion. If the process changes include updates to the plan, the process may be restarted.

4.1 Overview

Broadly, decision support refers to techniques, tools, or technologies that help an individual make decisions in support of achieving an objective. In the present context, the objectives are defined in terms of desired changes to be achieved in the behavior of a software development effort, such as increasing productivity rates to desired levels, or reducing the overall project duration. Management decision support tools may require expert input from project managers, for example, to estimate subjective parameters that are difficult to quantify by measuring work product. In this way, decision support tools are complementary rather than competitive with project management.

In the following sections, a decision support mechanism is constructed to address two classes of management objectives:

1. Objectives that can be defined in terms of deviation, over time, of modeled process outputs relative to desired model outputs, represented as a reference function; and

2. Objectives that can be defined in terms of minimization of certain modeled process outputs.

In the literature of control theory, these classes of objectives align with the concepts of “tracking”, and “regulation”, respectively. As developed below, the decision support
mechanism is constructed as a “controller” (in the sense of control theory), where the “plant” to be controlled is a model within the framework of Chapter 3. The controller acts on the model through control variables—additive components that augment a subset of the model inputs. The following assumption underlies this construction:

**Assumption 4.1.1** For small variation in the inputs to the model, the resulting output trajectory will represent the effect of an equivalent change in the development process.

Here, equivalence is defined in terms of representation: A change in a development process is equivalent to a change in model’s variable inputs precisely when the change in model inputs represents the change in the development process. In this control setup, the control loop is closed by the project manager, who observes the input changes selected by the controller (the “control values”), determines a candidate set of equivalent process changes, and at his/her option, assimilates the suggested changes into his/her decision making.

The controller is realized through the construction and solution of a constrained non-linear optimization problem. Here, the management objective is formalized as a “performance index”—a functional that assigns a score to tuples consisting of a set of candidate control values, and the resulting model output trajectories. Thus, the performance index establishes a partial order on the desirability of such tuples with respect to the management objective. The performance index guides an automated search through the space of valid control values, as defined by element-wise “control bounds”, with the best-scoring set of candidate control values encountered during the search being identified as the final control values.

Recall from Chapter 1 that the overarching objective of this work is to substantiate the proposal that quantitative feedback can help project management to mitigate the impact of inaccuracy and uncertainty in a priori cost prediction by providing guidance on the magnitude and timing of corrective process interventions. Project duration is a significant driver of project cost, as is seen by considering salaries and benefits of the workforce, depreciation of computing equipment, and the cost of facilities.
In the following sections, two management objectives are defined with respect to project duration. Performance indices are then developed to represent these objectives as examples of the general method for translating a management objectives into a performance index. Evaluation of the controllers constructed from these performance indices is the subject of the subsequent chapter.

4.2 Definitions and Convention

Chapter 3 presents a modeling framework rather than a specific model. As a result, the controller that implements decision support must be constructed without knowledge of the model specifics, relying only upon the constraints imposed by the framework. In this way, the decision support method remains applicable to all models that can be constructed in the modeling framework. Preserving this generality, however, incurs a notational cost, as described below.

4.2.1 Model Equations, Inputs, and Outputs

Recall that the models of Chapter 3 define the dynamics of a software development effort as a system of parameterized simultaneous equations describing the evolution of a vector of state variables. These equations define an initial value problem when augmented with (1) an initial state, (2) values of the variable inputs over time, and (3) concrete values for the fixed parameters. The solution of this initial value problem is a vector-valued function describing the model outputs as a function of time. When solved over a future time horizon, this provides a method for estimating the future behavior of the represented software development process.

Let

\[ y = F(x_0, u, v, t_0, t_f) \]  

(4.1)

denote a solution \( y \) to the initial value problem \( F \) defined by the model equations in conjunction with an initial state \( x_0 \), a set of variable inputs \( u \), and a set of fixed
parameters \( v \), defined over the time range \([t_0, t_f]\), where \( t_0 \) represents the time at which \( x_0 \) is the current state. Note that \( y \) is vector-valued and time varying. Equation 4.1 assumes an implicit discretization, and thus \( y \) is a sampling, expressed as a partial function defined on a fine time grid. The solution \( y \) may also be referenced as “the model output trajectory” or simply “the model output.”

4.2.2 Components of Interest in the Model Output

At times, the construction of the decision support method requires the use of certain elements of \( y \) (as they evolve over time) while ignoring others. In order to preserve independence from model specifics, a notion of “extraction matrices” is used.

**Definition 4.2.1** An “extraction matrix” is a sparse \( m \times n \) matrix \( E \), \( m \leq n \), such that left-multiplication of a column vector \( z \) by \( E \) results in a dense column vector containing only the elements of interest from \( z \) in a known order.

An extraction matrix can be formed from an identity matrix by deleting rows corresponding elements of the vector to be omitted. To simplify the presentation, the order of extracted elements is ignored, and it is assumed that other related vectors are constructed in the same order.

For example, to compute a weighted sum of the queue level members of a model output trajectory \( y \) as a function of time, one would define an extraction matrix \( E_{\text{levels}} \) such that the product \( E_{\text{levels}}y \) extracts the queue level elements from the model solution \( y \). An inner product of the result with a vector of weights produces the desired weighted sum, \( W_{\text{levels}} \cdot E_{\text{levels}}y \). Here it is clear that the order of the weights in \( W_{\text{levels}} \) must correspond to the order of queue level traces extracted by \( E_{\text{levels}} \) to apply the weights correctly, thus the discussion is omitted. Note that this notation using extraction matrices has facilitated the construction of a weighted sum of queue levels without specifying the number of queues, or their respective locations within the model output.
4.3 Objective 1: Minimize the Impact of Schedule Deviation

“Schedule deviation” denotes the difference between the actual progress of a development effort, and that anticipated by a planned schedule for the project. One component of the impact of schedule deviation, in cases where actual progress lags behind planned progress, can be directly understood in the costs associated with additional project duration. There may be additional components introduced through inefficiencies in the use of shared resources as a result of schedule deviation, through the need to acquire duplicate resources, or indeed through direct contract penalties for late completion, as examples. As a result, a project manager may desire to take actions to bring the actual project progress in line with the planned schedule. However, there may be costs associated with action; thus in general a balance must be struck between the cost of action and the cost of inaction.

To develop the performance index capturing this management objective, a more formal definition of schedule is required.

4.3.1 A Notion of Schedule

Informally, a software development schedule refers to the notion that “work completion” is to be accomplished such that various milestones are met no later than certain points in time. In practice, project schedules are expressed in various levels of detail. For example, GANTT charts express a schedule as a directed graph with vertices representing individual tasks and edges describing interdependencies. The edges, in conjunction with data about specific resource-assignments and effort-estimates, serve to constrain a layout of the vertices over a time axis, which can be used to produce a schedule expressed as specific expected tasks performed by specific personnel over time. In contrast, earned-value management can be performed with less detail: earned value management only requires a cumulative sum over time of an organization-specific scalar representation of “value” (a measurement of work completed in terms of value added to the deliverables) to represent the schedule.
A decision support tool that seeks to serve a broad audience must utilize a notion of schedule capable of abstracting a large proportion of schedule-specification methods. The following definition is used throughout the remainder of this work.

**Definition 4.3.1** Let “schedule” denote a vector-valued partial function, defined over a set of time values including at minimum the end-of-schedule time, $t_f$, with the following semantics:

1. Each element of the instantaneous output gives a real-valued organization-specific measure of cumulative work completed; and

2. Each dimension in the output vector represents a category of work.

Consider, for example, a schedule $s : D \rightarrow \mathbb{R}^n$ defined over a non-empty set of time values $D \subseteq \mathbb{R}$, $t_f \in D$. Then $s(t) = [w_1(t), w_2(t), \ldots]^T$, where $w_i : D \rightarrow \mathbb{R}$ gives the measure of cumulative work expected to be completed in work-category $i$ for all $t \in D$.

Definition 4.3.1 provides sufficient flexibility to describe both a detailed GANTT-type schedule, as well as a scalar earned-value-oriented schedule. Consider, by construction, that each GANTT task may be assigned to its own category of work, with $D$ containing the planned end-times for each task. For earned value oriented schedules, it is sufficient to note that Definition 4.3.1 describes a collection of earned value metrics, implementing the minimal example given above when the number of work categories is reduced to unity.

**4.3.2 Aligning the Schedule with the Development Activities**

The performance index development undertaken below assumes that the *a priori* project schedule is expressed in categories of work that align with the development activities in the corresponding process model. In cases where the *a priori* schedule is specified in terms of other work categories, average data about the fraction of total work in each category that is contributed by each development activity can be used to
compute a new representation of the schedule aligned with the modeled development activities. Consider

\[ \hat{w}_a(t) = \sum_{i \in \text{Cat}} p_{i,a} w_i(t) \]

(4.2)

where \( \hat{w}_a \) gives the component of a new schedule \( \hat{s} \) associated with development activity \( a \), \( \text{Cat} \) gives the categories of work present in the original schedule \( s \), \( p_{i,a} \) gives the constant fraction of work in original-schedule category \( i \) that is typically contributed by development activity \( a \), and \( w_i \) gives the component of the original schedule \( s \) associated with work category \( i \). Note that \( w_a(\cdot) \) is defined at the same time positions as \( \hat{w}_a(\cdot) \).

If a given schedule \( s \) does not contain sufficient detail for such a sum over average fractional splits to give a meaningful resulting schedule \( \hat{s} \), then data on the relative timing of contributions to the original-schedule work categories by each development activity can be used, if available, to construct an alternate schedule \( \hat{s} \) from \( \hat{s} \). To formalize the notion of relative timing of work, consider a histogram of historical work item completion events for a development activity, normalized to unit area, defined over bins of “fraction of schedule duration elapsed”. Let “work density function” denote the resulting piecewise-constant function. Then define \( \hat{s} \) as a schedule whose components, aligned to the development activities, are given by

\[ \hat{w}_a(t) = \hat{w}_a(t_f) \int_{t_0}^{t} w_{df,a} \left( \frac{\tau - t_0}{t_f - t_0} \right) d\tau \]

(4.3)

where \( w_{df,a} \) gives the work density function for development activity \( a \). This equation can be understood to compute the cumulative relative progress expected by time \( t \) from the work density function for each development activity \( a \), and then render it non-relative by multiplying the cumulative amount of work expected at schedule-end for the development activity as per the schedule \( \hat{s} \), as defined in Equation 4.2. Note that \( \hat{w}_a \) is a total function over the range \([t_0, t_f]\).

Development organizations may have sufficient historical metrics to estimate \( p_{i,a} \) and \( w_{df,a}(\cdot) \); alternatively the literature provides a few generic models. For example,
the CORADMO extension of the COCOMO II model [63] provides a method for determining the expected break-down of COCOMO II’s scalar total-effort result into per-activity effort figures (for common sets of development activities). Also, the U.S. Air Force, in studying acquisition success, notes that the proposed schedules for projects that are ultimately successful follow specific curves of progress over cost and cost over time [21] which are similar in definition to $\text{wdf}_a$—and hence give general data on the level of effort expected to be applied to various development activities over time.

4.3.3 Determining the Status of Schedule Adherence

Software development encompasses diverse types of work. Depending on the project and the organization, types of work may span the range from the creative processes of translating requirements to design to the rote tasks of collecting and analyzing test results to record test failures, or validating satisfaction of traceability matrices from formal specifications through final testing.

There exist well-known issues in obtaining accurate completion data for creative tasks in software development. One example is the so-called “90% complete syndrome” [19, 31] succinctly illustrated in the historical quotation, “during about 80 percent of the actual time span of a project, the supervisors reported the job to be about 90 percent completed” [24]. Such issues aside, it is the responsibility of a software project manager to monitor progress relative to the project schedule, and to take corrective action when it is determined that the actual progress will deviate significantly from the schedule.

Chapter 2 provides a brief survey of the methods that have been developed to aid managers in making the determination of whether corrective actions are necessary. Such determination is outside the scope of the present work, except to note that, for methods that detect the occurrence of schedule deviation through the use of trending, there may be an analogous approach that makes use of the predictive
modeling capability developed in Chapter 3 to implement the logic responsible for identifying the trend. This assumes that an extrapolation into the future of a model calibrated to recent data from an ongoing development effort can provide a rudimentary representation of the expected future trend. Pursuit of this idea is left to future work.

4.3.4 Scoring Corrective Actions for Schedule Adherence

As described in the introduction, a performance index is to be constructed to impose a partial order over candidate corrective actions. The problem of compensating for schedule deviation can be understood as a search, through the space of possible process changes, for a process change that will cause the future evolution of the development effort to re-align with the *a priori* schedule. This search is complicated by the fact that process changes cannot be achieved without an application of effort, which will incur some cost and some risk. A suitable performance index must therefore account for the balance between the cost incurred by schedule deviation, and the cost incurred by the application of corrective action.

Let \( p = E_{\text{cof}}y \) be a predicted schedule, where \( E_{\text{cof}} \) is a matrix that extracts the cumulative out-flow elements of a model output vector \( y \). Definition 4.3.1 clearly applies as \( p \) predicts real-valued measures of cumulative work completed, and each element of \( p \) represents a category of work to be completed.

Let \( r \) be an *a priori* planned schedule, with work-categories adjusted to align with the modeled development activities per the process above, if necessary. By Definition 4.3.1, \( r \) is a partial function. Let \( \bar{r} \) be the partial function constructed from \( r \) by defining \( \bar{r} \) at all points present in the domain of \( p \) via interpolation. Then \( \epsilon = \bar{r} - p \) gives the predicted schedule deviation as a vector-valued continuous function of time.

As \( \epsilon \) is vector valued, the instantaneous amount of schedule deviation may be captured using a norm. It is well-known that any symmetric positive definite matrix
$Q$ induces an inner product, $\langle a, b \rangle_Q = a^T Q b$. Let $\| \cdot \|_Q$ denote a 2-norm defined using such an inner product. Then, $\| \epsilon \|_Q^2 = \epsilon^T Q \epsilon = \| Q^{1/2} \epsilon \|_2^2$ can be interpreted as a square of the standard Euclidean norm of $\epsilon$, as weighted by the principal square root of $Q$. By selecting the elements of $Q^{1/2}$ (which is also symmetric positive definite [64]—permitting a diagonal matrix as a special case), one may assign specific weights to the elements of $\epsilon$ according to the particular categories of work the elements represent in the schedule. Recall that the elements of $\epsilon$ represent schedule deviation as quantities of work for the various development activities. The weights in $Q^{1/2}$ must consist of two components: (1) a scale factor to convert the elements of $\epsilon$ to a uniform representation of cost, so as to render the summation within the norm well-formed; and (2) a scale factor to weight the relative importance of deviation in specific elements of $\epsilon$. Note that only $Q$ is used in the performance index; the digression on $Q^{1/2}$ is intended to aid in constructing an appropriate $Q$.

Likewise, for $u$ representing the nominal process inputs leading to the predicted schedule $p$, let $\Delta u$ represent an adjustment to the model inputs, then $\| u + \Delta u \|_R^2 = (u + \Delta u)^T R (u + \Delta u)$ gives the square of a 2-seminorm of the adjusted model inputs, weighted by the symmetric positive-semidefinite matrix $R$. Analogous to the case above, this construction assesses a weighted quadratic cost as a function of the adjusted model inputs. Unlike $Q$, $R$ is symmetric positive semidefinite, therefore the interpretation as a squared standard Euclidean norm of a weighted vector must define the weighting matrix $L^T$ in terms of a Cholesky decomposition $R = LL^T$ [64], where $L^T$ is upper triangular—also permitting a diagonal matrix as a special case. As with the previous case, the elements of $L^T$ must be selected to scale the elements of the adjusted model input vector to a uniform representation of cost, and then apply an additional scaling based on the relative cost of increasing the specific type of input.

A method for scoring a candidate corrective action can then be constructed as

$$J_{sa}(\Delta u) = \int_{t_0}^{t_f} \| \epsilon \|_Q^2 dt + \int_{t_0}^{t_f} \| u + \Delta u \|_R^2 dt$$

(4.4)
subject to the relation \( \epsilon = \bar{r} - E_{cof} F(x_0, u + \Delta u, v, t_0, t_f) \). Under a Riemann interpretation, the left integral of \( J_{sa} \) can be understood to compute a sum of squared error goodness-of-fit metric between the planned schedule, and a predicted schedule derived using the candidate change to the model inputs, \( \Delta u \). The right term integrates the square of a seminorm of the instantaneous candidate input vector. It therefore computes a quadratic cost metric based on the resources expended over time. In the controls literature, \( J_{sa} \) is a reference-tracking performance index, and \( \bar{r} \) serves as the reference to track.

Note that \( \epsilon \) is a partial function. The integral in Equation 4.4 is well-formed because \( \epsilon \) represents the difference between two samplings on the same time grid, and therefore is itself a sampling. As a collection of discrete samples from an underlying continuous function, the integral over \( \epsilon \) is understood as the application of a numerical quadrature.

4.3.5 Extensions of Schedule Tracking

In the preceding construction, the references to be tracked were restricted to schedule elements per Definition 4.3.1. The definition of \( \epsilon \) can be trivially extended to include references for any component of the model output. Consider the following alternative rendering of \( \epsilon \):

\[
\epsilon = E_{ext}(\hat{r} - F(x_0, u + \Delta u, v, t_0, t_f))
\]

(4.5)

where \( \hat{r} \) gives an extended set of references, and \( E_{ext} \) is the extraction matrix that extracts the model output elements corresponding to those in \( \hat{r} \). Under this extension, \( \hat{r} \) can specify desired trajectories for any element of the model output, potentially including queue levels, out-flow rates, estimated defect totals, etc.
4.4 Objective 2: Minimize the Overall Project Duration

In cases where coordination with external entities is not required, a project manager may wish to reduce overall project duration rather than maximize adherence to an *a priori* schedule. That is, the project manager desires to determine a resource allocation strategy that balances the cost of resources expended against a cost assessed on time-to-completion. The key difference between Objective 1 and Objective 2 is that the shape of the model trajectories that lead to the shortest project duration are not known in advance. Thus one cannot use a reference-tracking controller as was developed for Objective 1.

For models in the form of Chapter 3, recognizing project completion is trivial: a project is complete when all queues are empty. Quantifying how much work remainins when a project is partially complete is a far more difficult task given the side-effect processes, potentially cyclic flows, and activation constraints.

4.4.1 A Notion of Project Completion

The modeling framework developed in Chapter 3 represents work items awaiting completion in terms of continuous queue levels. The queue levels are instantaneous representations of the amount of work waiting to be processed by the specific development activities. Recall, however, that the model also defines consumer/producer relations, and therefore the model implies additional work, captured only in the knowledge that when some activities consume their work items, they generate new work items for their downstream activities.

To apprehend a representation of “total work remaining”, consider that the outflow rate of a development activity’s queue represents the rate of actual work completion for the activity; therefore the integral of the out-flow rate, or the cumulative out-flow for a queue, represents the amount of total work completed to date for a given activity. Given a model output trajectory that represents project completion; i.e., there is a time $t_c$, $t_0 \leq t_c \leq t_f$ beyond which all queue level elements of the
model output vanish, the total work remaining (at a time \( t \)) for a modeled development effort can be computed as a sum of weighted difference between the cumulative out-flows for all activities as predicted at \( t_c \), and the cumulative out-flows from all activities as modeled at \( t \).

Note, however, that there are nonlinearities in the model that cause the total work processed during a development effort to vary under changes to the inputs. Also, depending on the model inputs, there may be no point \( t_c \) in the model output trajectory satisfying the definition above; i.e., some sets of inputs leave work incomplete. As a trivial example, consider the output trajectory of a model with zero workforce applied—the total work remaining is clearly not a function of cumulative queue out-flow in this case. To provide a general representation, the method for determining the total work to be completed during a development effort must be amended.

Let \( W_{cof} \) be a row vector of weights used to normalize the amount of work represented by work items for the various development activities. Then an upper bound for the weighted total work to be completed during the development effort is given by the scalar value

\[
  w_{\text{total}} = \max_{\Delta u} \left[ W_{cof} E_{cof} F(x_0, u + \Delta u, v, t_0, t_f)|_{t_f} \right]
\]

subject to project-specific bounds on \( \Delta u \). Here the model initial value problem \( F \) is solved over the interval \([t_0, t_f]\), yielding a vector-valued partial function, which is evaluated at \( t_f \) to capture the final sample of the model output vector. An extraction matrix is applied to isolate the cumulative out-flow members which are combined into a weighted sum representing the total work completed. It is this weighted sum that the maximization operation considers, yielding ultimately an upper bound on the amount of work that will be completed under any model inputs.

Practically, \( w_{\text{total}} \) can be estimated by evaluating Equation 4.6 for a large sample of the permitted space of input changes, then computing the sample mean \( \bar{\mu} \) and an appropriate sample standard deviation \( \hat{\sigma} \) for the observed distribution of the sample;
i.e., \( \hat{w}_{total} = \hat{\mu} + 3\hat{\sigma} \geq w_{total} \), where \( 3\hat{\sigma} \) reflects the common statistical definition for an outlier. Note that since \( w_{total} \) is intended as an upper bound, the factor 3 can be increased to accommodate an arbitrary confidence bound.

### 4.4.2 Scoring Corrective Actions for Duration Minimization

Let \( w_{cur}(\tau) = W_{cof} E_{cof} F(x_0, u + \Delta u, v, t_0, \tau) \) denote the weighted sum of cumulative work items completed at time \( \tau \). An upper bound on the weighted sum of total work remaining can then be computed as the scalar value \( \hat{w}_{rem} = \hat{w}_{total} - \hat{w}_{cur} \). Then, \( J_{md} \) gives a performance index that assesses a quadratic penalty on the amount of work remaining, and assesses a quadratic cost on the weighted total input magnitude:

\[
J_{md}(\Delta u) = \int_{t_0}^{t_f} \hat{w}_{rem}^T \hat{w}_{rem} \, dt + \int_{t_0}^{t_f} \| u + \Delta u \|^2_R \, dt
\]  

\[
= \hat{w}_{total}^T \hat{w}_{total} (t_f - t_0) - 2\hat{w}_{total} \int_{t_0}^{t_f} w_{cur} \, dt + \int_{t_0}^{t_f} w_{cur}^T w_{cur} \, dt + \int_{t_0}^{t_f} \| u + \Delta u \|^2_R \, dt
\]

subject to \( w_{cur}(t) = W_{cof} E_{cof} F(x_0, u + \Delta u, v, t_0, t) \), and the empirical derivation of \( \hat{w}_{total} \) from \( x_0 \), \( u \), and \( v \), given the bounds on \( \Delta u \).

### 4.4.3 Accommodating Fixed Workforce

In contract-work, schedule, workforce size, and budget figures are intimately intertwined in the production of a bid. If a contract is awarded in response to a bid, it is assumed that a pool of workforce resources of the planned size will be assigned to the project. Thus a project manager making use of the proposed decision support method may need to specify overall workforce-size constraints, while ideally leaving the controller free redistribute the fixed workforce over the activities arbitrarily.
Let $E_{wf}$ be a matrix that extracts the workforce size elements from the model input vector as a dense column vector, and let $w_{\text{fixed}}$ describe the scalar total fixed equivalent FTEs of workforce to be allocated during the development effort.

A fixed workforce can be accommodated in the controller construction by augmenting the chosen performance index with the additional equality constraint

$$w_{\text{fixed}} - [1, 1, \ldots] \cdot E_{wf}(u + \Delta u) = 0$$

(4.8)

where the inner-product with a vector of ones produces a summation of the extracted elements. It may also be necessary to loosen the control bounds for the workforce size variables of the activities, to give the controller an appropriate level of freedom.

### 4.5 Defining the Control Law

In Control Theory, a “control law” is a function that defines the input changes to be applied to a plant as a function of its behavior (as measured from its output). Given a performance index $J$, augmented with a set of constraints $C$, the control law is expressed concisely as

$$\Delta u^* = \arg\min_{\Delta u} J(\Delta u)$$

(4.9)

subject to the constraints $C$.

Evaluation of this expression is nontrivial, as the performance indices are not convex in general. Section 4.7 investigates the constrained performance index form from a perspective of convexity in $\Delta u$. As described in Section 4.8, a numerical minimization can be performed to evaluate the control law. The analysis in Section 4.7 indicates the need for the numerical minimization to be repeated for a sample over the space of permitted input changes, taking the minimum of the results obtained as sub-optimal solution.
4.6 A Justification of Quadratic Costs

The motivation for assessing quadratic costs in the performance indices may not be obvious; for example, a weighted linear cost might seem a more natural choice. The motivation differs for the two terms of the performance indices developed above.

The use of a quadratic cost under the left integral introduces an analogy with the residual sum of squares model-fit criterion. As the left term assesses a penalty on deviation of the trajectory of work completion from an \textit{a priori} predicted trajectory, the search for $\Delta u^*$ can be interpreted as a model fitting exercise, hence a standard model-fit criterion is appropriate.

The quadratic term under the right integral reflects assumed diseconomies of scale in process change. Consider that the average available workforce FTEs may be raised through temporary or periodic overtime and weekend work, however beyond a threshold such increases may be difficult to incentivize. Likewise, there may be a collection of process inefficiencies that can be readily remedied through the application of technology, training in the available technology, or elimination of deleterious practices [57] but beyond a point, process quality enhancements may require substantial cultural or organizational change.

It may be desirable, in future work, to include a third term in the performance indices to assess a quadratic cost on the magnitude of input change. The utility of this third term would be to encourage the use of small changes, and thereby increase confidence in the accuracy of the controlled model as a predictor of the controlled process, per Assumption 4.1.1.

4.7 Sub-Optimality

To understand the optimality of control suggestions obtained by minimizing a given performance index, one may consider the convexity of the constrained minimization problem in the minimization variables.
Consider $J_{sa}$ (Equation 4.4): It is well known that sums of convex functions preserve convexity, thus one may separately consider the addends that comprise $J_{sa}$. First consider $\int_{t_0}^{t_f} \| u + \Delta u \|^2_R \, dt$. Again, sums of convex functions preserve convexity, so it is sufficient (under a Riemann interpretation) to consider only the convexity of $\| u + \Delta u \|^2_R$. It is also well-known that convexity is preserved over affine transformation of the domain of a convex function; thus it is sufficient to consider the convexity of $\| \Delta u \|^2_R = \Delta u^T R \Delta u$. Given that $R$ is positive semidefinite, this quadratic form is convex in $\Delta u$.

Next, consider $\int_{t_0}^{t_f} \| \epsilon \|^2_Q \, dt = \int_{t_0}^{t_f} \| p - \bar{r} \|^2_Q \, dt$. Here, $\bar{r}$ is independent of $\Delta u$, but $p = E_{cof}F(x_0, u + \Delta u, v, t_0, t_f)$. Because $p$ is a function of time resulting as a solution of an initial value problem defined in terms of $\Delta u$, one may not strip away the integral and norm when considering convexity as with the preceding case. One must instead show that the scalar integral result is a convex function over variation in $\Delta u$ as it contributes to the definition of the initial value problem.

Consider instead the convexity of $\int_{t_0}^{t_f} \| p - \bar{r} \|^2_Q \, dt$ in $p$ (rather than in $\Delta u$); the term is clearly convex in $p$ by an argument similar to that above (though with $Q$ being symmetric positive definite). However, this result is irrelevant to the convexity of $J_{sa}$ in $\Delta u$ without the following constraint: $p$ must be restricted to trajectories satisfying $p = E_{cof}F(x_0, u + \Delta u, v, t_0, t_f)$. This constraint modifies the domain over which one considers the convexity of the term, potentially resulting in a non-convex domain, and thereby endangering the finding of convexity.

Thus, a convexity result for $J_{sa}$ may be established for convex subsets of the space of acceptable control inputs over which the resulting trajectories $p$ satisfy the criteria for defining a convex domain. It is unclear whether such subsets exist, or whether their number is finite, and thus this observation cannot be used to define a practical strategy for obtaining a global minimization. It does, however, motivate a sub-optimal strategy of taking the infimum over numerical minimizations begun from a set of randomly selected acceptable control inputs, as a coarse approximation of
taking the infimum over representatives taken from a partitioning of the acceptable control inputs satisfying the above convexity criterion.

4.8 Controller Implementation

The controller is implemented through numerical solution of the control law. A number of steps are required to translate the abstract presentation of the control law developed above, into a concrete implementation suitable for solution by a numerical toolkit such as GNU Octave or MATLAB\(^1\).

4.8.1 Specification of Control Variables and Bounds

Given a model within the framework of Chapter 3 one must identify the model inputs that can be modified by the controller, and augment them with control variables. Control variables should only be specified for model inputs within the project manager’s sphere of influence—i.e., from the set of model inputs that represent aspects of the development process he/she has the power (via corporate authority, political influence, and resource control) to modify through real-world changes.

Bounds must be established on the range of values permitted for each control variable. Some bounds will be constrained by the quantities that the inputs represent; for example, a negative workforce size has no translation to the real-world process. Other bounds will be constrained by the project manager’s ability to secure resource; such as a restriction that the manager can bring no more than three additional workforce members onto the project. In the end, the bounds define the search space for the numerical minimization.

The control variables are arranged sparsely in $\Delta u$, such that $u + \Delta u$ correctly augments the selected input elements by the corresponding control values. The upper and lower bounds are specified as correspondingly ordered sparse vectors; i.e., $\Delta u_{\text{lower}} \leq \Delta u \leq \Delta u_{\text{upper}}$.

\(^1\)MATLAB is a trademark of The MathWorks.
4.8.2 Selection of Weights

The selection of $Q$ and $R$ is partly analytical, partly based on experience, and partly based in trial and error. To establish a baseline of uniform “cost” for each category of work in the schedule (aligned to development activities, see Section 4.3.2) consider the method of constructing “total work.” Setting the elements of $Q$ such that

$$Q^{\frac{1}{2}}E_{\text{conf}}F(x_0, u + \Delta u, v, t_0, t_f)|_{t_f} = I$$

gives a weight matrix $Q$ that penalizes a notion of “percentage of maximum possible deviation” uniformly across activities. That is, if no work is performed, each activity is penalized equally. Such an initial baseline can be produced using any left pseudoinverse. From this baseline a manager can adjust the matrices to impose his own priorities, perhaps by composing the baseline matrix with a new matrix comprising weights on the diagonal. Finally, based on controller performance, a manager may choose to tweak $Q$ in a trial and error mode to fine tune the predicted results of applying control.

The construction of $R$ is analogous, with a starting point of $L^T(u + \Delta u_{\text{upper}}) = I$ where $R = LL^T$.

4.8.3 Establishment of Control Partitions

In the simplest case, a controller determines a set of input changes to be applied immediately. However, it can be advantageous to allow a controller to plan a sequence of changes to be applied at specific times. Optionally, the estimated project duration may be partitioned into “control partitions” - segments of the project over which the control variables are held constant.

When using multiple control partitions, the task of selecting controllable inputs and establishing bounds is performed independently for each partition. It is clear that independent control variables must be used to represent the designation of a model input as controllable in multiple control partitions. As in the preceding section, upper and lower bounds must be set for each independent control variable.
The use of control partitions can be seen to extend the definition of control variables—effectively augmenting each with a temporal scope. The non-partitioned case is trivially equivalent to the single-partition case, and hence $\Delta u$ may always be interpreted to contain partition-specific control variables. Likewise, $\Delta u_{\text{lower}}$ and $\Delta u_{\text{upper}}$ may be interpreted as partition-specific bounds or as global bounds.

4.8.4 Formulation of Equality and Inequality Constraints

The preceding development has introduced two types of constraints: those representing the objectives of the project manager, e.g., Equation 4.8, and those constraining terms within the performance index to honor the model equations, typically expressed as a relationship between a performance index variable and $\Delta u$ via $F$.

The constrained minimization implementations, “sqp” in Octave and “fmincon” in MATLAB, require constraints to be specified as functions of the variables over which minimization will occur, hence the latter type of constraint cannot be directly represented to the numerical minimizer. Appendix B describes two methods of resolving this issue. The present work adopts the method described in Appendix B.1.

4.8.5 Evaluation of the Control Law

To evaluate the control law, a constrained numerical minimization routine is executed supplying (1) a function that evaluates the performance index, yielding a scalar “score”; (2) functions to evaluate the equality and inequality constraint equations, also yielding scalar representations of satisfaction; (3) vectors specifying permitted bounds on the control variables; and (4) a vector of control values to serve as the starting point for the numerical minimization.

Recall from Section 4.7 that any given execution of the constrained minimization may terminate at a local minimum, and that computing the set of starting points that guarantee a global solution is impractical. It is therefore advised that the mini-
mization be executed from multiple starting points randomly distributed around an a priori best estimate of the control values that will lead to a global minimum.

4.8.6 Interpretation of Results

The final step in applying the controller is for project management personnel to translate the resulting control values into candidate real-world process changes. By choosing to implement the candidate changes in the real-world development process, the project manager closes the control loop.

4.9 Decision Support/Control Summary

A sub-optimal method is defined that is capable of deriving input changes that optimize model behavior with respect to quantitative performance indices defined in the model output, for any model defined per the framework of Chapter 3. Given a representative model calibrated to recent project data, such output serves to guide management personnel in selecting appropriate process changes to apply within the actual software development effort. In the preceding steps, the controller provides the project manager with guidance on the magnitude of changes to be made, the nature (in terms of which development activity/activities) of required changes, and, when using multiple control partitions, information on the timing of process changes.

Examples of reference tracking and regulating performance indices are defined, which serve as the basis for scenario-based evaluation of the control method in the subsequent chapter.
CHAPTER 5. CONTROLLER EVALUATION

The preceding chapters have defined a modeling and control framework. This chapter addresses the performance of controllers constructed by the process given in Chapter 4. The method of evaluation is demonstration through scenario development and application. The example software development project of Section 3.10.1 is used throughout the control examples.

Figure 5.1. Observed deviation in feature coding progress, scenario 1.
5.1 Scenario 1: Mis-prediction of Required Level of Effort

5.1.1 Scenario Narrative

Consider a scenario as depicted in Figure 5.1, in which a software development effort has been proceeding in reasonable accord with the planned schedule. A model-based prediction at week $t_n = 2$ indicates a likelihood of acceptable future schedule adherence, and so the development effort is permitted to continue without intervention. Near week $t_d = 5$, it is detected that the actual progress in the feature coding task has deviated from the predicted progress and the planned schedule. An analysis of the development effort subsequently rules out external influences and indicates that a mis-prediction of the amount of required effort underlies the schedule deviation. As a result, the predictive model is re-calibrated to recent project data, yielding a new predicted trajectory for feature coding progress—one that indicates a significant schedule overrun—given in Figure 5.2.

Figure 5.2. Re-calibrated estimate of feature coding progress, scenario 1.
Figure 5.3. Impact of delayed internal release on new test execution, scenario 1.

Figure 5.4. Impact of delayed internal release on failure analysis, scenario 1.
The schedule overrun in the feature coding activity significantly delays the second internal release, which impacts down-stream activities such as new test case execution (Figure 5.3), and failure analysis (Figure 5.4).

Based on this quantitative information the project manager decides that action is required. It is decided that a corrective action should be applied to return the feature coding activity to its planned trajectory by week $t_c = 10$. Further, it is determined that process improvements may exist for the feature coding activity representing an increase of up to 10% in the process quality parameter. Lastly, up to 3 FTE of additional appropriately trained workforce is available at need.

5.1.2 Analysis and Controller Construction

Following the steps of Section 4.8, one first identifies and bounds the control variables. The narrative clearly indicates that control variables are to augment the workforce size, and the process quality model inputs. The bounds on the control variable attached to process quality are also directly given as $-0.1 \leq \gamma_a \leq 0.1; a \in \text{Act}$ where Act is the set of development activities represented in the model.

The constraint bounding the number of permitted workforce additions does not dictate bounds for any particular development activity. In the absence of a scenario-driven bound, the workforce size control bounds are set conservatively as $-\omega_a \leq \Delta \omega_a \leq 32 - \omega_a; a \in \text{Act}$, where Act is the set of development activities, and 32 is 3 greater than the initial FTEs allocated across the activities.

The next step is to assign the weight matrices $Q$ and $R$. For this scenario, in the absence of management expertise, $Q$ and $R$ are constructed as diagonal matrices associating weights with development activities per Table 5.1 and Table 5.2. Of course, these weights represent a weighted prioritization of the different development activities.
### Table 5.1
Scenario 1 weights penalizing schedule deviation.

<table>
<thead>
<tr>
<th>Activity</th>
<th>$\epsilon$-Weight in $Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Coding</td>
<td>6</td>
</tr>
<tr>
<td>Test Case Coding</td>
<td>1</td>
</tr>
<tr>
<td>New Test Execution</td>
<td>1</td>
</tr>
<tr>
<td>Regression Test Execution</td>
<td>1</td>
</tr>
<tr>
<td>Failure Analysis</td>
<td>3</td>
</tr>
<tr>
<td>Feature Correction</td>
<td>2</td>
</tr>
<tr>
<td>Test Case Correction</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table 5.2
Scenario 1 weights penalizing total input magnitude.

<table>
<thead>
<tr>
<th>Activity</th>
<th>$\omega$-Weight in $R$</th>
<th>$\gamma$-Weight in $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Coding</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Test Case Coding</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>New Test Execution</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Regression Test Execution</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Failure Analysis</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Feature Correction</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Test Case Correction</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
Following this, the project manager selects the number of control (time) partitions over which the optimization is to take place. Resource reallocation is desired to achieve two “corrective” long-term behaviors:

1. A long-term increase in the productive output of the feature coding activity to realign behavior of the activity with the rate of completion anticipated in the *a priori* planned schedule, and

2. An additional short-term increase in the productive output of the feature coding activity required to outpace the rate of completion anticipated in the planned schedule, and thereby compensate for the amount of non-completion that triggered detection of the schedule adherence issue.

This observation motivates at least two control partitions: one to allow the controller to apply control for the objective of reaching the originally planned work completion by $t_c$, and a second for maintaining the higher rate of production required to keep pace with the planned schedule. The optimal position for the boundary between the control partitions is not immediately obvious. For this reason, three partitions will be used; two for the reasons just discussed, and a third as a transition period in case the control variable bounds are too restrictive to permit the controller to reach the planned level of feature coding completion by $t_c$. Select the following as the boundaries of the control partitions: $\langle t_d, t_c, t_c+\text{trans}, t_f \rangle$, where “trans” defines the transition period, and is arbitrarily set to 2 weeks.

Next, one addresses the issue of equality/inequality constraints. The narrative imposes a constraint that no more than 3 additional FTEs of workforce may be added. The following constraint fits the form of an inequality constraint, and implements the restriction

$$c_{\text{ineq}}(\Delta u) = 3 - (1, \ldots, 1)E_{w^f} \Delta u > 0 \quad (5.1)$$

where $E_{w^f}$ is the matrix that extracts the workforce-size members of the system-input vector (and therefore the vector of control values as well).
Using the approach of Appendix B.1, the controller is constructed as a constrained minimization with this single inequality constraint. Recall that $J_{sa}$ is constructed abstractly as a reference-tracking performance index for schedule adherence. To apply it toward the present scenario, $E_{cof}$ must be realized to extract the cumulative out-flow members of the underlying model’s output vector.

The last step in realizing $J_{sa}$ is to define the reference trajectories. Let $\bar{w}_{fc}$ denote the planned progress for the feature coding activity (interpolated if given as a partial function). Let $p_{fc}$ denote the current prediction of progress for the feature coding activity. Then a reference trajectory for the feature coding activity can be defined over the interval $[t_d, t_f]$ as the piecewise function

$$\tilde{r} = \begin{cases} 
\bar{w}_{fc} & \text{if } t > t_c \\
\frac{t - t_d}{t_c - t_d} \bar{w}_{fc} + \left(1 - \frac{t - t_d}{t_c - t_d}\right) p_{fc} & \text{if } t \leq t_c 
\end{cases}$$

(5.2)

constructed as a linear combination of the present trajectory $p_{fc}$ and the desired trajectory $\bar{w}_{fc}$ where the blend is adjusted as a function of the position within the interval $[t_d, t_c]$. Thus, Equation 5.2 captures the objective of returning the total progress of the feature coding activity to the original plan by week $t_c$. The resulting trajectory for the feature coding activity is given in Figure 5.5. The reference trajectories for the other development activities are constructed in an analogous manner.

5.1.3 Control Results

Figure 5.6 gives the result of applying the control results $\Delta u^*$ within the simulated process. As can be seen, the controller is largely successful in compensating for the schedule deviation by the target week $t_c = 10$. At week 10, there is a slight reduction in productivity visible as the controller uses the middle control partition to produce overall model inputs that track the initial predicted schedule. Interestingly, at the end of the second control partition, $t_{c+\text{thresh}} = 12$, the controller chooses to remove resources, incurring a slight schedule deviation penalty. Because the final control
Figure 5.5. Computed reference trajectory, scenario 1.

Figure 5.6. Result of control, feature coding activity, scenario 1.
partition spans weeks 12–30, the resource changes made at week 12 are long-lasting, and therefore incur cost over a long duration. First, note that the deviation present during the final control partition is relatively small. Because the cumulative outflow of the feature-coding task will saturate to the reference trajectory (i.e., there is a fixed number of planned features), over-allocation of workforce to the feature coding activity will result in wasted capability and a heightened penalty assessed on the total input size for feature coding. The controller identifies control values that result in a careful balance between over-allocating the workforce size and incurring a schedule-deviation penalty integrated over a long period.

Perhaps more importantly, the approximate recovery of performance to the initial schedule means that the internal releases are delivered near their original target dates. Hence the catastrophic cascade of delays through the schedule, predicted in Figures 5.3–5.4, should not occur. Figure 5.7 gives plots, for the other development activities, of the initial prediction, the updated prediction at week $t_d = 5$, the reference trajectory selected for the control step, and the outcome from applying the resulting control values. As can be seen, the cascade of delays is largely avoided.

In Figure 5.7(a), it can be seen that the test case coding activity operates independently from the feature coding activity. That is, the schedule deviation in feature coding had no direct impact on test case coding. The small discrepancy observed between the nominal and controlled traces is due to the controller slightly reducing the overall workforce assigned to test case coding. The new test execution activity (Figure 5.7(b)) is successfully driven back to the expected schedule, which it tracks subsequently. The same is true for the regression test execution activity in Figure 5.7(c).

The controlled trace of the failure analysis activity, Figure 5.7(d), shows a successful ascent to the reference trajectory by the target time $t_c = 10$. In the subsequent period, application of the control values yields a lower initial productivity. Perhaps surprisingly, the failure analysis task completes more overall work with the control values applied than without. Also, observe from Figures 5.7(e)–5.7(f) that the cor-
Figure 5.7. Result of control, remaining development activities, scenario 1.
rective activities achieve less overall progress with the control values applied. These phenomena are related; to see how, consider the defect detection model’s inherent dependence on the defect population. A reduction in progress by the corrective activities leads to the persistence of a larger defect population in the work products produced. This larger defect population raises the test failure rate, providing more work for the failure analysis activity.

As mentioned above, the corrective activities show reduced overall progress after application of the control values. Figures 5.9(e)–5.9(f) show that the controlled workforce sizes allocated to the corrective activities represent a reduction in workforce. This decision by the controller is due to the magnitudes in the corrective-activity traces being relatively small; that is, a schedule deviation of 100% is still an error-magnitude less than 10. Contrast this with a 10% schedule deviation in regression test execution, which has a magnitude of 20. The weights in the $Q$ matrix that penalize schedule-deviation for the corrective activities are not large enough, given the differences in magnitude involved, to cause the schedule deviation in the corrective activities to dominate the performance index computation and thereby garner the attention of the controller. Translated to the real-world process, such weights in $Q$ would indicate a strong management preference for completing the initial construction and test activities of the subject project, even if that means working around numerous defects. For example, a feature-complete alpha- or beta-quality product for use in controlled sales demonstrations.

Controlled Workforce Size

The controlled inputs, $u + \Delta u^*$ are given alongside the original nominal inputs in Figures 5.8–5.9. In Figure 5.8 one may observe that the controller has decreased the workforce size of some activities, and increased others. This represents a reallocation of personnel among the development activities, done to achieve a workforce increase of approximately 6 FTEs for the feature coding activity over the first control partition,
Figure 5.8. Nominal and controlled workforce size, feature coding, scenario 1.
while upholding the constraint that the total workforce size may not increase by more than three. During the third control partition, the controller selects a negative control value for the feature coding workforce size variable. This represents a removal of all but 1.5 FTEs from the activity, effectively leaving a skeleton crew to finish out the small amount of remaining work and rework.

Figure 5.9. Nominal and controlled workforce size, remaining activities, scenario 1.
Figure 5.9 gives the controlled workforce size for the remaining activities. Figure 5.9(a) shows that the workforce of the test case coding activity is reduced by a fractional amount. This modification is no doubt made in the effort to provide personnel to the feature coding activity.

Next, the new test execution activity in Figure 5.9(b) has nearly 0.7 FTEs added during the first control partition, yet experiences a reduction of the same magnitude during the subsequent two control partitions. This is partly expected because, once back on track, the activity tracks its former schedule—it is only the work for the feature coding task that is underestimated in this scenario. Figure 5.9(c) shows a similar control result.

The controlled traces shown in Figures 5.9(d)–5.9(f) reveal a reduction from the nominal workforce sizes. These reduction are motivated by cost savings. Due to the relatively small backlog sizes combined with relatively small weights in $Q$, the cost imposed by $R$ dominates these activities, and thus the controller permits them to deviate a few units of work away from their respective trajectories. This explanation is supported by Figure 5.10, where it can be seen that the controller has available personnel, but opts not to deploy them.

Figure 5.10 gives the total workforce size applied across all activities in the nominal and controlled cases. Here three distinct decisions can be seen with respect to the control partitions: In the first control partition, $t \in [5, 10]$, the feature coding development activity is allocated the largest proportion of the workforce. Here the controller is augmenting feature coding workforce to increase the productivity, which in turn increases cumulative out-flow. The increased cumulative out-flow directly reduces the tracking error, and thereby reduces the penalty term for schedule non-adherence in the performance index. The total workforce increase approaches the limit of 3 additional FTEs specified through the inequality constraint. From these facts, one may conclude that, for the first control partition, the controller is attempting to reduce the impact penalties based on schedule deviation.
Figure 5.10. Total nominal and controlled workforce size, all activities, scenario 1.
During the middle control partition, \( t \in (10, 12] \), the total controlled workforce size returns to the nominal level of 29, however the controlled workforce size assigned to feature coding activity remains near 11. Consequently, the remainder of the development activities remain at below-nominal controlled workforce sizes. During this control partition, there does not seem to be a dominant driver of costs within the performance index.

In the final control partition, \( t \in (12, 30] \), the controlled workforce size variables represent an across the board reduction. Here, it is clear that the total input size penalty, weighted by \( R \) is dominating the performance index computation. Hence, the controller’s actions can be interpreted as enacting cost-savings targeted toward a performance index dominated by the cost of the input magnitude.
Controlled Process Quality

Figure 5.11 gives the controlled and nominal values of the process quality input to the feature coding activity. As expected, the controlled process quality level reflects the same control strategy as for the workforce size variable: high levels for the first two control partitions, followed by a lowered level in the third control partition to avoid penalties via $R$.

In Figure 5.12, the controlled and nominal traces for process quality are given for the other activities. In Figures 5.12(a)–5.12(c), the control values for the process quality variables are specified as the maximal values permitted given the upper control bound. In contrast, Figures 5.12(d)–5.12(f) illustrate controller decisions to reduce process quality (with varied timing) to the minimal values permitted given the lower control bound.

Recall the steady-state equation for the productivity model from Equation 3.10. Combined with Equation 3.11 it gives

$$\rho = \frac{e^{-(\alpha + \theta \beta)^2 F_{\text{cap}}}}{\xi \omega \gamma}$$

which reveals that fractional (multiplicative) augmentations to $\omega$ and $\gamma$ have equivalent effect on the steady-state productivity.

Consider the failure analysis activity, for which $\omega_{fa} = 5$ and $\gamma_{fa} = 0.8$. The weights for these variables in the $R$ matrix are $R_\omega = 2$ and $R_\gamma = 5$, respectively. Let $\delta$ be an arbitrary scaling factor. Then for candidate fractional augmentations having identical effect on productivity, the question of interest is whether $R_\omega \omega_{fa}^2 \delta^2$ is less than $R_\gamma \gamma_{fa}^2 \delta^2$.

For the failure-analysis activity, $2 \cdot 25 \cdot \delta^2 > 5 \cdot 0.64 \cdot \delta^2$, and hence $\gamma$ is preferred. For the corrective activities, the nominal workforce size $\omega_{fcr} = \omega_{tcr} = 2$, which still yields: $2 \cdot 4 \cdot \delta^2 > 5 \cdot 0.64 \cdot \delta^2$, and hence the controller should prefer to use $\gamma$.

On average, comparing Figure 5.9 to Figure 5.12 one sees that whenever the controller applies an elevated workforce, the process quality control value is also
Figure 5.12. Nominal and controlled process quality, remaining activities, scenario 1.
Figure 5.13. Total nominal and controlled process quality, all activities, scenario 1.

... elevated to the control limit. Process quality is reduced at the same times as workforce size, indicating that the controller is intentionally reducing the productivity.

The case of feature correction warrants further analysis: in particular, the control value for its process quality input temporarily takes a non-extreme value of approximately 0.815. This co-occurs with a reduction of the workforce size to approximately 1.65; performing the preceding analysis indicates that while \(2 \cdot 1.65^2 \cdot \delta^2 > 5 \cdot 0.815^2 \cdot \delta^2\), the difference is far smaller than the preceding cases (5.45 > 3.32), indicating that an increase to either side could tip the balance; hence the controller uses both together. On average, per Figure 5.13, the controller applies a 7–9 percent increase in process quality over the first two control partitions, returning to near-nominal average levels in the third partition.
5.1.4 Details of Scenario Construction

Starting from the nominal model configuration used in Chapter 3, a solution is obtained for the interval \([t_0, t_f]\). This provides the “prediction at time \(t_n\)”.

Then, the calibrated average force-contribution of the workforce \((F_{\text{cap}}\omega)\) for the feature coding activity is reduced by 50 percent. The adjusted model is solved over the interval \([t_n, t_d]\), using as initial state the state \(x_n\) resulting from the earlier solution. Concatenating the initial solution over \([t_0, t_n]\) with this latter solution over the interval \([t_n, t_d]\) provides the trajectories labeled “actual progress”. This construction is consistent with a calibration of the model parameters during a period of work on tasks well-estimated in of terms work, followed by a transition to tasks for which the work estimates are too low. The result is a management-visible impression of initial schedule adherence, followed by a clear deviation, as observed in Figure 5.1. Continuing this solution over the interval \([t_d, t_f]\) generates the “prediction at time \(t_d\)”.

5.2 Scenario 2: Allocating a Fixed Workforce

5.2.1 Scenario Narrative

In many development environments, a project manager has a fixed team from which to manage all activities of a project. Consider a scenario in which a new project is being planned for the purpose of submitting a competitive bid: Engineering estimates for the technical work have been produced, and the task interdependencies have been captured. At this point a project manager is tasked with determining a budget and schedule for the project given his fixed team comprising 29 FTEs of workforce.

His prior experience leads him to estimate bounds on the process quality values he can achieve as \([0.6, 0.92]\). Also prior experience leads him to prefer individual teams of fewer than 15 members.
5.2.2 Analysis and Controller Construction

The nominal input for this scenario is fixed as a zero vector—that is, no work will be accomplished without controller intervention. A control variable is defined for each element of the external model inputs. In this way, the controller will directly select the total external input to the model (as the control variables are an additive augmentation of the zero vector).

It should be noted that this scenario could also have been built with non-zero nominal inputs. The present construction offers an interesting alternative in which the only management input is made through the weight matrices $Q$ and $R$, and the control bounds.

Per the steps of Section 4.8, all control variables are bounded as $0.6 \leq \Delta \gamma_a \leq 0.92$ and $0 \leq \Delta \omega_a \leq 15$, for all development activities $a$, based on the managers experience and preference.

Next, to facilitate more aggressive optimization, the project is divided into 5 control windows. To place the initial boundaries of the control windows, a nominal guess of the model parameters is made, and the resulting duration is partitioned into 5 segments. Here, the nominal duration of 30 time units is divided into windows of width 6. As the objective of the narrative is to produce a competitive bid, the performance index $J_{md}$ (from Chapter 4) is selected for its ability to operate without an a priori schedule. The model equations are incorporated by the method of Appendix B.1, leaving the only the fixed size of the workforce to be implemented as an equality constraint, per the method of Section 4.4.3:

$$c_{eq}(\Delta u) = 29 - [1, 1, \ldots] \cdot E_{wf}(u + \Delta u) = 0 \quad (5.4)$$

where 29 gives the available FTEs of workforce to be utilized, $E_{wf}$ is the matrix that extracts the workforce size members from the vector of external inputs, and the inner product with the vector of ones indicates a summation of the extracted elements.
Table 5.3
Weights relating the relative cost of work between activities.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Weight in ( W_{cof} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Coding</td>
<td>3</td>
</tr>
<tr>
<td>Test Case Coding</td>
<td>3</td>
</tr>
<tr>
<td>New Test Execution</td>
<td>2</td>
</tr>
<tr>
<td>Regression Test Execution</td>
<td>2</td>
</tr>
<tr>
<td>Failure Analysis</td>
<td>1</td>
</tr>
<tr>
<td>Feature Correction</td>
<td>1</td>
</tr>
<tr>
<td>Test Case Correction</td>
<td>1</td>
</tr>
</tbody>
</table>
The weights normalizing the effort represented by work from the various development activities are set per Table 5.3. $R$ is a diagonal matrix identical to that specified in Table 5.2. Lastly, $\hat{w}_{\text{total}}$ is estimated at 1578.4 given the nominal model calibration.

5.2.3 Control Results

Figure 5.14 gives the simulation results that arise from supplying $u + \Delta u^*$ as the model input; that is, augmenting the nominal inputs with the control values. These “controlled” plots are presented alongside results generated from the naïve nominal workforce allocation, and uniform process quality used for other nominal simulations in preceding sections.

The reader is cautioned against placing too much value on measurement of the improvement over the nominal schedule achieved by the controller: The nominal inputs do not represent a carefully expert-estimated resource allocation plan. Instead, the nominal inputs should be understood as a reasonable guess at an appropriate a priori static allocation of resources. Any comparison with the controller results should therefore be understood in a context of similarity of effort: Given the same relative level of effort, one could guess and evaluate a few candidate sets of input parameters, or one could construct and evaluate the controller.

A Systemic Predilection for Slow Saturation

In Figure 5.14 one observes a theme in which development activities are driven quickly to near-saturation, but then slowly climb to the saturation point. This is observed clearly in Figures 5.14(a)–5.14(c). It is desirable to understand the contributors to this behavior.

In the case of Figure 5.14(a), the model of the feature coding process includes no channels for dynamic production of in-flow, and there are no activation constraints. Thus the slow saturation of cumulative out-flow is not attributable to sustained slow inflow, nor due to mis-regulation of the modeled productivity. The slow upward climb
Figure 5.14. Predictions, controller-selected versus \textit{a priori} allocation, scenario 2.
of cumulative out-flow is due entirely to the resource reduction in the second control partition (weeks 6–12) and its influence on the productivity model.

Figure 5.14(b) shows that the test case coding activity model is also subject to a slow increase in cumulative out-flow from week 6 through week 19; the observed rise in cumulative out-flow, approximately 10 units of work over the 6 weeks of the third control partition gives a slope of 1.67 units of work per week. This is consistent with completing the new in-flowing work, as given in Figure 5.15(a), however it does not explain the slow rise over the second control partition. As the test case coding process is not governed by activation constraints, the only remaining explanation for the slow saturation over the second control partition is reduced productivity due to application of the control values.

In Figure 5.14(c) a rise of 20 work-units over weeks 13–18 is observed, which is a slope of 4 units of work per week. Thus the slow saturation of the new test execution activity is well-explained by the in-flow rate over the third control partition, as shown in Figure 5.15(b), assuming the same rapid completion of dynamically arriving work holds as for the statically known work.

Thus, the behavior of the controller is sometimes shown to incur the observed rapid progress followed by slow saturation through resource starvation, and other times through dynamic in-flow. This is sufficient to reject the hypothesis that the
slow-saturation behavior is systemic in the controller design, and instead supports that it is a feature of the specific weights and/or selection of control partitions.

Comparison

Subject to the caveat in the introduction, a comparison between the nominal and controlled trajectories yields a few interesting points: Firstly, the feature coding task completes at that same time for both the nominal and controlled inputs. The controlled trajectory reaches the 90 percent complete mark far sooner, yet languishes in a simulated “90 % complete syndrome” for half of the active duration of the activity. The effect of this more-rapid rise, however, is to reach more quickly the threshold satisfying the activation constraint for work associated with the second modeled internal release.

Test case coding reaches initial completion near week 14, versus week 26 in the nominal trajectory; a reduction of 45%. New test execution completes in week 18, rather than week 27 (a 30% reduction). Regression test execution completes in week 14, rather than week 27 (a 50% reduction). Failure analysis completes in week 19, rather than week 27 (a 30% reduction). Finally, the corrective activities exhibit a reduction in schedule of approximately 30%.

Controlled Workforce Size

Figure 5.16 gives the controlled workforce size values over the five control partitions. From the plots, one observes that the controller first allocates personnel to the coding tasks (Figures 5.16(a)–5.16(b)) by borrowing the personnel from every activity except for regression test execution. Next, the controller scales down the allocation to the coding tasks in order to ramp up the testing tasks (Figures 5.16(c)–5.16(d)). After that, the controller focuses on the failure analysis activity and feature correction (Figures 5.16(e)–5.16(f)), then finally onto test case correction (Figure 5.16(g)).
Figure 5.16. Nominal and controlled workforce size, scenario 2.
Figure 5.17. Incomplete failure analysis work, scenario 2.
Note that the majority of work is completed by the time the final control partition is entered in the controlled simulation; however the controller does not set the workforce size to a uniform value over the activities, as one might expect, in an attempt to minimize the quadratic cost weighted by $R$. Surprisingly, it controls the workforce size value for the failure analysis activity to zero in direct contrast with this strategy. Looking into the cause of this reveals the empirically derived $w_{\text{total}}$ fails to provide an upper bound on the total work for the particular inputs generated by application of the control values. Thus, to avoid penalties for completing too much work in the failure analysis activity, the controller removes all resources to prevent further progress. A surprising behavior, but clearly correct with respect to the semantics of the performance index. Figure 5.17 gives the accumulation if failure analysis work in the backlog that results. For reference, the amount by which the actual value of total work exceeds the estimated $w_{\text{total}} = 1574.4$ is causing the controller to disable the activity is approximately 4 units of work. Had the failure analysis activity been able to complete, it would’ve contributed approximately 20 more units of work to this total.

Lastly, Figure 5.18 gives the total workforce assigned across all activities, for all control partitions. From the plot it can be verified that the controller succeeds in
honoring the equality constraint implementing the narrative-motivated requirement that exactly 29 FTEs of workforce be allocated at all times.

Controlled Process Quality

As with the first scenario, the controller allocates process quality in a mostly binary fashion; this is expected as the second scenario uses a very similar weighting to the first. Hence the earlier analysis applies.

Note the apparent anomaly in Figure 5.19(c): the controller has selected a value for the process quality variable that is at the upper control bound, yet per Figure 5.16(c), has assigned a near-zero workforce to the activity. Upon closer inspection, the workforce value is small but non-zero, 0.0021, and therefore does not absorb the effect of the increased process quality.

Figure 5.20 gives the mean process quality assigned, per control partition, in the controlled outputs.

5.3 Summary of Controller Evaluation

In summary, both controllers perform the tasks set for them. In the first scenario, close tracking of the provided reference trajectories was achieved resulting in quantitative feedback to project management about the model inputs required over time to achieve such tracking.

In the second scenario, the controller effectively generates a phased schedule using its fixed workforce, wherein the development activities wax and wane more or less in an overlapping sequence. This is consistent with expected staffing levels per development activity over time found in the literature. Again, the primary result of the second scenario is the production of quantitative feedback regarding the magnitude and timing of resources to be applied in order to achieve the best trajectory the controller was able to find; also, a proposed schedule is generated to help the project manager in his planning task.
Figure 5.19. Nominal and controlled process quality, scenario 2.
Figure 5.20. Mean controlled process quality, scenario 2.
CHAPTER 6. APPLICATION IN PRACTICE

The primary barriers to application of the techniques developed in this work lie in two factors: First, the large cost of software projects makes project managers (who are responsible for the successful execution of such projects) leery of trying a new management technique. Such are seen as unnecessary sources of risk. This is understandable, as many of the early parametric cost and effort estimation models increase their predicted estimates if the project involves new technology. The second factor impeding use of the technology is the technical effort required to prepare the significant body of parameters required by the models.

The former issue cannot be addressed herein. However, as described in Chapter 2, practitioners have developed methods for estimated model parameters from historical data for more than 50 years. What complicates this task for the present work is its explicit accounting for the constraints that gate task availability based on relationships between development activities (i.e., the activation constraints). When modeling parametrically, one determines the power of certain metrics based on aspects of the project, process, or team to predict the eventual cost or effort figures. In contrast, the models of Chapter 3 are causal and dynamical; they do not simply predict a future value, they show the path that leads such a value.

6.1 Calibrating from Historical Data

The interconnected system-of-subsystems paradigm employed in Chapter 3 preserves the encapsulation of the submodels, and hence the calibration task can be undertaken on per-activity basis, so long as one carefully accounts for the impact of activation constraints. Looking within the development activities, only certain modeling components require calibration. For example, the backlog queue component has
only τ, a time-constant defining the gain by which the outflow rate will track the commanded rate; this parameter can need only be set to a reasonable value given the time-step for simulation, so a suitable value can be obtained by trial and error. Likewise, the activation controller components have no parameters. This leaves only the productivity model.

6.1.1 Calibrating the Productivity Model

The productivity model contains 4 calibrated parameters: α—a factor describing the scaling (stretch) to be applied to a Gaussian curve to represent the impact of “flow” [59]; β—a factor describing a translation (in its input space) of the center of Gaussian that represents the impact of flow; \( F_{\text{cap}} \)—the raw productive capability of one FTE of workforce; and ξ—a scale factor translating momentum to resistive force, per the dashpot analogy underlying the productivity model.

To calibrate instances of the productivity model, techniques in the field of parameter identification can be employed. The reader is referred to the appendices of [65] in which a calibration method is developed for the software system-test phase model of Cangussu [54], which in turn provided the initial motivation for the productivity model in Chapter 3.

6.1.2 Defect Detection

The defect detection model requires calibration for \( \mu \), the “encounter actualization probability” for enumerated potential encounters (per the analogy with the Lotka-Volterra predator/prey population dynamics model discussed in Chapter 3.)

The defect model calibration is relatively straight-forward, as the defect population model is based on COQUALMO. One needs simply fit the parameter to historical data in light of the COQUALMO estimate of defects present, less the number of defects that have been removed, given the number of test cases that have been executed. To reduce opportunities for statistical bias, this calibration should divide projects into
equally sized time slices, yielding an overdetermined set of equations for the fitting operation.

6.2 Managing the Impact of Activation Constraints

The method proposed for dealing with the activation constraints in the preparation of calibration data is an act of data grooming. That is, one must isolate cases which are subject to reduced productivity because of unsatisfied dependencies.

Modern development methodologies, such as “Scrum” [66], encourage daily updates of task status as “not started”, “in progress”, “blocked”, etc. In particular, the “blocked” attribute maps directly to the notion of an unsatisfied prerequisite and could serve as a ready source of information on the impact of activation constraints, if accurately captured.
CHAPTER 7. SUMMARY

This work sets forward a modeling and control framework for progress in software development activities. It undertakes an evaluation of the framework through demonstration using a typical model built within the framework, and scenarios based on nominal management control objectives.

7.1 Modeling

The modeling framework of Chapter 3 specifies a class of linearly interconnected systems comprising state-based and algebraic subsystems.

7.1.1 Submodels

The representation of work for all development activities is abstracted into the concept of a backlog queue, where progress is given as the measure of cumulative queue outflow. This abstraction permits arbitrary and independent substitution of modeling components as future research arrives at improved methods.

Queues are driven as the balance between work in-flow rates, and the effect of productivity, as regulated by activation constraints, to produce an out-flow rate.

Activation constraints capture a coarse notion of inter-task dependencies by specifying the maximal permitted cumulative queue out-flow as a time-varying threshold function, defined in the progress of other development activities.

The submodels of productivity, backlog, and activation constraints are combined into a generic development activity model. This activity model is combined with side-effect models through linear interconnections to represent the dominant dynamics of work completion for a specific development effort.
7.1.2 Interconnection

Some linear interconnections between subsystem models specify the in-flow channels, establishing producer-consumer relations between the development activities; others establish parametric dependencies between the subsystem models. In this manner, the instantiation of specific development activities, and the specification of the interconnections provides the mechanism for constructing a model of a specific development effort.

7.2 Decision Support

The decision support capability developed in Chapter 4 is based in the practices of Automatic Control, and in particular, on the synthesis of controllers implementing quadratic tracking and quadratic regulation.

Control is implemented through constrained minimization of a performance index establishing a partial order on candidate sets of control values. Two performance indices are developed, yielding the ability to control with respect to a notion of overall project completion, or to fine-grained specification of the expected evolution of each development activity.

The flowchart in Figure 4.1 captures the decision support process. It can be interpreted in two ways: The first is to execute the flowchart upon certain detected events in order to determine corrective actions. The second is as an instance of Model Predictive Control [67], wherein the flowchart is executed periodically, applying the control values from the first control partition, but (typically) recalibrating and recomputing the control values before the subsequent control partitions are reached. The distinction between these methods of application is summarized in the business context as the difference between management-by-exception and active management—where the former is practiced by higher-level managers, and the latter by technical leadership. Thus one observes a mapping of the proposed decision support method to the needs of various layers of the management hierarchy.
7.3 Evaluation

The evaluation undertaken in Chapter 5 demonstrates that the controllers developed according to Chapter 4 are successful in controlling models of the form developed in Chapter 3. This establishes the utility of the method for decision support, dependent only upon the following factors:

1. The degree of representation provided by the model to reflect the dynamics of the modeled process;

2. The “persistence of excitation” in the training data; informally, the richness of the behavior exhibited in the data over which the representative model is calibrated [68]; and

3. The numerical minimization’s ability to find a minimum that represents an improvement in the process—a probabilistic process based on the set of starting points, and the amenability of the model equation to the specific numerical minimization technique.

Given these factors, the combination of model and controller offers a project manager the capability to compute quantitative feedback on the nature (staffing capability vs. process efficiency), magnitude, and timing (by reference to specific control partitions) of interventions capable of improving achievement of management objectives, as specified through quantitative performance indices. Given a project manager’s ability to translate such quantitative feedback into corresponding process changes, he provides the capability to close the control loop. In this way, the decision support capability may be described as manager-in-the-loop control, or as an expert system. In any case, the controller provides quantitative suggestions to a project manager, which he, at his sole discretion, integrates into his decision making process.
7.4 Future Work

The primary undertaking left to future work is a detailed empirical validation of the modeling and control efficacy in an industrial context. While all component models are based on industrially validated formulae, there is a potential for process behavior to arise as more than the sum of its components. Such an empirical study is needed to establish the representationality of the models of Chapter 3. Alternatively, such study may motivate enhancement of the submodels, or the method of constructing the performance indices.

Additionally, investigation of methods for determining when a development effort is deviating from the plan are of interest. One potential method for undertaking this objective is to investigate the substitution of a frequently calibrated predictive modeling capability in place of work-product size measurement within existing methods of deviation detection. Such may construct a capability to predict progress deviations earlier leaving more opportunity to apply a corrective intervention.

Lastly, while the motivation, context, and examples within this work are all drawn from the domain of software development, there is seemingly little to prevent application of the model and controller to other social collaborative efforts, particularly where there exists a technical component or other source of intellectual challenge.
Appendix A. Example Model Parameters

A.1 Submodel Interconnections

A sparse-matrix representation is used to describe the connections in the $L_{11}$ matrix. Each connection is represented by a source, destination, and scale factor. Each source is described as a model component and a specific output element. Likewise, each destination is described as a model component and a specific input element. Lastly, a shorthand notation is used to identify the model components; it consists of a one-letter identifier for the type of component, followed by a short code to identify the development activity to which it belongs. This notation is abused to describe the

<table>
<thead>
<tr>
<th>Letter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Side-Effect Component</td>
</tr>
<tr>
<td>C</td>
<td>Activation Controller</td>
</tr>
<tr>
<td>P</td>
<td>Productivity Model</td>
</tr>
<tr>
<td>Q</td>
<td>Backlog Queue</td>
</tr>
</tbody>
</table>

Table A.1
Identifiers for model component types.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC</td>
<td>Feature Coding</td>
</tr>
<tr>
<td>TC</td>
<td>Test-case Coding</td>
</tr>
<tr>
<td>NTE</td>
<td>New Test Execution</td>
</tr>
<tr>
<td>RTE</td>
<td>Regression Test Execution</td>
</tr>
<tr>
<td>FA</td>
<td>Failure Analysis</td>
</tr>
<tr>
<td>FAF</td>
<td>Failure Analysis (Feature Failure Queue)</td>
</tr>
<tr>
<td>FAT</td>
<td>Failure Analysis (Test Failure Queue)</td>
</tr>
<tr>
<td>FCR</td>
<td>Feature Correction</td>
</tr>
<tr>
<td>TCR</td>
<td>Test-case Correction</td>
</tr>
<tr>
<td>DD</td>
<td>Defect Detection</td>
</tr>
<tr>
<td>DP</td>
<td>Defect Population</td>
</tr>
<tr>
<td>SPL</td>
<td>Proportional Splitter</td>
</tr>
</tbody>
</table>

Table A.2
Development activity/side-effect identifiers.
side-effect processes as well. Table A.1 gives the one-letter type-identifiers, Table A.2 gives the activity/side-effect codes, and the sparse matrix is described in Table A.3.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connections to feature coding inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PFC Backlog size</td>
<td>QFC Queue level</td>
<td>1</td>
</tr>
<tr>
<td>CFC Dependent queue cumulative outflow</td>
<td>QFC Cumulative outflow</td>
<td>1</td>
</tr>
<tr>
<td>CFC A priori commanded outflow rate</td>
<td>PFC Potential productivity</td>
<td>1</td>
</tr>
<tr>
<td>QFC Commanded outflow rate</td>
<td>CFC Regulated commanded outflow rate</td>
<td>1</td>
</tr>
<tr>
<td><strong>Connections to test-case coding inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PTC Backlog size</td>
<td>QTC Queue level</td>
<td>1</td>
</tr>
<tr>
<td>CTC Dependent queue cumulative outflow</td>
<td>QTC Cumulative outflow</td>
<td>1</td>
</tr>
<tr>
<td>CTC A priori commanded outflow rate</td>
<td>PTC Potential productivity</td>
<td>1</td>
</tr>
<tr>
<td>QTC Commanded outflow rate</td>
<td>CTC Regulated commanded outflow rate</td>
<td>1</td>
</tr>
<tr>
<td>QTC In-flow rate</td>
<td>QFAF Out-flow rate</td>
<td>1</td>
</tr>
<tr>
<td><strong>Connections to test-case execution inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PNTE Backlog size</td>
<td>QNTE Queue level</td>
<td>1</td>
</tr>
<tr>
<td>CNTE Dependent queue cumulative outflow</td>
<td>QNTE Cumulative outflow</td>
<td>1</td>
</tr>
<tr>
<td>CNTE A priori commanded outflow rate</td>
<td>PNTE Potential productivity</td>
<td>1</td>
</tr>
<tr>
<td>QNTE Commanded outflow rate</td>
<td>CNTE Regulated commanded outflow rate</td>
<td>1</td>
</tr>
</tbody>
</table>

*(Continued on next page)*
Table A.3 *(Continued from previous page)*

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNTE In-flow rate</td>
<td>QTC Out-flow rate</td>
<td>1</td>
</tr>
<tr>
<td>QNTE In-flow rate</td>
<td>QFCR Out-flow rate</td>
<td>1</td>
</tr>
<tr>
<td>QNTE In-flow rate</td>
<td>QTCR Out-flow rate</td>
<td>1</td>
</tr>
<tr>
<td>CNTE Dependent queue cumulative outflow</td>
<td>QFCR Cumulative outflow</td>
<td>-1</td>
</tr>
<tr>
<td>CNTE Dependent queue cumulative outflow</td>
<td>QTCR Cumulative outflow</td>
<td>-1</td>
</tr>
<tr>
<td>CNTE Progress measure for $c(\cdot)$</td>
<td>QFC Cumulative outflow</td>
<td>1</td>
</tr>
<tr>
<td>CNTE Productivity measure for $c(\cdot)$</td>
<td>QFC Out-flow rate</td>
<td>1</td>
</tr>
</tbody>
</table>

*Connections to regression execution inputs*

<table>
<thead>
<tr>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRTE Backlog size</td>
<td>1</td>
</tr>
<tr>
<td>CRTE Dependent queue cumulative outflow</td>
<td>1</td>
</tr>
<tr>
<td>CRTE A priori commanded outflow rate</td>
<td>1</td>
</tr>
<tr>
<td>QRTE Commanded outflow rate</td>
<td>CRTE Regulated commanded outflow rate</td>
</tr>
<tr>
<td>CRTE Progress measure for $c(\cdot)$</td>
<td>QFC Cumulative outflow</td>
</tr>
<tr>
<td>CRTE Productivity measure for $c(\cdot)$</td>
<td>QFC Out-flow rate</td>
</tr>
</tbody>
</table>

*Connections to failure analysis inputs*

<table>
<thead>
<tr>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFA Backlog size</td>
<td>1</td>
</tr>
<tr>
<td>PFA Backlog size</td>
<td>QFAT Queue level</td>
</tr>
<tr>
<td>CFA Dependent queue cumulative outflow</td>
<td>QFAT Cumulative outflow</td>
</tr>
<tr>
<td>CFA Dependent queue cumulative outflow</td>
<td>QFAT Cumulative outflow</td>
</tr>
</tbody>
</table>

*(Continued on next page)*
Table A.3 *(Continued from previous page)*

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFA</td>
<td>A priori commanded outflow rate</td>
<td>PFA</td>
</tr>
<tr>
<td>QFAF</td>
<td>In-flow rate</td>
<td>SDD</td>
</tr>
<tr>
<td>QFAT</td>
<td>In-flow rate</td>
<td>SDD</td>
</tr>
<tr>
<td>QFAF</td>
<td>Commanded outflow rate</td>
<td>SSPL</td>
</tr>
<tr>
<td>QFAT</td>
<td>Commanded outflow rate</td>
<td>SSPL</td>
</tr>
</tbody>
</table>

*Connections to feature correction inputs*

| PFCR | Backlog size | QFCR | Queue level | 1 |
| CFCR | Dependent queue cumulative outflow | QFCR | Cumulative outflow | 1 |
| CFCR | A priori commanded outflow rate | PFCR | Potential productivity | 1 |
| QFCR | Commanded outflow rate | CFCR | Regulated commanded outflow rate | 1 |
| QFCR | In-flow rate | QFAF | Out-flow rate | 1 |

*Connections to test correction inputs*

| PTCR | Backlog size | QTCR | Queue level | 1 |
| CTCR | Dependent queue cumulative outflow | QTCR | Cumulative outflow | 1 |
| CTCR | A priori commanded outflow rate | PTCR | Potential productivity | 1 |
| QTCR | Commanded outflow rate | CTCR | Regulated commanded outflow rate | 1 |
| QTCR | In-flow rate | QFAT | Out-flow rate | 1 |

*(Continued on next page)*
Table A.3 (Continued from previous page)

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connections to defect population Estimation inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDP Size (Features)</td>
<td>QFC</td>
<td>1</td>
</tr>
<tr>
<td>SDP Total defects removed (Feature)</td>
<td>QFCR</td>
<td>1</td>
</tr>
<tr>
<td>SDP Size (Tests)</td>
<td>QTC</td>
<td>1</td>
</tr>
<tr>
<td>SDP Total defects removed (Test)</td>
<td>QTCR</td>
<td>1</td>
</tr>
<tr>
<td><strong>Connections to defect detection inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDD Estimated defect population (Feature)</td>
<td>SDP Estimated defects present (Feature)</td>
<td>1</td>
</tr>
<tr>
<td>SDD Test execution rate (Regression)</td>
<td>QRTE</td>
<td>Out-flow rate</td>
</tr>
<tr>
<td>SDD Estimated defect population (Test)</td>
<td>SDP Estimated defects present (Test)</td>
<td>1</td>
</tr>
<tr>
<td>SDD Test execution rate (New tests)</td>
<td>QNTE</td>
<td>Out-flow rate</td>
</tr>
<tr>
<td><strong>Connections to proportional splitter inputs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSPL Reference 1</td>
<td>QFAF</td>
<td>1</td>
</tr>
<tr>
<td>SSPL Reference 2</td>
<td>QFAT</td>
<td>1</td>
</tr>
<tr>
<td>SSPL Quantity to be split</td>
<td>CFA</td>
<td>1</td>
</tr>
</tbody>
</table>

A.2 Incorporation of External Inputs

The elements of the $L_{12}$ are described in a fashion similar to those of $L_{11}$. The letter ‘U’ denotes an element of the external input vector. Table A.4 gives the sparse matrix.
Table A.4
Incorporation of external inputs via $L_{11}$.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFC Workforce size</td>
<td>UFC Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PFC Process quality</td>
<td>UFC Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PTC Workforce size</td>
<td>UTC Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PTC Process quality</td>
<td>UTC Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PNTE Workforce size</td>
<td>UNTE Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PNTE Process quality</td>
<td>UNTE Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PRTE Workforce size</td>
<td>URTE Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PRTE Process quality</td>
<td>URTE Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PFA Workforce size</td>
<td>UFA Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PFA Process quality</td>
<td>UFA Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PFCR Workforce size</td>
<td>UFCR Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PFCR Process quality</td>
<td>UFCR Process quality</td>
<td>1</td>
</tr>
<tr>
<td>PTCR Workforce size</td>
<td>UTCR Workforce size</td>
<td>1</td>
</tr>
<tr>
<td>PTCR Process quality</td>
<td>UTCR Process quality</td>
<td>1</td>
</tr>
</tbody>
</table>
A.3 Model Parameters

Table A.5 gives the parameters for the backlog queue components. Table A.6 gives the parameters for productivity model components. Table A.7 gives the defect population model parameters, and Table A.8 gives the defect detection model parameters.
A.4 Initial State

The scenario used throughout the examples is constructed to assume 30 units of feature coding work, 200 units of new test case coding work, and 200 units of regression test execution work. This is represented as an initial state-vector $x_0$ consisting of zeros, except for elements associated with the backlog-queue “level” elements described above; these are set to the values given above.
Appendix B. Two Methods of Minimization

Control laws defined in terms of the “arg.min” operator are frequently implemented numerically through constrained minimization. Below, two methods of implementing such control laws for continuous-time integral performance indices are described. The first method is used throughout the present work.

Numerical constrained minimization implementations are typically based on a loop defined as follows: (1) evaluate the performance index, repeatedly, to obtain numerical approximation of the Hessian matrix/Gradient/local topology for a given point in the input space (the “working minimization state”); (2) based on this numerical approximation, determine a state update; (3) update the working minimization state and repeat. Examples include sequential quadratic programming, gradient descent methods, and the Nelder-Mead [69] simplex.

The two methods below differ in the way they make use of this basic loop.

B.1 In-line Model Solution

The in-line model solution method relies upon having an oracle that can produce the solution to the initial value problem (IVP) defined by the model equations, the initial model state, and the variable model inputs, and the fixed model parameters. The performance index can then be computed directly from the model solution.

This method is designed to take advantage of the limited way that numerical constrained minimization algorithms use the performance index: they evaluate it at discrete points in the input space. That is, there are no symbolic operations—a closed form specification of the performance index is never required. Thus, for the purposes of the present work, the modified improved-Euler method developed in Section 3.9 provides a suitable oracle for solving the model IVP. This, in conjunction with the Trapezoidal Rule for evaluating the integrals in the performance index, provides a practical method for evaluating the control law. (The Trapezoidal Rule is required
to evaluate the integrals because the model solution produced by the improved-Euler method is rendered as a partial function/sampling.)

Construction

The numerical constrained minimization algorithms, “fmincon” in MATLAB\(^1\) and “sqp” in GNU Octave, require the performance index to be provided as a function handle. This function must be defined to receive, as its sole argument, the vector of variables over which the constrained minimization is to be performed. This means that the model parameters—and indeed, the model’s initial state—must be built into the function or must be defined globally so that all information required to solve the IVP will be available upon demand. Figure B.1 gives a block-diagram for this method.

\(^1\)MATLAB is a trademark of The MathWorks
Advantages and Disadvantages

To the constrained numerical minimization algorithm, this method does not require the addition of minimization variables, leading to a minimally-sized Hessian matrix/Gradient for derivative-based methods. However, having fewer minimization variables implies that each variable must explain a greater portion of the performance index’s curvature about a point in the input space. This implies that the neighborhood about the current minimization state over which the numerical approximation is a valid representation of the performance index, is smaller. This in turn implies a need for smaller steps, and a higher iteration count.

Additionally, because the model equations are effectively simulated upon each evaluation of the performance index, all constraints related to the modeling equations can be transferred to the IVP solution instead, leaving to the constrained minimization only those constraints that are inherent in the performance index. This can result in substantial time savings.

B.2 Model Equations as Equality Constraints

Under this method, the set of minimization variables is augmented. The new variables are intended to represent the model state as sampled on a time-grid with step size $h$, spanning the minimization period. Given variables representing the state-trajectory, and variables representing the values of the model inputs, the performance index can be evaluated directly using a numerical quadrature to evaluate the integrals (assuming the fixed model parameters have been built-in, or are globally available).

Unfortunately, the numerical minimization algorithms have no support for the concept of “model equations” nor do they distinguish certain minimization variables as “state-trajectory” variables, versus “standard” minimization variables. Left unconstrained, the numerical minimization algorithm will treat the new variables as completely free variables.
Thus, in order to cause the minimization process to produce a valid state trajectory, equality constraints are applied to constrain the values permitted in adjacent state-sample-variables. These equality constraints are designed to require that adjacent state-samples honor a discretization of the model equations given the time step \( h \) (e.g., using the Midpoint Rule). In this way, the constrained minimization algorithm is coerced to search for both the state trajectory, and the inputs that produce it, concurrently.

Construction

Let \( \dot{x} = f(x, u) \) define the model equations for a sample system, with time-dependent state \( x \), and fixed vector-valued input, \( u \). Let \( J(\chi_1, \chi_2, \ldots, \chi_n, u) \) be a performance index defined in terms of the input vector \( u \), the known initial state \( x_0 \), and a sequence of model-state samples \( \{\chi_i\} \) spaced with time step \( h \) on a regular grid that represent a solution to the model IVP.

Then the control law is given as

\[
\arg \min_{\{\chi_i\}, u} J(\chi_1, \chi_2, \ldots, \chi_n, u) \tag{B.1}
\]

subject to equality constraints (based on the Midpoint Rule)

\[
(\chi_1 - x_0) - hf\left(\frac{x_0 + \chi_1}{2}, u\right) = 0 \tag{B.2}
\]
\[
(\chi_2 - \chi_1) - hf\left(\frac{\chi_1 + \chi_2}{2}, u\right) = 0 \tag{B.3}
\]
\[
\vdots
\]
\[
(\chi_n - \chi_{n-1}) - hf\left(\frac{\chi_{n-1} + \chi_n}{2}, u\right) = 0 \tag{B.4}
\]

and specifying additional constraints as necessary. Here, the equality constraints force the model equations to hold insofar as a Midpoint Rule evaluation represents an approximate numerical solution of the model equations.
Advantages and Disadvantages

With this method, the number of minimization variables increases, leading to a larger Hessian matrix or Gradient (where relevant). Certain performance indices, however, become convex or nearly convex in the minimization variables under this method. This means the numerical approximation is a valid representation of the performance index over a larger neighborhood about the working minimization state. This, in turn, implies larger steps/fewer iterations.
LIST OF REFERENCES


VITA
VITA

Scott D. Miller was awarded the Ph.D. in computer science from Purdue University in 2013. At Purdue, Scott studied applications of feedback control to software process modeling and control under advisors Aditya Mathur and Raymond DeCarlo.

Upon completion of the Ph.D., Scott continued in his role as Principal Scientist within the Power and Microelectronics Group of Microsemi Corporation where he leads a small security-focused research and development team.

Born in rural northwestern Indiana, Scott was introduced to computer programming at age 9 by his father who one day brought home a Commodore PET, and a stack of issues of Byte magazine, stating only “computers are going to be the future.”

Scott enjoys playing various musical instruments and spending his newly found free time with his lovely wife Michelle, and two rambunctious boys.