Models and Algorithms for Integrated Agile Software Planning and Scheduling

by

Ákos Szőke

Supervisor: Prof. András Pataricza, DSc.

Electrical Engineering and Informatics
Department of Measurement and Information Systems

March, 2014
The focus of my PhD research is on software release and iteration planning to support the development coordination in agile development organizations in both co-located and distributed environments. The main contribution of the research proves the design, application and validation of an Integrated Agile Planning Approach (IAPA). My elaborated IAPA supports development coordinators to cope with the complexity and the dynamics of software development by providing conceptual models, optimization models and algorithms for semi-automatic software release and iteration planning.

Industrial software development is a highly complex and dynamic process. The success of software organizations depends on the effectiveness and efficiency of development activities where development coordination takes a key role. The coordination usually faces a decision problem which aim is to determine which features should be delivered in the next sequence of releases. The result of this decision making is manifested by release plans. These plans give a description about which features to implement in which software systems’ deliveries to provide maximal business value. Once the features are selected the coordination usually meets an other decision problem which goal is to find out how to realize the planned features. The outcomes of this decision making are expressed by iteration plans, which establish resource allocation to the realization tasks of features while they consider the constraints of delivery.

The presented contribution is fundamentally different from the existing – mainly intuitive – methods in its information fusion and its mathematical optimization approach. Integration of managerial and software engineering information, and the provided algorithmic optimization methods easily resolves complex decision situations, gives the business increased visibility, and it can also provide constantly up-to-date decision supports considering changes necessitated by shifting business priorities.

The IAPA was worked out in the frame of the Pythia research project which was supported in part by a GVOP grant.

The IAPA is implemented in three prototypes and has been successfully applied in everyday basis for many years at a software development organization. The effectiveness and efficiency of the approach are supported with empirical evidence from industrial case studies and post-mortem simulations.
Nyilatkozat

Alulírott Szőke Ákos kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amely szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2014. február 27.

____________________________________

Szőke Ákos

A dolgozat bírálatai és a védésről készült jegyzőkönyv a későbbiekben a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában elérhetők.
“The ideal work planning process should always provide the development team with best thing to work on next, no more and no less.”

Corey Ladas
Acknowledgments

First of all, I would like to thank my supervisor Prof. András Pataricza DSc. for helping my work during the last few years. I learned a lot during this time and I am convinced that this knowledge will help me in the future. He proposed the initial direction of the research for me that triggered several parallel tracks and led to the results described in the dissertation.

I received support for my research from Dr. György Strausz PhD. associate professor also, my colleague at my workplace at Multilogic Ltd – where I have been working as a project leader and a CTO for 10 years. Our relationship dating back prior to the beginning of my PhD studies. Our conversations generated important contributions in the dissertation.

I am very grateful to my colleagues at the Multilogic for the stimulating environment. Discussions with many of them on various topics helped me in my research work. I should also acknowledge Dr. András Balogh PhD. and Dr. István Ráth PhD. (BME MIT FTSRG/OptWare), András Millinghofffer, Dr. Péter Antal associate professor PhD. (BME MIT) and Gábor Kiss (Multilogic) for the a fruitful joint work in Pythia Project that resulted in PYTHIA PROJECT PLANNER™ prototype.

I am grateful to Gábor Károlyi (Deputy Director of the IT Development department at Hungarian Post) and his group, Magdolna Udvardy (Director of IT Strategy department at Hungarian Tax and Financial Control Administration (APEH)), Dr. Sándor Dolgos (Head of IT Development department at BBraun Medical Hungary), Péter Halácsy (Head of Product and co-founder of Prezi.com) and colleagues at Prolan for their support during my visit on the field. The joint work on several projects were very interesting and valuable for me. The inspiring conversations kept me productive enough to make significant progress in my work even during the short visits.

I am also grateful to Dr. Marjan van den Akker PhD. (assistant professor at University of Utrecht) for the data sets to validate my algorithms, Dr. Zornitza Racheva PhD. (associate at University of Twente), Dr. Vladimir Mandic PhD. and Dr. Kari Liukkunen PhD. (associate professors at University of Oulu), and Dr. Val Casey PhD. (assistant professor at Dundalk Institute of Technology) for the fruitful discussions, joint work, and/or their encouragement and support.

I received very valuable feedbacks and inspirations from many agile software practitioners also. Most of them I met at the Budapest Agile Meetup series organized by the Agile Alliance Hungary.

Part of my work has been supported by the GVOP R&D grant (GVOP-3.3.3-05/1.-2005-05-0046/3.0) and realized by the cooperation of Multilogic and OptXware.

Finally, I would like to dedicate this Dissertation to those people who did everything to establish a stable background for me – they gave love, encouragement and support: Klára (my wife), Klárika and Zita (my daughters), my parents Éva and Sándor. Without their support, I could not have finished this work.
## Contents

Abstract ................................................................. ii
Declaration ............................................................... iii
Acknowledgments ....................................................... v

List of Figures ......................................................... xi
List of Tables .......................................................... xii
List of Algorithms .................................................... xiii

1 Prelude: Introduction ............................................... 14
   1.1 Overview of Agile Software Development .................. 14
      1.1.1 Specific Features of Agile Methods .................. 15
      1.1.2 Characteristics ....................................... 16
      1.1.3 Target Field ........................................ 16
      1.1.4 Main Representatives of Agile Methods ............ 17
      1.1.5 Characteristics in Distributed Environments ...... 18
   1.2 Planning of Software Development Environments ...... 19
      1.2.1 Predictive and Adaptive Planning Processes .... 20
      1.2.2 Comparison of Predictive and Adaptive Approaches Using Control Theory Analogy 21
   1.3 Overview of Agile Planning ................................ 22
      1.3.1 Agile Planning Cycles ............................... 22
   1.4 Motivation .................................................... 25
      1.4.1 Relevancy of the Research ........................... 26
      1.4.2 Agile Release Planning ............................... 28
      1.4.3 Agile Iteration scheduling ........................... 29
      1.4.4 Distributed Extension of Agile Release Planning 30
   1.5 Existing Approaches ......................................... 31
      1.5.1 Requirements Modeling ................................ 31
         1.5.1.1 Use Case-based Requirements .................. 31
         1.5.1.2 Goal-oriented Requirements Modeling ........ 32
      1.5.2 Dependencies between Requirements ................. 33
      1.5.3 Effort estimation ..................................... 34
         1.5.3.1 COCOMO II ....................................... 34
         1.5.3.2 Use Case Point Method ......................... 35
      1.5.4 Prioritization ......................................... 37
      1.5.5 Release planning ...................................... 37
      1.5.6 Distributed Software Development ................... 38
   1.6 Overview of the Research .................................... 38
      1.6.1 Open Problems and Differences in Agile Software Development 39
      1.6.2 Objectives of the Research .......................... 40
      1.6.3 Research Methodology ................................ 40
   1.7 Structure of the dissertation ............................... 41

2 Discrete Mathematical Background ............................. 44
   2.1 Discrete Optimization ....................................... 44
      2.1.1 Outline .............................................. 44
   2.2 Combinatorial Optimization ................................ 45
   2.3 Integer Programming ......................................... 45
   2.4 Knapsack Problems .......................................... 47
CONTENTS

3.3.2 Data Collection and Results ........................................ 85
3.3.3 Analysis ............................................................... 86
  3.3.3.1 Qualitative Analysis ............................................. 86
  3.3.3.2 Quantitative Analysis ........................................... 88
3.3.4 Computational Benchmarking ....................................... 90
  3.3.4.1 Solving Time ...................................................... 90
  3.3.4.2 Quality of Plans ................................................ 91
3.4 Discussion ............................................................... 93
3.5 Conclusions .............................................................. 96

4 Optimized Agile Iteration Scheduling .................................. 98
  4.1 Synopsis ............................................................... 98
    4.1.1 Related Work ..................................................... 99
    4.1.2 Problem Statement and Analysis .................................. 99
    4.1.3 Objectives ........................................................ 100
     4.1.3.1 Outline ....................................................... 100
  4.2 Decision Support in Iteration Scheduling ......................... 100
    4.2.1 Conceptual Model of Agile Iteration Planning .................. 101
    4.2.2 Mapping to RCPSP ................................................. 102
    4.2.3 Formulating AISP Optimization Model ......................... 102
     4.2.3.1 Temporal and Resource Constraints ......................... 102
     4.2.3.2 Optimization Model ......................................... 103
    4.2.4 Solving the AISP Optimization Problem ....................... 103
     4.2.4.1 Solution Analysis ............................................ 105
    4.2.5 Tool Support ..................................................... 105
    4.2.6 Experimental Results ........................................... 105
     4.2.6.1 Research Questions ........................................... 106
     4.2.6.2 Context and Methodology ................................... 106
     4.2.6.3 Data Collection and Results ................................ 106
     4.2.6.4 Analysis ...................................................... 107
    4.2.7 Computational Benchmarking ................................... 109
     4.2.7.1 Solving Time .................................................. 109
     4.2.7.2 Quality of Plans ............................................. 110
  4.3 Iteration Scheduling from Use Case-based Specification ....... 111
    4.3.1 Mapping Use Case-based Specification to AISP Optimization Model ......................................................... 111
     4.3.1.1 Scheduling Properties of Use Cases and Actors .......... 112
     4.3.1.2 Duration Calculation from Use Case Diagrams .......... 112
     4.3.1.3 Precedence Constraints from Use Case Diagrams ........ 112
     4.3.1.4 Solution Analysis ............................................ 113
    4.3.2 Tool support ..................................................... 114
    4.3.3 A Case Study ..................................................... 114
     4.3.3.1 Setting the hypotheses ...................................... 115
     4.3.3.2 Context and Methodology ................................... 115
     4.3.3.3 Data collection and Results ................................ 115
     4.3.3.4 Case Study Analysis ......................................... 116
  4.4 Goal-oriented Extension of Iteration Scheduling ............... 117
    4.4.1 Mapping GRL to Use Case-based Specification .............. 117
    4.4.2 Tool support ..................................................... 118
    4.4.3 A Case Study ..................................................... 118
     4.4.3.1 Setting the hypotheses ...................................... 118
     4.4.3.2 Context and Methodology ................................... 119

viii
CONTENTS

4.4.3.3 Data collection and Results ........................................... 119
4.4.3.4 Case Study Analysis .................................................... 120
4.5 Discussion ............................................................................ 121
4.6 Conclusions .......................................................................... 123

5 Distributed extension of Agile Release Planning .......................... 125
  5.1 Synopsis ............................................................................. 125
    5.1.1 Challenges in Distributed Software Development ................. 126
    5.1.2 Related Work .................................................................. 126
    5.1.3 Problem Statement and Analysis ...................................... 127
    5.1.4 Objectives ..................................................................... 127
      5.1.4.1 Structure of the Chapter ............................................ 128
  5.2 Distributed Extension of Agile Release Planning ....................... 128
  5.3 Feature Partitioning Method ................................................... 130
    5.3.1 Feature Architectural Similarity Analysis ......................... 130
      5.3.1.1 Module Relatedness of Features ................................. 130
      5.3.1.2 Feature Architectural Similarity Measure .................... 131
    5.3.2 Feature Chunk Construction ........................................... 132
      5.3.2.1 Mapping to Minimum k-cut ........................................ 132
    5.3.3 Formulating FCCP Optimization Model ............................. 133
    5.3.4 Tool Support .................................................................. 134
  5.4 Experimental Results ........................................................... 134
    5.4.1 Research Questions ........................................................ 134
    5.4.2 Context and Methodology .............................................. 135
    5.4.3 Data Collection ................................................................ 135
    5.4.4 Results and Analysis ..................................................... 135
    5.4.5 Computational Benchmarking ......................................... 137
      5.4.5.1 Solving Time ............................................................ 138
      5.4.5.2 Quality of Plans ....................................................... 138
  5.5 Discussion ............................................................................ 140
  5.6 Conclusions .......................................................................... 141

6 Postlude: Conclusions ................................................................. 143
  6.1 Fulfillment of Research Objectives ....................................... 144
  6.2 Utilization of the New Results and Open Problems ................... 149
  6.3 Epilogue ............................................................................... 149
  6.4 List of Publications ............................................................... 150

7 References .................................................................................. 153
  7.1 Additional References .......................................................... 161

A Notations ..................................................................................... 163

B Acronyms .................................................................................... 166
## CONTENTS

### C Algorithms
- **Algorithm produceCCPrec** .................................................................................. 168
- **Algorithm mksched** ............................................................................................... 170
- **Algorithm lscap** ...................................................................................................... 175

### D Models for Agile Software Development
- **UML Profile Pythia** .................................................................................................. 178
- **Ontology of Integrated Agile Software Planning and Scheduling** .......................... 183

### E Overview of the Prototypes
- **PROPAS\textsuperscript{TM}** .......................................................................................... 190
- **PYTHIA PROJECT PLANNER\textsuperscript{TM}** ............................................................ 192
  - **Pythia Planner** ......................................................................................................... 192
  - **Pythia Decision** ......................................................................................................... 195
- **SERPA\textsuperscript{TM}** ................................................................................................ 196

### F Complexity of Algorithms and Problems
- **Complexity of Algorithms** ....................................................................................... 199
  - **Theoretical Time Complexity** .................................................................................. 200
- **The complexity of problems** ..................................................................................... 201
  - **Basic Complexity Classes** ...................................................................................... 201
  - **The complexity of optimization problems** .............................................................. 202
- **Dealing with $\mathcal{NP}$-hardness** .......................................................................... 202
  - **Local Search Techniques** ....................................................................................... 202
  - **Branch-and-Bound Algorithms** .............................................................................. 204
  - **Heuristic Algorithms** .............................................................................................. 205
  - **Approximation Algorithms** .................................................................................... 205

### Index

---

x
<table>
<thead>
<tr>
<th>Section</th>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CMMI Project Planning and its Specific Goals</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>The Adaptive Approach as a Feedback Control</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Agile Planning Levels</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>Agile Planning Cycles</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>Example of Agile Software Delivery Cycles</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>Agile Adoption Rate by Different Organization Size</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>Followed Agile Methods</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>Employed Agile Techniques</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>Agile Release Planning</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>1.10</td>
<td>Agile Iteration Planning</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>1.11</td>
<td>Distributed Agile Release Planning</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>1.12</td>
<td>Contributions in the Context of Agile Planning Cycles</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Branches of Discrete Optimization Problems that Relate to the Research</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Project Planning within the Life-cycle of a Project</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Example AoN Project Network</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Visualization of Release Planning Dependencies: Coupling with Cluster and Precedence with Directed Edge Notation</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of the Agile Release and Iteration Scheduling Model (ARISM) Concepts</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Agile Release and Iteration Scheduling Model (ARISM)</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Illustration of Feature Package Construction</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Release Scheduling Using the mksched Algorithm (Alg. 3)</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>4P of Historical (upper) and Optimized (lower) Plans</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Agile Iteration Scheduling Model (AISM)</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Generated Iteration Schedule</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Boxplots of Intuitive (above) and Optimized (below)</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Precedence Relations of Credit Controller Example</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Release Plan of the Credit controller example</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Credit Controller Description Using GRL (left), Use cases (right), and their Mappings (dotted line)</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of MR, SC, and PE</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Distributed Agile Release Planning</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Example Values on Module Relatedness Variable (MRV): each column denotes one dimension and every row shows an instance of the MRV</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>FASG with 33 Elements in Partitioned Form</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>E.1</td>
<td>Pythia Planner: Requirements Editor Screen</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>E.2</td>
<td>Pythia Planner: the Pre-scheduling State of the Project Network</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>E.3</td>
<td>Pythia Planner: the Built-in CO*MO (COCOMOII, CORADMO, COQUALMO) Estimation Toolset</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>E.4</td>
<td>Pythia Planner: the Exported Project Plan Can Be Seen in Microsoft Project</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>E.5</td>
<td>Pythia Decision: Main Screen</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>E.6</td>
<td>SERPA™: a Prototype Website at Multilogic</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>E.7</td>
<td>SERPA™: a Prototype Website for Collaborating Workspace of Agile Projects</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>F.1</td>
<td>Basic Complexity Classes</td>
<td>201</td>
<td></td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Decision criteria for predictive and agile approaches</td>
<td>17</td>
</tr>
<tr>
<td>1.2</td>
<td>Most referenced agile software development methods</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Main difference between plan-based and agile software development approaches</td>
<td>22</td>
</tr>
<tr>
<td>1.4</td>
<td>Type of dependencies between requirements</td>
<td>33</td>
</tr>
<tr>
<td>1.5</td>
<td>Examples of Dependencies</td>
<td>34</td>
</tr>
<tr>
<td>1.6</td>
<td>COCOMO II Project Scale Factors ($W_i$)</td>
<td>35</td>
</tr>
<tr>
<td>1.7</td>
<td>COCOMO II Effort Multipliers ($EM_i$) (Post Architecture Phase)</td>
<td>35</td>
</tr>
<tr>
<td>1.8</td>
<td>Technical Complexity Factors ($T_i$)</td>
<td>36</td>
</tr>
<tr>
<td>1.9</td>
<td>Environmental Complexity Factors ($E_i$)</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Mapping requirement dependencies to feature dependencies</td>
<td>70</td>
</tr>
<tr>
<td>3.2</td>
<td>Concepts of Agile Release and Iteration Planning Model (ARISM)</td>
<td>74</td>
</tr>
<tr>
<td>3.3</td>
<td>Relations of Agile Release and Iteration Planning Model (ARISM)</td>
<td>75</td>
</tr>
<tr>
<td>3.4</td>
<td>Historical release schedule values ($R_A - R_D$)</td>
<td>85</td>
</tr>
<tr>
<td>3.5</td>
<td>Optimized release plan values ($R_A^* - R_G^*$)</td>
<td>85</td>
</tr>
<tr>
<td>3.6</td>
<td>Comparison with descriptive statistics</td>
<td>88</td>
</tr>
<tr>
<td>3.7</td>
<td>Comparison of the historical and the $R_A^* - G$ paddings</td>
<td>89</td>
</tr>
<tr>
<td>3.8</td>
<td>Simulation results</td>
<td>91</td>
</tr>
<tr>
<td>3.9</td>
<td>Simulation results of plan quality</td>
<td>92</td>
</tr>
<tr>
<td>4.1</td>
<td>Iteration Planning Data</td>
<td>106</td>
</tr>
<tr>
<td>4.2</td>
<td>Intuitive Schedules</td>
<td>107</td>
</tr>
<tr>
<td>4.3</td>
<td>Optimized Schedules</td>
<td>108</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparison of Schedules</td>
<td>109</td>
</tr>
<tr>
<td>4.5</td>
<td>Simulation results</td>
<td>110</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of Schedules</td>
<td>111</td>
</tr>
<tr>
<td>4.7</td>
<td>Type of dependencies between requirements in precedence relations</td>
<td>112</td>
</tr>
<tr>
<td>4.8</td>
<td>Comparison of Projects’ State Variables</td>
<td>116</td>
</tr>
<tr>
<td>4.9</td>
<td>Comparison of Projects’ Response Variables</td>
<td>116</td>
</tr>
<tr>
<td>4.10</td>
<td>Mapping GRL to Use Case-based Specification</td>
<td>118</td>
</tr>
<tr>
<td>4.11</td>
<td>Comparison of Project Measures - I</td>
<td>120</td>
</tr>
<tr>
<td>4.12</td>
<td>Comparison of Project Measures - II</td>
<td>120</td>
</tr>
<tr>
<td>5.1</td>
<td>Historical requirements dispersion</td>
<td>135</td>
</tr>
<tr>
<td>5.2</td>
<td>CCI between teams</td>
<td>135</td>
</tr>
<tr>
<td>5.3</td>
<td>Team capacity differences in the historical and the algorithmic cases</td>
<td>136</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of Statistics</td>
<td>137</td>
</tr>
<tr>
<td>5.5</td>
<td>Simulation results</td>
<td>138</td>
</tr>
<tr>
<td>5.6</td>
<td>Simulation results of plan quality</td>
<td>139</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of Contribution 1</td>
<td>146</td>
</tr>
<tr>
<td>6.2</td>
<td>Summary of Contribution 2</td>
<td>147</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of Contribution 3</td>
<td>148</td>
</tr>
</tbody>
</table>
List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Date-driven and scope-driven agile release scheduling process</td>
<td>69</td>
</tr>
<tr>
<td>2</td>
<td>produceCCPreq algorithm</td>
<td>71</td>
</tr>
<tr>
<td>3</td>
<td>mksched algorithm (2/1)</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>mksched algorithm (2/2)</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>lcap algorithm with AF strategy</td>
<td>104</td>
</tr>
<tr>
<td>6</td>
<td>Iterative Improvement</td>
<td>203</td>
</tr>
<tr>
<td>7</td>
<td>Branch-and-Bound</td>
<td>204</td>
</tr>
</tbody>
</table>
Chapter 1

Prelude: Introduction

Modern software development methodologies rely on iterative development processes (IDP), and implement different best practices depending on multiple factors like application domain, development environment etc. The common characteristic of different IDPs is the staged-delivery (or incremental) approach, which constitutes to cyclic development lifecycle. The agile software development can be considered as the cleanest form of IDP. This chapter focuses on the comparison of agile approach with traditional methods and provides background information to the contributions.

1.1 Overview of Agile Software Development

In the late 1990’s several methodologies (e.g. RUP, XP, FDD, Scrum [29, 30, 31, 32]) began to get increasing public attention. Each had a different combination of new ideas, old ideas, and altered old ideas. However, they all emphasized 1) close collaboration between the programmers and business experts; 2) face-to-face communication between programmers and customers (as more efficient than written documentation); 3) frequent delivery of new deliverable business value; and 4) tightly cooperative and self-organizing teams [33]. These characteristics formed the idea of agile software development.

Agile software development phrase is used to label many recently emerged software development methods. These methods are suggested by experienced practitioners and have had a huge impact on how software is developed worldwide nowadays. The number of agile methods can be explained by the large number of different software development environments where their creator worked: they often included the specificities of the environments [34]. The most popular agile methods are usually simple and prescribe as few rules (that should be followed when someone apply them) as possible (e.g. Scrum [32]). Although, their primary aims are to deliver faster, better, and cheaper solutions both for the developers’ and the customers’ side. Generally, they offer numerous benefits to organizations, including quicker return on investment, higher product quality, and better customer satisfaction.

The thesis focuses on agile software development without loss of generality of the theoretical results for other IDPs, as the agile methodology is the cleanest form of IDP. Agile software development itself constitutes an entire group of methodologies based on the iterative and incremental approach, where requirements and solutions evolve through collaboration between self-organizing, and cross-functional
teams. The methodology was created as a reaction to the unsuccessfulness of rigid methodologies leading to the software crisis in the late 90s in order to support the adaptation to the frequently changing and rapidly evolving user demands.

The Agile Manifesto defined the term of ”agile software development” in 2001 [35]. The underlying four principles are the followings [30, 32]:

1. *Individuals and interactions*: encouraging self-organization and increasing motivation.
2. *Working software*: presenting working software to customers as soon as possible.
3. *Customer collaboration*: continuous user and/or customer involvement.
4. *Responding to change*: focusing on quick responses to change and continuous development.

Agile Manifesto also defines twelve principles as follows [35]:

1. Customer satisfaction by rapid delivery of useful software,
2. Welcome changing requirements, even late in development,
3. Working software is delivered frequently (weeks rather than months),
4. Working software is the principal measure of progress,
5. Sustainable development, able to maintain a constant pace,
6. Close, daily co-operation between business people and developers,
7. Face-to-face conversation is the best form of communication (co-location),
8. Projects are built around motivated individuals, who should be trusted,
9. Continuous attention to technical excellence and good design,
10. Simplicity,
11. Self-organizing teams, and
12. Regular adaptation to changing circumstances.

### 1.1.1 Specific Features of Agile Methods

Software development methods are sometimes characterized as a point on an interval from *predictive* to *adaptive*. Predictive methods focus on visioning and planning the future in full details. A predictive team announces exactly what features are planned for the entire duration of the development process. Their planning lookahead typically start from 3 months up. The plan is typically derived from the original objectives, and predictive teams have difficulty changing direction. Predictive teams often institute to ensure that only the most important changes are considered. [36, 29]

Agile methods, in contrast, lie on the adaptive end of the interval. An adaptive team will have difficulty describing what features are planned for the entire duration of the development process. Their planning
lookahead typically ends in 3 months or less. Adaptive methods focus on adapting to changing realities quickly. When project changes emerge then the team adapt oneself to the changes as well.\cite{37, 38, 34}

Both extreme approaches have their pros and cons. Predictive processes are supported by many technologies like model-driven architecture. This approach is typical in medium and large scaled projects (a few of tens of developers and more). Considering the difficulties of adaptation of large teams and programs, the development lead to a chaotic concept and implementation without this attitude. In contrast, adaptive approaches typically dealing with moderate size problems allow refactoring as the core technology ensuring the architectural consistency of the individual subsequent versions each providing a dedicated solution to the actual problem to be solved.

On the other hand, adaptivity reduces or eliminates the mismatch between the user requirements and the services delivered by the final product originating in the uncertainty of the requirements. From technical point of view, best predictive practices delegate reaction to the evolution of user needs to the phase of augmentative maintenance. Preservation of a consistent software functional and implementation architecture is served by the localization of changes, for instance by a strict componentisation.

### 1.1.2 Characteristics

There are many specific agile development methods\cite{37, 38, 34}. Most methods promote development, collaboration, teamwork, and process adaptability throughout the life-cycle of the software project.

Agile methods break deliverables into small *iterations*. Iterations are short time frames (*timeboxes*) that typically last from one to four weeks. Each iteration involves a full software development cycle, including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working software is demonstrated to users and/or customers. This minimizes overall risk and allows quick adaptation to changes. Multiple iterations constitute to a release when the new features delivered to the live environment of the customer’s IT infrastructure.

Agile methods emphasize *face-to-face communication* over written documents when all the team members are in the same location. Most agile teams work in a single open office, which facilitates such communication. Team composition in an agile project is usually *cross-functional* and *self-organizing* without considering any corporate hierarchy or the corporate roles of team members. Team size is typically limited in 5 – 12 members to ensure team communication and team collaboration. Larger development efforts may be delivered by multiple teams working towards to a common goal after partitioning a complex problem into different parts.

Agile development emphasizes functioning software as the primary measure of progress. Specific tools and techniques are often used to improve the quality and enhance project agility such as continuous integration, automated test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques.

### 1.1.3 Target Field

Recent surveys explain the growing popularity of agile approaches showing that agile teams are often more successful than traditional ones\cite{34, 39}. Several studies pointed out an increase as high as 60%...
1.1. Overview of Agile Software Development

in productivity, quality and improved user and/or customer satisfaction [39, 40], 40% faster time-to-market, and 60% and 40% reduction in pre-, and post-release defect rates [41] compared to the industry average. The most popular agile methods are Scrum [32](54%), Extreme Programming (XP) [30](2%), and Scrum/XP Hybrid (11%) [42]. Recently, many practitioners have adopted the ideas of agile software development [35].

However, there are some well visible limitations and risks related to agile processes [43, 44, 45]:

- Large-scale development efforts in a co-located team (> 20 developers),
- Distributed development efforts (physically dispersed teams), and
- Mission-critical systems where failure is not an option at any cost (e.g. surgical procedures).

Additionally, agile development methods have also been criticized by some practitioners and academics, mainly focusing on the following aspects [34]:

- The lack of focus on architecture leads to suboptimal design-decisions. [46]
- There is little scientific support for many of the agile claims. [46]
- Agile development are suitable for small teams. For larger projects other processes are more appropriate. [47]
- Agile development is nothing new, such practices have been in place in software development since the 1960s. [48]

In [49], risk analysis is used to estimate the best matching use cases for predictive and agile methods. The suggested decision criteria are shown in Table 1.1.

<table>
<thead>
<tr>
<th>Viewpoints</th>
<th>Predictive approach</th>
<th>Agile approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criticality of functions</td>
<td>High criticality</td>
<td>Low criticality</td>
</tr>
<tr>
<td>Team maturity</td>
<td>Junior developers</td>
<td>Senior developers</td>
</tr>
<tr>
<td>Environmental changes</td>
<td>Requirements are stable</td>
<td>Requirements change often</td>
</tr>
<tr>
<td>Team size</td>
<td>Large number of developers</td>
<td>Small number of developers</td>
</tr>
<tr>
<td>Team coordination</td>
<td>Culture that demands order</td>
<td>Culture that thrives on chaos</td>
</tr>
</tbody>
</table>

1.1.4 Main Representatives of Agile Methods

In [50], the history of iterative and incremental development is described, in which Dynamic Systems Development Method is identified (DSDM) [51] as the first agile method. It followed by eXtreme Programming (XP) [30], which is originated from the Chrysler C3 project in 1996 [52]. After that time several further methods followed them – see Table 1.2 for an overview of the most referenced agile development methods.
### 1.1. Overview of Agile Software Development

#### Table 1.2: Most referenced agile software development methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
</table>
| **Crystal methodologies**    | - A family of methods for co-located teams of different sizes and criticality: Clear, Yellow, Orange, Red, Blue  
- Crystal Clear, the most agile method, focuses on communication in small teams developing not life-critical software  
- Characteristics: frequent delivery, reflective improvement, osmotic communication, personal safety, focus, easy access to expert users, and requirements for the technical environment | [53]      |
| **Dynamic software development method** | - Divides projects into three phases: pre-project, project lifecycle, and post project  
- Defines nine principles: user involvement, empowering the project team, frequent delivery, addressing current business needs, iterative and incremental development, allow for reversing changes, high-level scope being fixed before project starts, testing throughout the lifecycle, and efficient and effective communication | [51]      |
| **Extreme programming**      | - Focuses on best practice for development  
- Consists of twelve practices: the planning game, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, 40-h week, on-site customers, and coding standards | [30]      |
| **Feature-driven development** | - Combines model-driven and agile development by emphasizing on division of work in features, initial object model, and iterative design  
- An iteration of a feature development includes: design and development phases | [31]      |
| **Lean software development** | - An adaptation of principles from lean production and, in particular, the Toyota production system to software development  
- It consists of seven principles: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity, and see the whole | [54]      |
| **Scrum**                    | - Focuses on project management where the feedback loops constitute the core element of the method  
- Claims developing software by self-organizing teams in increments, starting with planning and ending with a review  
- Team members coordinate their work in a daily stand-up meeting  
- One team member, the scrum master, is in charge of diminishing impediments of the development | [32]      |

#### 1.1.5 Characteristics in Distributed Environments

*Distributed Software Development (DSD)* is another trend, which have been becoming a common practice in today’s industry [55]. The key advantages that DSD aspires to achieve are 1) *lower cost of labor*, 2) *increase or decrease work forces without employing or laying-off*, and 3) *obtain locally not available expertise* [56].

---

18
Software development organizations are striving to blend agile development methods and distributed development to reap the benefits of both. However, agile and distributed development approaches differ significantly in their key tenets. While agile methods mainly rely on informal processes to facilitate coordination, distributed development typically relies on formal mechanisms.

In distributed agile software development, additional challenges may be observed comparing to the co-located situations [43], [44], [45]. These challenges can be categorized into 1) communication (geographical separation hinders informal interactions such as see or speak in person), 2) trust (shared view of goals are difficult to observe in dispersed locations), and 3) control (lack of negotiation between developers and customers) areas [45]. The obstacles of informal communication seem a contradiction to the ideas of agile methods [35], [37], [34], [57], [45] and seem to preclude the use of agile methodologies. Communication and coordination problems result in reduced productivity of the team, increased production interval, increased communication cost, and difficult process control across distributed teams. If a team is distributed, one solution is to minimize the effects of the informal communication deficiency.

The obstacles of informal communication can be reduced by increasing the formality (ceremony) of the interactions. In this case, the lack of communication can be (theoretically) replaced with detailed documentation (i.e. specifications, design plans, project plans) and conventions (i.e. coding standards, templates). Although, this approach reacts and contradicts to the ideas of agility, but it is a well-tried approach to aim at geographical distances.

Addition to the previous challenges (i.e. communication, trust and control), projects in distributed environments take about two and one-half times longer to complete – comparing to similar projects where the project team is co-located [45]. The significant difference is explained by the communication and coordination issues rather than the size or complexity of the cross-site development [57]. As a consequence, distributed software development requires considerable effort from the team in order to be successful. The decision makers must understand risk/reward tradeoff needs before deciding to distribute software development, because it decreases the project’s likelihood of success, increases the delivery time and quality, and reduces the team’s performance.

1.2 Planning of Software Development Environments

Industrial software development is a highly complex and dynamic task. The success of software organizations depends on the effectiveness and efficiency of development activities where development coordination takes a key role. A major problem faced by coordination is determining which features should be in the next sequence of releases. This decision making is manifested by development plans. These give the description about which features to implement in which deliveries of software systems to provide maximal business value. Although there are some process standards (e.g. CMMI, ISO 9000, ISO 12207, and ISO 9126) relating to software development, neither of them provide operational support to planning software projects, i.e. how to carry out software project planning.

Project planning are usually decomposed into a three-level hierarchy: strategic, tactical, and operational [58]. Each planning level is responsible for realizing the objectives of both the given and its superior level. Strategic-level aims at long-term decisions such as needs, make-or-buy choices, visioning, chartering and funding. Tactical-level relates to medium-term decisions such as scope, deadlines, and
capacities. Operational-level focuses on short-term decisions, including resource allocation to realization tasks. Agreed requirements drive direct planning of system development projects at tactical-, and operational-management level in addition to the implementation of the development process [59].

Capability Maturity Model-Integrated (CMMI) [60] is an effort to create a complete set of integrated frameworks for the holistic coverage of all engineering processes of an entire organization. The Gane-Sarson data-flow diagram in Figure 1.1 depicts Projects Planning (PP) with its specific goals (SG1-3) and its closely related Requirement Development (RD) and Project Monitoring and Control (PMC) process areas [60]. CMMI does not distinguish planning level hierarchies, it relates to all levels.

This process is carried out at both tactical and operational levels. While tactical level relates to a longer time period with multiple software delivery points, operational level relates to a shorter time period with a single software delivery point which aim is to deliver functioning software at a given time instance. At both levels, the planning process is made up of the following major steps: 1) feature set based estimation of project/release effort and duration (SG1), 2) setting project/release deadlines, dependencies and required capacities (Planning Data), 3) developing project/release execution plan from the data (SG2), 4) producing project plan/schedule (including critical path analysis and Gantt chart) (SG2), 5) obtaining commitment to the plan/schedule (SG3) from the project management [60].

1.2.1 Predictive and Adaptive Planning Processes

Generally, two kinds of planning approaches adhered to IDP (predictive and adaptive). The main difference between the two approaches originates in their different scheduling aspect [61]. Predictive planning produces a single detailed schedule of the software development activity based on its detailed design. So, the team works backward from the details of the software design to arrive at its schedule. It is often also called as push scheduling.

In contrast, adaptive planning produces software based on the constraints of the schedule. So, the team starts with the constraints of the system and develops the schedule to accommodate those constraints. It is often also called as pull scheduling.

There are two substantial differences between the two approaches. Firstly, the predictive one usually covers the entire software lifecycle (lookahead typically ranges from \( n \) months/years), comparing to the adaptive one which covers only a well conceivable time period ahead and the plan is adapted to the given situation from time-to-time (lookahead typically ranges 1 – 2 months). Secondly, adaptive approach locks in details just in time and therefore use no more slack (i.e. buffer) than is necessary. Comparing to the predictive one when the development is carried out according to the plan (i.e. just in case).
1.2. Planning of Software Development Environments

Typically, the predictive planning approach is good for development projects where the constraints are invariants (e.g. well known domain/customer). Whereas, the adaptive planning approach is usually appropriate for those projects where the constraints fluctuate (e.g. broad markets, not well known domain). Characteristic example of adaptive planning include agile software development.

1.2.2 Comparison of Predictive and Adaptive Approaches Using Control Theory Analogy

In the following I use control theory analogy to emphasize the differences of the predictive and adaptive approaches. The predictive approach is similar to open-loop control systems: at the beginning of the process the desired output (deliverable features: \( W \)) is defined. Then the controller (development coordinator: \( D \)) determines the necessary inputs (development planning: \( \mathfrak{P} \)) in other to realize the output (features: \( W' \)). So an open-loop control system determines the input using only the current knowledge about the desired output. It means that the system does not observe the output of the process that it is controlling. Therefore, an open-loop control system cannot engage in machine learning, cannot correct any errors that it could make, and it also may not compensate the disturbances that influence the system. Open-loop control system is useful for well-defined control systems (development situations) where the relationship between input and the desired output are well known.

Unfortunately, only one thing is usually constant in software development situations: the change. The changes are emanating from different sources, for example, changes of customer requirements, business goals, and technological demands. Consequently, these software development processes should respond to these changes by adapting its execution to the changed situation. Continuing the previous analogy, I can characterize adaptive approach as closed-loop control system: additional to the open-loop control system the disturbances (changes: \( \Delta W \)), which affect the system, are measured by different sensors (development reviews /feedbacks/: \( \phi \)) that are transferred to the input of the controller (developer coordinator: \( D \)) in other to the input of the system (development plan: \( \mathfrak{P} \)) can be adapted to the changed situation. In the adaptive approach, there are several feedback loops and all of them have different objectives (\( O \)) and constraints (\( C \)). The intuitive mapping of the adaptive approach to closed-loop control system can be seen in Figure 1.2.

![Figure 1.2: The Adaptive Approach as a Feedback Control.](image-url)
1.3 Overview of Agile Planning

Methods for agile software development constitute a set of practices that have been created by experienced practitioners. These methods can be seen as a reaction to traditional rigid plan-based methods which emphasize the ‘rationalized and engineering-based approach’. In the plan-based approach it is claimed that problems are fully specifiable and optimal and predictable solutions exist for every problem therefore follows the predictive planning strategy. It advocates extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity [62, 63].

In contrast with plan-based methods, agile processes address the challenge of an unpredictable world by relying on ‘people and their creativity rather than on processes’ [62, 63]. Agile methods generally encourage frequent inspection and adaption where features and solutions evolve through collaboration between self-organizing cross-functional teams, and these methods also promote business approach that aligns development with customer needs and company goals [37, 64]. Agile methods are a set of development processes – not a single approach to software development – that state that agile development should focus on the previously listed four core values (c.f. Sec. 1.1).

The Table 1.3 summarizes the main differences between the plan-driven and the agile development approaches [65].

<table>
<thead>
<tr>
<th>Viewpoints</th>
<th>Plan-based method</th>
<th>Agile method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fundamental assumption</strong></td>
<td>Systems are fully specifiable, predictable, and are built through extensive planning</td>
<td>Systems are developed adaptively by small teams with using the principles of continuous design improvement and testing based on rapid feedback and change</td>
</tr>
<tr>
<td><strong>Management style</strong></td>
<td>Command and control.</td>
<td>Collaboration and leadership</td>
</tr>
<tr>
<td><strong>Knowledge management</strong></td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>Formal</td>
<td>Informal</td>
</tr>
<tr>
<td><strong>Development model</strong></td>
<td>Life-cycle model (waterfall, spiral or their variation)</td>
<td>The evolutionary-delivery model (iterative and incremental)</td>
</tr>
<tr>
<td><strong>Desired organizational form/structure</strong></td>
<td>Highly formalized (bureaucratic) – typically large organizations</td>
<td>Organic (flexible and cooperative) – typically small and medium sized organizations</td>
</tr>
<tr>
<td><strong>Quality control</strong></td>
<td>Heavy planning, strict control and late heavy testing</td>
<td>Continuous control of requirements and design and continuous testing of the solutions</td>
</tr>
</tbody>
</table>

1.3.1 Agile Planning Cycles

Due to the characteristics of agile approaches, agile planning is significantly different from the traditional plan-based methods. Agile methods typically use the adaptive planning approach, therefore, it implements the closed-loop control approach. It also includes typically three kind of plans: a coarse-grained long-time (roadmap), a medium-grained short-time (release) and a fine-grained (iteration) plan [61, 66, 67]. Each planning level is responsible for realizing the objectives of both the given and its superior level. Additionally, the previously mentioned roadmap, release and iteration maps are extended with two additional levels to support the principles of continuous design improvement and continuous testing based on rapid feedbacks. These extensions result in the following five levels:
1.3. Overview of Agile Planning

These additional two levels (Daily and Continuous) aim at resolving the inherent discrepancies of operational-level plans during team’s daily meetings and during the days with informal talks. The five agile planning levels are visualized in Figure 1.3. The loops demonstrate the iterative characteristics of agile planning, and the size of the loops positively correlate with the duration of the planned time interval.

From development coordination point of view, the agile IDP process is made up of the following phases [66, 67] (Figure 1.4). (Please note, the daily loop is not shown because it is not important in the Research point of view.):

1) **Roadmapping**: to define a project roadmap and a product vision ($\pi$) that is broken down into high-level ranked deliverable features ($W$),

2) **Release planning** ($\mathcal{G}_{AR}$): to determine feature-to-iteration assignments ($X$) regarding constraints ($C^R$: effort for feature development /requirements and defect corrections/, priority of deliverables that express the importance of the features, team development capacity, and dependencies between features that influence the delivery sequence) while considering the objective of the release ($O^R$: delivering maximal value to the customer),
3. Overview of Agile Planning

Road-mapping
Release planning
Iteration planning
Iteration loop
Release loop

![Diagram of Agile Planning Cycles](image)

3) *iteration planning* ($\mathcal{S}_{AI}$): to determine task-to-developer assignments ($S$) regarding constraints ($C^I$: effort for technical task development /these are broken down from the features/, dependencies between tasks that influence the implementation sequence, and resource availability) while considering the objective of the iteration ($O^I$: minimal execution time),

4) *iteration*: to discuss the daily progress concerning writing tests, codes and fixing defects ($\Delta W$),

5) *delivery*: to package and deploy software to customers,

6) *iteration review ($\phi^I$)* and *release review ($\phi^R$)*: to demonstrate product increment or release ($W'$) to the customers and conduct iteration or release retrospective for the next iteration or next release respectively.

As agile development usually relies on personal frequent informal interactions between team members (Figure 1.4), the parameter $\gamma$ characterizes the communication and coordination needs (i.e. how many person hours are spent on communication and coordination during a given time period) of a given agile development process. This parameter is important in distributed agile development in particular.

Release planning aims at assigning software features to iterations of evolving software products. A proper release plan should satisfy customer needs while provide maximal business value by selecting the right set of features into the next iteration(s). Feature selection considers the demands of the users, managers, developers, or their representatives. As a consequence, it is often not obvious which choice is better, because several concurrent aspects must be taken into account.

Once the high priority features are selected for the next iteration, the following step is to realize them. In agile methods, the software is rolled out in increments (i.e. *staged-delivery*) over time with iterative development approach to reduce the overall risk of realization [66, 67]. Therefore, a release is made up several iterations (typically from 1 to 4 and with duration 1 or 2 weeks) which delivers intermediate features to the customers, so it receives both a sense of value and an opportunity to provide early feedback. Iteration schedule is operational level support for realization of feature development technical tasks, and it focuses on resource allocation to these tasks in iterations [67].
1.4 Motivation

The Figure 1.5 set an example of agile software delivery cycles focusing on the planning and delivery aspects. This example shows two releases, and within each release it depicts three iterations. Each release starts with a release planning activity \((S_{AR})\) and ends with a product release \((W')\) that is usually delivered to the customer’s live (production) IT infrastructure. Each iteration also begins with an iteration planning activity \((S_{AI})\) and ends with a product increment \((W')\) that is usually treated as a demo version and often delivered to the customer’s test IT infrastructure. The demo versions within a release constitute to the live version of the software.

Several reports indicated that the average cost overruns of software projects were between from 50% to 200% between 1994 and 2004 [68, 69, 70, 71]. The three of ten principal factors for ‘challenged’ and failed projects were imprecise planning, lack of proper development coordination, and changing requirements and specifications. In [70] ten principal factors for project success were identified, where a number of the success-factors (clear business objectives, development coordination, scope optimization) directly related to release and iteration planning.

In a survey [42] pointed out that 35% of the respondents worked in distributed development teams – where the members of the teams are physically dispersed (ranging from being over adjacent buildings to being over different continents). In previous research [57] it was shown that, distributed projects take about two and one-half times longer to complete as similar projects where the project team is co-located. The delays can be explained by the communication and coordination issues rather than the size or complexity of the cross-site work [57]. As a consequence, distributed development requires significant effort from the team in order to be truly successful [72].

**SEMAT (Software Engineering Method and Theory)** is an initiative to reshape Software Engineering. The initiative was launched in October 2009. SEMAT initiative believes that current Software Engineering is seriously unsatisfactory, due to lack of solid theoretical basis and existence of large number of competing methods, practices and fads. The goal of SEMAT is to ‘refound software engineering based on a solid theory, proven principles, and best practices’. A significant number of world-class experts in the field of Software Engineering endorse the initiative.

In my professional life (I have been working as a project leader and a CTO for 10 years) I faces the before mentioned problems and difficulties on a daily basis. My personal experiences also reinforced the deficiencies of the agile release- and iteration planning: such as lack of upfront planning and the lack of predictability. According to my experiences we could have been persuaded any project sponsors without
1.4. Motivation

sound project plans to fund the project (both on the customer’s and the developer’s side). Therefore, in
my practice these deficiencies were surmounted with intuitively and manually constructed project plans – in spite of the fact that this kind of planning resulted in suboptimal and sometimes contradictory plans.

1.4.1 Relevancy of the Research

Agile processes offer numerous benefits to organizations, including quicker return on investment, higher product quality, and better customer satisfaction (see Sec. 1.1.3). However, they lack a sound methodological support of agile release and iteration planning – contrary to the traditional, plan-based approaches. The previously cited survey [42] points out that the second and the fifth principal factors from the identified 26 ones are iteration and release planning respectively. Besides these facts, the survey also revealed the 13 most commonly cited greatest concerns listed by respondents about adopting agile within companies. The three out of the five most important ones are 1) the loss of management control (36%), 2) the lack of upfront planning (33%) and 3) the lack of predictability (27%). These considerations are closely connected with the present informal practice of agile planning. These critics underline the importance of providing a more established method for agile planning, that lacks of solid theoretical basis currently due to its novelty. The aim of this research is to elaborate methodical support for agile release and iteration planning in both co-located and distributed environments.

The seventh annual ‘State of Agile Development’ survey was conducted between August 1st and November 1st 2012 [42]. In the survey, respondents were recruited from web sites, mailing list. The survey data includes information from 4.048 participants worldwide and the data was analyzed and prepared into a summary report by an independent survey consultancy firm. The following Figure 1.6, 1.7 and 1.8 show some important statistics considering this research.

Figure 1.6 shows the adoption rate of agile methods considering different size of software development organizations (they include not only developers) – classified into four groups. The growing popularity of the agile approaches can be explained by the fact that agile teams are often more successful than the traditional ones (see Sec. 1.1.3). This bar chart points out the fact that the adoption rate is roughly one out of four. It also reports on two important characteristics, namely, 84% of the respondents worked in organizations that used agile development practices to some degree, and 58% of respondents worked in companies with distributed development teams. Gartner Corporation, in [73], predicts that by 2012, agile development methods will be utilized in 80 percent of all software development projects.

![Figure 1.6: Agile Adoption Rate by Different Organization Size.](image-url)
Figure 1.7 denotes the popularity of the various agile software development methods. It shows that Scrum (an iterative, incremental framework for agile project management [32]) and XP (an agile software development methodology which is intended to improve software quality and responsiveness to changing customer requirements [30]) are the most popular ones.

Figure 1.7: Followed Agile Methods.

Figure 1.8 shows that the first and the fourth important technique in agile are iteration and release planning. Besides these facts, the Management opposition to change, Lack of upfront planning, Loss of management control, and Lack of predictability are the four most important barriers to apply agile methods by software development organizations. In this research, by providing tools for iteration and release planning, my aim is to diminish these barriers – therefore it underlines the relevancy of this research.

Figure 1.8: Employed Agile Techniques.
1.4. Motivation

1.4.2 Agile Release planning

The essential aim of release planning ($\mathcal{S}_{AR}$) is to determine a feasible coarse-grained plan for the development that determines which feature ($W$) in which iteration ($I$) will be delivered ($X$ – feature-to-iteration assignments).

The optimized version of the release planning problem can be derived by selecting the extreme-valued plan from the potentially feasible alternatives. This optimal release plan can be considered as an optimization problem in which a set of items (features) are given with their value (business priority) and size (required effort), and it is desired to select one or more disjoint subsets (iterations) so that the sum of the sizes in each subset does not exceed (or equals) a given bound and the sum of the selected values is maximized.

Release planning usually addresses two kinds of typical customer questions: 1) Fixed-time question: ‘How much of the features can be delivered by a given date?’ and/or 2) Fixed-scope question: ‘When can the selected (or all) features be delivered?’ The Fixed-time question mandates date-driven, while Fixed-scope one requires scope-driven planning approach. In fact, the former one can be interpreted as a temporal constrained version of the latter one.

Without sound decision support the following typical constraint (i.e. $C^R$) are managed informally (implicitly and intuitively) by development coordinators: 1) iterations ($I$ – where features should be assigned), 2) dependencies ($D$ – temporal constraints between features), 3) resource capacities ($R$ – set of developers, $e$ – team effectiveness factor, $c$ – resource demands during releases), 4) priorities ($p$ – importance of each feature delivery), 5) effort ($w$ – effort for developing of each features), 6) release length ($l^R$), and 7) iteration length ($l^I_i$).

Therefore, optimality of plans (i.e. maximal business value) is heavily based on the manager’s right senses – nevertheless optimized project plans are crucial from technical and economic point of view. More formally, I can say that the design space of agile release scheduling is made up of the previous factors ($W$, $I$, $D$, $R$, $e$, $c$, $p$, $w$, $l^R$, $l^I_i$, $X$) – in short: ($W$, $C^R$, $X$) –, and its objective is to find an optimal mapping of

$$\mathcal{S}_{AR} : (W, C^R) \rightarrow X$$

while it considers maximal value delivery. This research proposes methodical support to this mapping in Contribution 1.

![Figure 1.9: Agile Release Planning.](image_url)
1.4. Motivation

1.4.3 Agile Iteration scheduling

Once the high priority features are selected for the next release the following step is to realize them. In agile approaches, software is rolled out in increments – in iterations – to reduce the overall risk of realization [67, 66]. Technical tasks are the main concepts of iteration scheduling. These tasks are the fundamental working units accomplished by one developer, and usually require some working hour (Wh) realization effort that is estimated by the team. The aim of iteration scheduling is to break down selected requirements into technical tasks and to assign them to developers [67].

Agile release planning and scheduling relates to to medium-term decisions such as scope, deadlines, and capacities in the next time period (cca. one-three months). In contrast, iteration planning and scheduling focuses on short-term decisions, such as planning of detailed resource allocation to realization tasks (i.e. technical tasks). In agile methods, the software is rolled out in increments (i.e. staged-delivery) over time with iterative development approach to reduce the overall risk of realization [66, 67]. Therefore, a release is made up several iterations (typically from 1 to 4 and with duration 1 or 2 weeks) which delivers intermediate features to the customers, so it receives both a sense of value and an opportunity to provide early feedback.

Generally, iteration scheduling ($\mathcal{S}_{AI}$) aims at determining a feasible fine-grained plan for the development that schedules the implementation of selected features within an iteration ($\mathbf{S}$ – task-to-developer assignments ($A$)) [67].

The optimized version of the iteration scheduling problem can be derived by selecting the extreme-valued schedule from the potentially feasible alternatives. This problem can be considered as an optimization problem in which the resource allocation consists in assigning time intervals to the execution of the activities (realization tasks) while taking into account both temporal constraints (precedences between tasks) and resource constraints (resource availability) and the minimal execution time objective.

Iteration planning is usually a complex task due to the following typical constraints (i.e. $C^I$): 1) precedences ($P$ – temporal precedences between realizations), 2) balancing resource workloads ($R$ – avoiding resources overloading), 3) effort ($w$ – effort for implementing of each task), 4) pre-assignment ($a$ – manual assignment of the appropriate developer to some tasks) and 5) iteration length ($l^I$ – deadline of iteration end to provide staged-delivery).

In traditional approaches, scheduling is usually carried out by a project scheduling software package (e.g. [74]) that helps coping with constraints and objectives. Since traditional approaches require manual construction and take relatively long time (several hours) they are too heavyweight for IDP (particularly in agile environments) therefore often omitted. However without adequate decision-support they are managed informally (implicitly and intuitively) whose inherent discrepancies must be resolved during team’s meetings (see Figure 1.4) [61, 67]. Informal approaches work well in smaller projects, however as the size and complexity increases scheduling becomes a very complex process and advocates tool support [75, 76]. More formally, I can say that the design space of agile iteration scheduling is made up of the previous factors ($A, P, R, w, a, l^I, S$) – in short: ($A, C^I$) –, and its objective is to find an optimal mapping of

$$\mathcal{S}_{AI} : (A, C^I) \rightarrow S$$

(1.2)
1.4. Motivation

while it considers minimal execution time objective. This research proposes methodical support to this mapping in Contribution 2.

\[
C^I
\]

\[
\begin{array}{ccc}
\mathcal{A} & \xrightarrow{\text{Iteration planning} \ (\mathfrak{G}_{AI})} & S
\end{array}
\]

**Figure 1.10:** Agile Iteration Planning.

1.4.4 Distributed Extension of Agile Release Planning

Distributed extension of agile release planning (AF) can be defined as a decision making process, where the goal is to determine an feasible feature chunk assignment to each distributed team before the local agile release planning is carried out. Cohesive feature chunks are usually constructed by considering the module relatedness, the development competence, etc. of features. In this thesis, the elaborated feature chunk construction considers the communication and coordination needs (i.e. how many person hours are spent on communication and coordination during a given time period) between the dispersed teams during implementation.

The optimized version of the distributed extension of agile release planning problem can be derived by selecting the extreme-valued plan from the potentially feasible alternatives. This problem can be considered as an optimization problem that requires finding a set of edges (communication paths) between vertices (teams) whose removal would partition the graph into connected components (feature partitions) [77]. The minimization objective reflects the aim of minimizing the intensity of communication and synchronization needs (see Figure 1.4) in order to minimize their negative effects – such as reduced team productivity, increased production interval, increased communication cost and difficult process control across distributed teams [37, 34, 57, 45].

Feature chunks are usually identified according to their cohesiveness from the standpoint of architectural impact. The higher the cohesiveness between features, the stronger the need to group those features together. To identify cohesiveness between features, I introduce a binary relation between features (W) and software modules (M), called ImplementedIn (⊗), to express the fact that a given feature is implemented in a given software module. My goal is to group those features which are to be implemented in the similar set of software modules, i.e. they require similar architectural impact. With this approach, arranging development work (features) according to the identified feature chunks, it can significantly decrease the communication needs and coordination complexity of the distributed team (T). Traditionally, the feature chunk composition is manually accomplished that requires relatively long time (several hours) and the optimality objective (partitioning the features into k cohesive chunks) hardly can be realized. More formally, I can say that the design space of feature distribution is made up of the previous factors \((W, M, T, \otimes, W^*)\), and its objective is to find an optimal \(k\)-partitioning of

\[
\mathfrak{A}_F : (W, M, T, \otimes) \rightarrow W^* \tag{1.3}
\]
1.5. Existing Approaches

This research proposes methodical support to this mapping in Contribution 3.

\[
\mathcal{M}, \mathcal{T}, \otimes \\
\downarrow \\
\mathcal{W} \xrightarrow{\text{Feature partitioning}} \mathcal{W}^* 
\]

**Figure 1.11**: Distributed Agile Release Planning.

1.5 Existing Approaches

The following subsections provide an overview to the existing approaches to planning in both co-located and in distributed environments.

1.5.1 Requirements Modeling

Requirements models are used when gathering requirements, and during systems analysis. Requirements models are used to discover and clarify the functional and non-functional requirements for software and business systems. Moreover, the requirements models are used as specifications for the designers and builders of the system. In the following I present those techniques that are used in this dissertation.

1.5.1.1 Use Case-based Requirements

In software engineering, client-valued functions are typically expressed with scenarios. Scenarios help in communicating with the customer and improve mutual understanding of the system to be developed. Use cases hold together scenarios with a simple notation helping in both drive the whole development process and provide traceability of requirements’ realization [78, 71]. *Use case diagrams* express scenarios in an easily understandable visual form [79].

The core concepts of Use case diagrams are associated with actors and use cases. Actors model entities that are outside of the system (users/other systems), and system’s required behavior is specified by one or more use cases.

Use cases provide constructs for two main categories of concepts that represent usage scenarios of the system: 1) behavioral elements, and 2) behavioral links. Behavioral elements model ‘what’ of requirements at a very high level with the following basic elements:

- **Actor**: a role played by a user/other system that interacts with the subject
- **Use case**: a set of actions performed by the system

Behavioral links specify different relationships:
1.5. Existing Approaches

- **Association**: communication path between an actor and a use case
- **Extend**: inserts additional behavior into a base use case that is unaware of the extension
- **Include**: inserts additional behavior into a base use case that explicitly describes the insertion
- **Generalization**: generalizes and determines relationships between model elements

There are two complementary elements to complete the high level Use case specification: 1) textual description (e.g. Cockburn’s form [80]), and 2) UML interaction definition (e.g. Activity, Sequence diagrams) to specify the dynamic’s aspect of the system [29, 80]. These elements are not relevant according to my high level modeling aspect.

UML 2.0 has 13 types of diagrams that can be categorized into i) **behavior models**: to represent dynamic aspects of systems (such as Use cases, Activity), and ii) **static models**: to provide the collection of components and relations among them (such as Class and Component). The use case diagram captures the behaviors of the system as it appears to an outside user [79] and partitions the functionality into a set of actors and use cases. There are many ways of expressing scenarios, such as Activity-, Sequence Diagram in UML [79], or Use Cases Maps [81, 82], to express fairly detailed design (how aspect). The reason why I have selected Use case is that project estimation and planning must be carried out at the early requirements’ phase.

1.5.1.2 Goal-oriented Requirements Modeling

Scenario-based specifications describe possible ways to use a system to accomplish some desired functions or implicit purposes [83, 82]. **Scenarios** are operational examples of system usage, they can help to describe (what) and understand (how) emergent behavior of complex and dynamic systems. The UML Use cases [79] express scenarios, their intuitive graphical notation facilitates communication with users and/or customers and a popular documentation form for intended system behavior at high level. Since Use cases determine client-valued functions, they both drive the whole development process and provide traceability of realization [29, 78, 84], hence they also enable documenting required system behavior in contractual agreements [59].

As a complementary technique, goal-oriented requirements modeling has been proposed for a number of years and several approaches have been published [83, 85]. **Goals** are intentions (why aspect) that are expected to be fulfilled by the system [85]. The GRL is built on the well established NFR (Non-functional Requirements) framework [86]. GRL enables modeling business and system goals, (non-functional) requirements, alternatives, and decision rationales [87]. The goal-scenario combined approach complements each other and facilitates decision-making from early requirements to detailed design [83, 82].

Requirements from various users and/or customers are often expressed as objectives or goals. Directly modeling goals, rather than entities, allow searching for alternatives before deciding a particular solution. Moreover, goal representation enables validating requirements’ completeness with tasks and scenarios. GRL is a graphical language for supporting goal-oriented modeling and reasoning, especially at non-functional requirements [87].

GRL provides constructs for three main concept categories that are used: i) intentional elements, ii) intentional links, and iii) actors [82].
1.5. Existing Approaches

Intentional elements model the ‘why’ of the requirements with the following basic concepts:

- **Goal**: quantifiable (functional) objective
- **SoftGoal**: qualifiable (non-functional) objective
- **Task**: operationalized solution that *achieves* a goal or *satisfies* a SoftGoal
- **Resource**: entity described in terms of its availability
- **Belief**: design assumption and environmental factors

The following link types can connect intentional elements:

- **Contribution**: how SoftGoals, Tasks, Beliefs, and relations *contribute* to each other, that can be qualified by a degree: equal, break, hurt, some+/-, help, or make
- **Means-end**: how Goals are in fact achieved
- **Decomposition**: what other sub-elements need to be achieved in order to a Task can be performed
- **Dependency**: two actors depending on each other
- **Correlation**: knowledge about interactions between elements

Finally, the last category contains:

- **Actor**: an active entity that carries out actions to achieve goals by exercising its know-how

There are several mature tools that support GRL modeling and analysis, such as OpenOME [88] and jUCMNav [89]. They also support XML model export in order to increase easy integration with other tools.

### 1.5.2 Dependencies between Requirements

The complexity of scheduling arises from the interaction between requirements by *implicit* and *explicit* constraints. While the previous is given by the scarcity of resources, the latter one is emerged from different dependencies between requirements. The main sources of dependencies are identified in [90] and summarized in Table 1.4. To clarify the meaning of dependencies, the following examples are given in Table 1.5. (Note, $j', j$ denote requirements):

<table>
<thead>
<tr>
<th>Dependency name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>$j$ and $j'$ jointly serve a unit of functionality</td>
</tr>
<tr>
<td><strong>REQUIRES</strong></td>
<td>$j$ builds on functionality $j'$ to function</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td>a function can be served by either $j$ or $j'$</td>
</tr>
<tr>
<td><strong>VALUE</strong></td>
<td>$j'$ influences the customer value of $j$, so rational to realize $j'$ earlier</td>
</tr>
<tr>
<td><strong>ICOST</strong></td>
<td>$j'$ influences the implementation cost of $j$, so rational to realize $j'$ earlier</td>
</tr>
<tr>
<td><strong>TEMPORAL</strong></td>
<td>technological/organizational constraints between $j$ and $j'$, so it is rational to realize $j'$ earlier</td>
</tr>
</tbody>
</table>

Theoretically, there can be $C_n^2 = 6n(n-1)/2 = 3n(n-1)$ number of dependencies (where $n$ denotes the number of requirements, and 6 the number of dependency types). Generally, the number of dependencies is dependent on the case, and in [90] it was found that this number is roughly between $(n, 2n)$.  

33
1.5. Existing Approaches

<table>
<thead>
<tr>
<th>Dependency name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>A customer report functionality needs a report visualization component ((j)) (e.g. Crystal Reports [91]) and a data query ((j')) to function.</td>
</tr>
<tr>
<td>REQUIRES</td>
<td>An email alert function ((j)) requires that the given users to be stored with their appropriate attributes ((j')) in the system.</td>
</tr>
<tr>
<td>XOR</td>
<td>A customer report visualization function may be based on Crystal Reports component ((j)) xor on an in-house solution ((j')).</td>
</tr>
<tr>
<td>CVALUE</td>
<td>A well-organized user interface (e.g. with determined form fill in sequence) ((j')) usually adds value to the user experience and diminishes the need of detailed user documentation ((j)).</td>
</tr>
<tr>
<td>ICOST</td>
<td>Implementing a configurable report functionality ((j')) often adds extra cost to the implementation, but it may ease the development of data queries ((j)).</td>
</tr>
<tr>
<td>TEMPORAL</td>
<td>Extending the customer module ((j')) should be implemented before the development of the customer report ((j)) since the previous may effect on the report contents.</td>
</tr>
</tbody>
</table>

1.5.3 Effort estimation

Effort estimation is an active research in the software industry, since proper planning is not possible without reliable estimations. Generally, in software effort estimation, collected principal factors of past projects provide a comparison with the project under estimation. The same factors of the estimated project are used to extrapolate the questioned effort size with the predictive rules of the given estimation method.

Significant research on software effort modeling began with the extensive SDC study of the 104 attributes of 169 software projects[92]. The 1980s and 1990s produced robust parametric models which use of a formula derived from historical data. Perhaps the most common parametric models are COCOMO [93] and SLIM [94]. The estimation approaches based on functionality-based size measures are also based on research conducted in the 1970s and 1980s. The most common models approaches are Function Point Analysis (FPA) [95] and Use case Points [96, 97]. Function points are also the units of measure used by the IFPUG Functional Size Measurement Method[98]. The IFPUG FSM Method is currently an ISO recognized standard (ISO/IEC 20926 Software Engineering - Function Point Counting Practices Manual) for functionally sizing software. In [99], a cost estimation method is proposed with a special emphasis on the cost estimation driven system design. This COCOMO-based method helps to analyze solution alternatives in dependable system design by integrating effort prediction with technical design activities.

In the following, two effort estimation models are presented that are used in my solutions later.

1.5.3.1 COCOMO II

The Constructive Cost Model II (COCOMO) is an algorithmic software cost estimation model that uses a basic regression formula with parameters that are derived from historical project data and current project characteristics [100]. COCOMO computes software development effort (and cost) as a function of program size expressed in estimated KSLOC (thousands of source lines of code). In COCOMO, effort is expressed as \textit{person months}: the amount of time one person spends working on the software development project for one month. The equation of COCOMO can be seen in Eq. 1.4.

\[
P_{\text{adjusted}} = A \times (Size)^{0.91+0.01s} \sum_{i=1}^{s} W_i \times \left( \prod_{i=1}^{m} EM_i \right)
\] (1.4)
1.5. Existing Approaches

In the equation above, 1) the factor $A$ shows the project productivity factor, 2) the factor $\text{Size}$ denotes the size of software development in KSLOC, 3) the exponent of the $\text{Size}$ stands for the scaling drivers that have exponential effect on the project effort (and includes the five project scale factors ($W_i$) – see Table 1.6), and finally 4) the last factor involves the cost drivers that have a multiplicative effect on predicting effort (and involves the seven or seventeen effort multipliers ($EM_i$), where the number of multipliers depends on the phase of the project (Early design: $m = 7$ or Post architecture: $m = 17$) – see Table 1.7).

**Table 1.6: COCOMO II Project Scale Factors ($W_i$)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_1$</td>
<td>Precedentedness</td>
</tr>
<tr>
<td>$W_2$</td>
<td>Development flexibility</td>
</tr>
<tr>
<td>$W_3$</td>
<td>Architecture/Risk Resolution</td>
</tr>
<tr>
<td>$W_4$</td>
<td>Team Cohesion</td>
</tr>
<tr>
<td>$W_5$</td>
<td>Process Maturity</td>
</tr>
</tbody>
</table>

**Table 1.7: COCOMO II Effort Multipliers ($EM_i$) (Post Architecture Phase)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EM_1$</td>
<td>Required Software Reliability</td>
</tr>
<tr>
<td>$EM_2$</td>
<td>Database Size</td>
</tr>
<tr>
<td>$EM_3$</td>
<td>Documentation Match to Lifecycle Needs</td>
</tr>
<tr>
<td>$EM_4$</td>
<td>Product Complexity</td>
</tr>
<tr>
<td>$EM_5$</td>
<td>Required Reusability</td>
</tr>
<tr>
<td>$EM_6$</td>
<td>Execution Time Constraint</td>
</tr>
<tr>
<td>$EM_7$</td>
<td>Main Storage Constraint</td>
</tr>
<tr>
<td>$EM_8$</td>
<td>Platform Volatility</td>
</tr>
<tr>
<td>$EM_9$</td>
<td>Analyst Capability</td>
</tr>
<tr>
<td>$EM_{10}$</td>
<td>Applications Experience</td>
</tr>
<tr>
<td>$EM_{11}$</td>
<td>Programmer Capability</td>
</tr>
<tr>
<td>$EM_{12}$</td>
<td>Platform Experience</td>
</tr>
<tr>
<td>$EM_{13}$</td>
<td>Language and Tool Experience</td>
</tr>
<tr>
<td>$EM_{14}$</td>
<td>Personnel Continuity</td>
</tr>
<tr>
<td>$EM_{15}$</td>
<td>Use of Software Tools</td>
</tr>
<tr>
<td>$EM_{16}$</td>
<td>Required Development Schedule</td>
</tr>
<tr>
<td>$EM_{17}$</td>
<td>Multisite Development</td>
</tr>
</tbody>
</table>

The rating of the previous factors ($W_i$ and $EM_i$) can range from Extra Low to Extra High choosing from seven different values. Since, in the beginning of a project, it is very difficult to estimate the size of the deliverable software in KSLOC, there are complementary techniques in COCOMO to provide conversion between different measures such as Unadjusted Function Point [98] to KSLOC. Additionally, there are some model extensions to COCOMO, such as quality model extension (COQUALMO), or rapid application development extension (CORADMO).

The detailed description of the model can be found in [100]. There are also some tools that support COCOMO-based estimations (e.g. COCOMO [101]).

1.5.3.2 Use Case Point Method

The *Use Case Point Method (UCPM)* is more adequate and easy in Use case-based development, since effort is measured by counting the number of actors and transactions included in the specification [102]. In UCPM, system size is expressed in Unadjusted Use Case Point ($UUCP$), where actors and use cases are weighted with their realization complexity (Eq. 1.5):
1.5. Existing Approaches

\[
UUCP = \sum_{i=1}^{n} AW_i + \sum_{i=1}^{m} UCW_i \quad (1.5)
\]

Actor and use case weight values \((AW_i \in [1, 3]; UCW_i \in \{5, 10, 15\})\) are assigned to each actor and use case. To estimate size of development, \(UUCP\) is assessed by 13 technical \((T_i - \text{such as code reusability, system distributivity})\) and 8 environmental \((E_i - \text{such as motivation, experience})\) factors to characterize the difficulty of building and to determine the efficiency of project execution, respectively. These factors are listed in Table 1.8 and 1.9.

**Table 1.8: Technical Complexity Factors \((T_i)\)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Distributed system</td>
</tr>
<tr>
<td>T2</td>
<td>Response/throuput performance objectives</td>
</tr>
<tr>
<td>T3</td>
<td>End-user efficiency</td>
</tr>
<tr>
<td>T4</td>
<td>Complex internal processing</td>
</tr>
<tr>
<td>T5</td>
<td>Reusable code</td>
</tr>
<tr>
<td>T6</td>
<td>Easy to install</td>
</tr>
<tr>
<td>T7</td>
<td>Easy to use</td>
</tr>
<tr>
<td>T8</td>
<td>Portable</td>
</tr>
<tr>
<td>T9</td>
<td>Easy to change</td>
</tr>
<tr>
<td>T10</td>
<td>Concurrent</td>
</tr>
<tr>
<td>T11</td>
<td>Includes security features</td>
</tr>
<tr>
<td>T12</td>
<td>Provides access for third parties</td>
</tr>
<tr>
<td>T13</td>
<td>Special user training facilities are requirement</td>
</tr>
</tbody>
</table>

**Table 1.9: Environmental Complexity Factors \((E_i)\)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Familiarity with Unified Process</td>
</tr>
<tr>
<td>E2</td>
<td>Application experience</td>
</tr>
<tr>
<td>E3</td>
<td>Object Oriented experience</td>
</tr>
<tr>
<td>E4</td>
<td>Lead analyst capability</td>
</tr>
<tr>
<td>E5</td>
<td>Motivation</td>
</tr>
<tr>
<td>E6</td>
<td>Stable requirements</td>
</tr>
<tr>
<td>E7</td>
<td>Part-time workers</td>
</tr>
<tr>
<td>E8</td>
<td>Difficult programming language</td>
</tr>
</tbody>
</table>

The adjusted \(UCP\) is calculated using the Eq. 1.6:

\[
TFactor = \sum_{i=1}^{13} FW_i \times T_i \\
EFactor = \sum_{i=1}^{8} FW_i \times E_i \\
TCF = 0.6 + (0.01 \times TFactor) \\
EF = 1.4 + (-0.03 \times EFactor) \\
UCP = UUCP \times TCF \times EF \quad (1.6)
\]

A weight value \((FW_i \in [0, 5])\) is assigned to each component of the factors, where the value of 0 means irrelevant and 5 means essential.

The total person-hour \((TPH)\) effort is computed by the multiplication of mean resources needed per \(UCP\) \((MR)\):

\[
TPH = UCP \times MR \quad (1.7)
\]
1.5. Existing Approaches

In [96] a factor of 20 staff hours per use case point is proposed to MR, while others showed that effort can range from 15 to 30 [103]. Since MR denotes organizational productivity, it must be adjusted before using the method.

There are several industrial tools to support use case point counting and estimation (e.g. Enterprise Architect [104], MagicDraw [105]) by extracting actors, use cases and related factors from Use case diagrams.

1.5.4 Priorization

Projects usually have more candidate requirements than can be realized within the time and resource constraints of the release. Priorization helps to select the more important requirements from the set of candidates to support release-centered decisions.

Basically, there are two different techniques to prioritize requirements. The **absolute priorization technique** (e.g. MoSCoW rules [67]) determines the importance of requirements without comparing to any other requirement, while the **relative priorization technique** is based on the expressed relative importance among requirements. Relative techniques tend to be more accurate and informative than absolute ones [106]. One relative technique is the $100-test presented in [84] to assign a fixed amount of units among all requirements by different users or customers in order to construct relative weighted list of requirements. The **Wiegers’ method** calculates the priority of a requirement by dividing the value of a requirement by the sum of the costs and technical risks associated with its implementation [107]. The requirement’s value depends on the business value provided to the customer on a scale from 1 to 9. Karlsson’s pair-wise comparison technique [106] is based on the Analytical Hierarchy Process (AHP) [108, 109]. In this technique, all requirement pairs are compared according to their importance on the same scale that is employed in AHP: 1 (equal); 3 (moderate difference); 5 (essential difference); 7 (major difference), and 9 (extreme difference). Then these comparisons lead to the understanding of each requirement value regarding to the total value of the requirements. In [110], an improved technique of the Karlsson’s pair-wise comparison technique is introduced by using cost and value as high-level factors against which each requirement pair is compared. The **IFM method** provides insight into the impact of development decisions with a financially-informed approach to maximize Net Present Value (NPV) [111]. Finally, in [112], values to each requirement is assigned then formulated selection of requirements as BINARY KNAPSACK problem.

1.5.5 Release planning

Compared to the extensive research on requirements prioritization only few researches dealt with release planning – which can be explained by the relatively late general usage of Iterative Development Process (IDP). In [90], release planning is formulated as an INTEGER LINEAR PROGRAMMING problem (ILP), and in [113] as a BINARY KNAPSACK problem where dependencies between features were treated as constraints. In [75], the ILP planning approach with users’ and/or customers’ opinions is extended. In [114], the previous ILP approach is further extended with some managerial steering mechanism that enabled what-if analysis. In [115], the selection and scheduling is combined to provide feature selection and on-time-delivery project plan simultaneously. There are also techniques aimed at release planning, in particular when several users and/or customers are involved, such as **EVOLVE** [116] and
Quantitative WinWin [117]. In [118], utilized Bayes belief network is presented to take uncertainty of effort for resource decisions into consideration. In [119], a method to focus on schedule level minimization is proposed, but it did not examine different productivity level of developers. In [120] a two-phased optimization approach is proposed that combines the planning and resources allocation tasks of one release. This method assumes that each feature is decomposed into a sequence of tasks (e.g. design, implementation and testing) in release planning time. The second phase, which can be used alone in small problems, performs an unfocused search to allocate tasks to developers by formulating the problem as a special case of Job Shop Scheduling Problem and this problem is solved with a genetic algorithm. The optional first phase is used to reduce the search space by formulating the problem as a binary knapsack decision problem. The advantage of the first phase becomes considerable when the feature count is relatively large (> n * 10).

1.5.6 Distributed Software Development

Software outsourcing is an increasingly attractive business model for many large organizations. In [55] three outsourcing strategies are presented to maximize business value. In [121] good practices are presented that were observed in a very large (5,000 engineers) globally distributed development situation at Alcatel. Besides cross-locations, differences in culture and language also results in low progress in globally distributed environments. To cope with these issues, in the literature, some strategies are proposed including the use of straddlers (technical or managerial liaisons) [55], bridgehead teams [122], or rotation of management [121].

In [123] a method is offered to calculate the degree of relatedness of the work items at different sites using code change history. The calculated relatedness is used to distribute work in a way that minimizes the need for coordination across sites. In [124] experiences of a rapid production process are described using software components suited for distributed development in a large, geographically distributed situation. In this approach, each component can be owned by a particular site to promote independent work and to minimize the need of coordination and communication.

Comparing to the extensive research on distributed software development in general, only few research dealt with DSD in the agile environment in specifically [44]. Lately Scrum [32], an agile management practice, has gained considerably popularity. Experiences and practices of the adoption of Scrum by large companies such as Yahoo! or Microsoft is presented in [125], and in [126] respectively. In [72] experiences and proven practices to address challenges faced by geographically distributed agile teams are presented by the Microsoft’s Patterns & Practices group. It pointed out that the decision makers must understand risk/reward tradeoff needs before deciding to distribute software development, because it decreases the project’s likelihood of success, increases the delivery time and quality, and reduces the team’s performance.

1.6 Overview of the Research

The growing pressure to reduce costs, time-to-market and to improve quality catalyzes transitions to more automated methods and tools in software engineering to support project planning, scheduling and decisions [127]. Although, there are some tenets to manual agile planning [34, 67] algorithmic solution could not be found due to the relatively novel agile development approach to software engineering.
1.6. Overview of the Research

1.6.1 Open Problems and Differences in Agile Software Development

All methods from Sec. 1.5.3 to Sec. 1.5.5 relate to requirements prioritization, selection and scheduling aspect of non-agile (traditional) methods. However, agile development is fundamentally different from the traditional approach in terms of process execution and planning lookahead that impede the application of traditional automated methods in agile environments.

Although each agile iteration involves a full software development cycle – including planning, requirements analysis, design, coding, unit testing, and acceptance testing – the process sequence (sequence of activities) are not carried out linearly but in parallel (they repeated continuously) to provide quick adaption to the customers’ needs and the main goal is to satisfy customers by rapid delivery of useful software.

An other significant difference is the planning lookahead. The planning lookahead of traditional methods typically start from 3 months. In contrast, agile planning lookahead typically end up to 3 months. Therefore, an agile team will have difficulty describing what features are planned for the entire length of the development process. Instead, agile methods focus on adapting to changing realities quickly (see feedback loops in Sec. 1.3). When the customers’ needs change the agile team changes as well.

Additionally to the process sequence and planning lookahead differences, the major development activities are performed also differently (activity performance):

- **Planning**: agile methods break deliverables into small iterations that minimizes overall risk and allows quick adaption to changes. Iterations are short time frames (timeboxes) that typically last from one to four weeks.

- **Requirements analysis**: agile methods advocate ‘just enough’ requirements specifications where the fundamental issue is the communication, not the documentation [32, 128]. In agile methods, usually, there are no deliverable software requirement specifications (SRS) or system design documents (SDD) but lightweight specifications. Agile methods consider the SRS and SDD documents as useless deliverables that doesn’t add value.

- **Design**: agile approaches encourage starting with the simplest solution. Extra functionality can then be added later that may lead to refactoring the code. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed. Therefore, agile methods focus on ’good enough’ (not perfect) solutions.

- **Coding**: agile coding can also be used to figure out the most suitable solution to the customers’ problems. Coding can also help to communicate developer thoughts to customers to get rapid feedbacks. Feedback is critical to learning and making changes.

- **Unit testing**: determines whether a given feature works as intended. Writing automated unit tests is usually the part of the specification and design activity and carried out before feature implementation. Passing the tests confirms correct behavior as developers evolve and refactor the code. This reversed process execution attitude is often named as test-driven development.

- **Acceptance testing**: verifies that the requirements as understood by the programmers satisfy the customer’s actual requirements. Agile welcomes changing requirements, even late in development.
1.6.2 Objectives of the Research

Considering the differences of traditional and agile software process execution (i.e. process sequence, planning lookahead, activity performance), the planning aspect of decision support of agile release and iteration scheduling are important and up-to-date research areas since the informal planning and scheduling approaches are highly labor-intensive and error-prone. Moreover, even a minor modification on the input data (e.g. requirements, constraints and objectives) may require rewriting the complete plan or schedule – although the perpetual changes of these data is a basic characteristic of agile environments.

To address this situation, the goal of my research was to support decisions at release and iteration planning levels. In the dissertation, I present methods, models and algorithms relating to release and iteration planning of agile software development projects. The research objectives of my work were the following ones:

RO1 Improve the productivity of agile software development planning by introducing interactive, semi-automated methods and tools in different phases of the development process.

RO2 Reduce cognitive complexity of agile software development planning to resolve complex decision situations easily by formulating mathematical models and tools.

RO3 Improve the quality of agile software development planning to provide lower level risks by considering all major planning factors (e.g. dependencies, capacities) in mathematical optimization models.

RO4 Support decisions in agile software development planning to tailor the best plan for the specific project context and users’ and/or customers’ feedbacks by altering constraints, capacities and priorities.

RO5 Improve communication and coordination efficiency of distributed agile software development teams by introducing semi-automated tools and methods.

1.6.3 Research Methodology

The introduced objectives determined the direction of my research. The first step of the research was to explore the decision space of agile project planning. For this reason I have constructed a consistent ontology-styled information model to specify the components of this decision space. This model involves the main concepts and their relations.

Next, I have formulated three combinatorial optimization problems to model the three levels of agile project planning. These formulations provide improved efficiency and effectiveness on the different decision levels. The main novelty of my approach lies in the mathematical precise formulation of the problems.

Third, I have developed algorithms that solve the formulated optimization problems. The algorithms were implemented as a part of my PROPAS™ (Project Planning and Scheduling) Matlab toolbox. The components of this toolbox are separately published¹ under the MIT licence² that makes the free access, reuse and validation possible.

¹https://bitbucket.org/aszoke/matlab or via http://www.kese.hu/
²The MIT licence http://opensource.org/licenses/MIT
1.7. Structure of the dissertation

Finally, I validated my agile project planning approaches. To support the experimental validation, I developed an MS SharePoint-based tool – named SERPA™ – according to the elaborated agile conceptual model. This tool was used to collect data for simulations from different real development projects. The tool has been used in everyday basis for many years at a software developer company for project planning and monitoring.

I used two validation types to check, establish and reinforce the findings of the contributions by analyzing them from multiple perspectives – this approach is known as triangulation. One of the validation types was a real life experiment (small number of real contexts) and the complementary one was a laboratory experiment (large number of artificial contexts) [129].

The real life experiment helped to validate the contributions at a software developer company in real agile software development contexts. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical.

On the other hand, the laboratory experiment helped to generalize the findings of the former experiments by investigating representative large number population of generated problems (120, 360 and 480 different cases). In order to model the typical agile development situations I carefully generated the data sets in terms of problem complexity factors (such as features, team sizes, temporal dependencies) and environmental factors (such as team distribution) that constitute the parameters of the agile planning problems. The values of these factors were used as inputs during the laboratory experiment (simulation).

In order to determine the possible values for the problem complexity factors (parameters) I used the results of literature reviews ([37, 39, 38, 126, 130]), surveys ([66, 42, 131, 132, 133]), conversations with agile practitioners (especially at Agile Hungary meetups3) and my personal experiences.

The two kinds of validation types were carried out using three well-known scientific methods:

1. **Post mortem analysis**: I extracted representative data sets from a software development company in order to make a comparison between the algorithmic method and the manual planning. The goal of the analysis was to compare the manual and the optimized approaches using the same input variables.

2. **Case study**: I applied my algorithmic approach parallel with the manual approach in a real-life pilot software planning situation to make comparison between the approaches. For the study a special integrated scheduling tool, named PYTHIA PROJECT PLANNER™ prototypic software, was also implemented with the UML2 technology on the Eclipse platform.

3. **Simulation**: I carried out simulations on numerous generated representative data sets – by varying parameters of the scheduling problem – to get an insight into the performance and quality of the presented approach and to filter out the statistical staggering of different agile planning problems.

### 1.7 Structure of the dissertation

The main contribution of the thesis lies in the design, application and validation of an Integrated Agile Planning Approach (IAPA). The elaborated IAPA supports development coordinators to cope with the complexity and the dynamics of software development by providing mathematical models and algorithms

---

1.7. Structure of the dissertation

for agile release planning and iteration planning. The presented approach is different from the existing approaches in two aspects:

1. **The presented approach proposes a mathematical precise formulation of agile planning and scheduling problems.** The problem formulation realizes managerial (resource and timing data) and software engineering (deliverable requirements and defect corrections) information fusion to form a common, consistent repository for both development and coordination. This consistent repository can collect all necessary information for release and iteration planning of software projects. Therefore, it can provide a solid basement of scheduling decisions in both co-located and distributed environments.

2. **The presented approach proposes a mathematical optimization methods to support semi-automatic plan generations that is based on the repository.** The provided algorithms support decision making in complex planning situations, gives the development process increased visibility, assists in doing what-if analysis, and it can also provide adaptation support considering changes necessitated by shifting development priorities.

I formed three contributions relating to the previous research objectives (Sec. 1.6.2). **Contribution 1** and **Contribution 2** draw up my novel concepts, models and algorithms for agile release planning and iteration planning respectively; finally **Contribution 3** presents my method to extend the distributed agile release planning to distributed environments. All of my contributions constitute to the building blocks of the Integrated Agile Planning Approach (IAPA). In Figure 1.12, my contributions are summarized within the agile software development lifecycle to provide a visual overview of the research: **Contribution 1** refers to agile release planning, **Contribution 2** relates to agile iteration planning, and **Contribution 3** refers to agile feature distribution in distributed development environments. This figure, continuing the previous control theory analogy, points out the inputs and outputs of the different planning steps. These inputs and outputs are detailed in the following chapters.

The rest of the dissertation arranged as follows: Chapter 2 presents theoretical background information for the contributions; Chapter 3–Chapter 5 outlines the Contributions 1–3 respectively; and finally Chapter 6 concludes the research.
1.7. Structure of the dissertation

Integrated Agile Planning Approach (IAPA)

Road-mapping (Feature partition).

Release planning (Iteration planning).

Release review (φR)

Iteration review (ϕI)

Delivery

Iteration loop

Release loop

Contrib. 1

Contrib. 2

Contrib. 3

Product increment / Product release (W)

Changes (ΔW)

Objectives, Constraints (O, C)

Desired product (W)

Realization tasks (A)

Release plan (X)

Contributions in the Context of Agile Planning Cycles.

Figure 1.12: Contributions in the Context of Agile Planning Cycles.
Chapter 2

Discrete Mathematical Background

Discrete mathematics has increased dramatically within the last few decades. The word ‘discrete’ is used in the sense of ‘separated from each other’, the opposite of ‘continuous’. Discrete mathematics usually includes set theory, enumeration, number theory, graph theory, discrete optimization, linear and abstract algebra, data structures and algorithms.

The goal of this chapter is to present the discrete mathematical background for the research only and not to introduce the whole subject thoroughly. The chapter provides essential definitions, core notions and common notations that are important according to the research point of view.

2.1 Discrete Optimization

Optimization is a unifying paradigm in almost all economic analysis [134]. The Figure 2.1 provides a general representation of the range of discrete optimization problems relating to this research. The different branches of discrete optimization are closely intertwined since many combinatorial optimization problems can be modeled as integer programs (e.g. shortest path). Conversely, a combinatorial interpretation can also be given to integer programs.

Discrete optimization is a branch of optimization in applied mathematics and computer science. These optimization problems can be defined as follows.

Definition 2.1.1 (Discrete optimization [134]). For a given finite set $S$ and a given function $c : S \rightarrow \mathbb{R}$, one has to find a solution $s^* \in S$ with $c(s^*) \leq c(s)$ for all $s \in S$.

2.1.1 Outline

The rest of the chapter is arranged as follows: Sec. 2.2 and Sec. 2.3 very briefly introduces combinatorial optimization and integer programming; Sec. 2.4 presents the main definitions of knapsack problems; Sec. 2.5 introduces the major notions of scheduling problems; and Sec. 2.6 points out the main definitions of graph partitioning. The definitions and concepts of Sec. 2.4, Sec. 2.5, and Sec. 2.6 directly relate to the contribution C1, C2, and C3 respectively.
2.2 Combinatorial Optimization

Combinatorial optimization (CO) has its roots in combinatorics, operations research, and theoretical computer science [135]. The most CO problems can be formed naturally in terms of graphs and as (integer) linear programs. The classically studied topics in this field are: minimum spanning trees, shortest paths, network flows, matchings and matroids [135].

Combinatorial optimization problems are discrete optimization problems that have two important characteristics: i) the set of feasible solution is finite, and ii) there is an objective function (goal function) which has to be minimized (or maximized). A CO problem is solved if a feasible solution with a minimal (or maximal) value is found. Though, the set of feasible solutions is finite, it might turn out that the set is ‘too big’. As a consequence, the complete enumeration of possible solutions is usually practically impossible as it would take an inadmissible amount of computation time and/or memory.

The number of feasible solutions of a problem P on the length of input might be polynomial, as well as exponential. In the latter case, we are not able to find an algorithm for problem P with polynomial dependence. Typically, problems with exponential dependence are much heavier to solve than the problems with polynomial dependence. If there are several algorithms with different degrees of the polynomial, then the algorithm with the smallest degree is preferable, since it takes less computational time.

2.3 Integer Programming

LINEAR PROGRAMMING (LP) is a technique for optimization of a linear objective function, subject to linear equality and linear inequality constraints. Informally, linear programming determines the way to achieve the best outcome (such as the lowest cost or the maximum profit) in a given mathematical model in which the constraints are represented as linear equations.
2.3. Integer Programming

More formally, given a polytope (e.g. a polygon or a polyhedron), and a real-valued function is defined on the polytope such as in Eq. 2.1:

\[ f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \ldots + c_n x_n \]  

(2.1)

A linear programming method finds a point in the polytope where this function has the smallest (or largest) value. Such points may not exist. But if there are such points, searching through the polytope vertices guarantees to find at least one of them. Linear programs are problems that can be expressed in the following canonical form:

Maximize \( c^T x \)  
Subject to \( Ax \leq b \)  

(2.2a)  
(2.2b)

In Eq. 2.2, \( x \) represents the vector of variables (to be determined), while \( c \) and \( b \) are vectors of (known) coefficients, and \( A \) is a (known) matrix of coefficients. The expression to be minimized (or maximized) is called the objective function \( (c^T x) \). The equations \( Ax \leq b \) are the constraints that specify a convex polytope over which the objective function is to be optimized.

An INTEGER PROGRAMMING problem is any mathematical optimization program in which some or all variables are restricted to be integer. In many settings, the term ‘integer program’ is used as short-hand for INTEGER LINEAR PROGRAMMING (ILP). In contrast to linear programming, which can be solved efficiently in the worst case (i.e. in \( \mathcal{P} \)), INTEGER PROGRAMMING problems are in \( \mathcal{NP} \)-hard in many practical situations [136]. (For the definitions of computational complexities, please see Appendix F.2.1).

A BINARY INTEGER PROGRAMMING (BIP) problem is a special case of INTEGER PROGRAMMING where all variables are required to be 0 or 1 (rather than arbitrary integers). This problem is also classified as \( \mathcal{NP} \)-hard, and in fact, the decision version of this problem was one of Karp’s 21 \( \mathcal{NP} \)-complete problems (see Appendix F.2.1) [137].

If only some of the unknown variables are required to be integers, then the problem is called a MIXED INTEGER PROGRAMMING (MIP) problem. These are generally also \( \mathcal{NP} \)-hard problems. So it is very unlikely that there exists an algorithm which guarantees to find the optimal solution in a time that is polynomial. Finding the optimal solution requires an amount of time which is in the worst case grows exponentially with the problem size. Advanced algorithms for solving integer linear programs include: cutting-plane method, branch and bound, branch and cut, and branch and price [135].

Linear programming (including ILP, BIP, MIP) can be applied to various fields of study. Most extensively it is used in business and economic situations, but can also be utilized for many engineering problems. Some industries that use linear programming models include transportation, energy, telecommunications, and manufacturing. It has proven useful in modeling diverse types of problems in planning, routing, scheduling, assignment, and design [135, 136, 134].
2.4 Knapsack Problems

A special type of LP problems are KNAPSACK problems. Suppose a hitch-hiker has to fill up his/her knapsack by selecting from among various possible items which will give him/her maximum value (comfort). This knapsack problem can be mathematically formulated by numbering the items from 1 to \( n \) and introducing a vector of binary variables \( x_j (j = 1, 2, ..., n) \) having the following meaning [138, 139]:

\[
x_j = \begin{cases} 
1 & \text{if item } j \text{ is selected} \\
0 & \text{otherwise}
\end{cases}
\]

Additionally, let \( p_j \) the measure of the comfort and \( w_j \) the size of the item \( j \) and \( c \) the size of the knapsack. Thus, our goal is to select a vector from among all binary vectors \( x \) which satisfies the constraint

\[
\sum_{j=1}^{n} w_j x_j \leq c
\]

and maximizes the objective function

\[
\sum_{j=1}^{n} p_j x_j.
\]

This problem is considered as the representative of a variety of KNAPSACK problems (KP). In these problems it is desired to select one or more disjoint subsets so that the summed size of each subset does not exceed (or equals) a given bound and the summed value of the each subset is maximized.

2.4.1 Some areas of application

From practical point of view, these problems can model many industrial situations: capital budgeting, cargo loading, cutting stock – to mention the most classical applications. As we will see it later, in Chapter 3, it can be successfully applied in the field of software engineering also.

In the following sections, we will examine knapsack variants that are important to the Contribution 1’s (Chapter 4) point of view.

2.4.2 Binary Knapsack Problem

A special type of KP problems are BINARY KNAPSACK problems (BKP):

**Definition 2.4.1 (BINARY KNAPSACK problem (BKP) [138]).** There are \( n \) number of items, and each item has a value (called profit) and a size (called weight). I denote the value and size of item \( j \) as \( p_j \) and \( w_j \) respectively \((j = 1, 2, ..., n)\). This problem is known as the BINARY (0-1) KNAPSACK problem and can be formulated as follows:
Maximize $\sum_{i=1}^{n} p_j x_j$ \hspace{1cm} (2.6a)

subject to $\sum_{j=1}^{n} w_j x_j \leq c$ \hspace{1cm} (2.6b)

where $y_i = 0$ or 1, and $x_j = 0$ or 1 ($i, j \in \mathbb{N}$), and

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \hspace{1cm} (2.7a)$$

The BKP is the most important knapsack problem and one of the most intensively studied discrete optimization problems. The reason for such interest basically derives from three facts. First, it can be viewed as the simplest INTEGER LINEAR PROGRAMMING problem. Second, it appears as a subproblem in many more complex problems. Finally, it may represent a great many practical situations. During the last few decades, BKP has been studied through different approaches, according to the theoretical development of combinatorial optimization [138, 139].

### 2.4.3 Binary Multiple Knapsack Problem

An important generalization of the BINARY KNAPSACK problem is the BINARY MULTIPLE KNAPSACK problem (BMKP). The BMKP is arising when $m$ containers of given capacities $c_i : i = 1, ..., m$ are available. This knapsack problem can be mathematically formulated by introducing a binary variable $x_{i,j}$. This variable takes value 1 if item $j$ is selected for container $i$. Otherwise, the variable takes value 0 [138, 139]:

**Definition 2.4.2 (BINARY MULTIPLE KNAPSACK problem (BMKP) [138]).** There are $n$ number of items and a value (profit) and a size (weight) is associated with each item. I denote the value and size of item $j$ as $p_j$ and $w_j$ respectively ($j = 1, 2, ..., n$). There are also $m$ containers with given capacities $c_i : i = 1, ..., m$. This generalization of BKP is known as BINARY MULTIPLE KNAPSACK problem and can be formulated as follows:

Minimize $\sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{i,j}$ \hspace{1cm} (2.8a)

subject to $\sum_{j=1}^{n} w_j x_{i,j} \leq c_i$ \hspace{1cm} (2.8b)

$$\sum_{i=1}^{m} x_{i,j} = 1$$ \hspace{1cm} (2.8c)
2.4. Knapsack Problems

where \( y_i = 0 \) or 1, and \( x_{i,j} = 0 \) or 1 \((i,j \in N)\), and

\[
x_{i,j} = \begin{cases} 
1 & \text{if item } j \text{ is selected } i \\
0 & \text{otherwise} 
\end{cases} 
\tag{2.9a}
\]

\[
y_i = \begin{cases} 
1 & \text{if knapsack } i \text{ is used} \\
0 & \text{otherwise} 
\end{cases} 
\tag{2.9b}
\]

2.4.4 Binary Binpacking Problem

The well-known BINARY BINPACKING problem (BBP) is usually not included in the knapsack area. The BBP can be interpreted as a special case of MULTIPLE BINARY KNAPSACK problems (actually it is called as MULTIPLE SUBSET-SUM problem (MSSP) [138]). In this problem class, all containers have the same capacity \( c \), all items must be selected and it is desired to minimize the number of containers that are used. I can formulate the problem as follows [138, 139]:

Definition 2.4.3 (BINARY BINPACKING problem (BBP) [138]). There are \( n \) number of items and a size (weight) is associated with each item. I denote the size of item \( j \) as \( w_j \) \((j = 1, 2,...,n)\). There are also \( m \) containers with given same capacities \( c \). This generalization of BKP is known as BINARY MULTIPLE KNAPSACK problem and can be formulated as follows:

\[
\text{Maximize } \sum_{i=1}^{n} y_i 
\tag{2.10a}
\]

subject to

\[
\sum_{j=1}^{n} w_j x_{i,j} \leq c_i (1 - y_i) 
\tag{2.10b}
\]

\[
\sum_{i=1}^{n} x_{i,j} = 1 
\tag{2.10c}
\]

where \( y_i = 0 \) or 1, and \( x_{i,j} = 0 \) or 1 \((i,j \in N)\), and

\[
x_{i,j} = \begin{cases} 
1 & \text{if item } j \text{ is selected} \\
0 & \text{otherwise} 
\end{cases} 
\tag{2.11a}
\]

\[
y_i = \begin{cases} 
1 & \text{if bin } i \text{ is used} \\
0 & \text{otherwise} 
\end{cases} 
\tag{2.11b}
\]

2.4.5 Complexity of Knapsack Problems

In the last decades, an impressive amount of research on knapsack problems has been published in the literature. We will now show that all these problems are \( NP \)-hard [136]. The following results relate to the problems investigated in the Contribution 1 (Chapter 3).

For each problem \( P \), we either prove that the recognition version of the problem \( R(P) \) is \( NP \)-complete or that the generalization of the problem is already proved to be \( NP \)-hard. (Please note, the special class
of optimization problems are recognition version of problems. These problems require only a "yes" or "no" answer. The theory of computational complexity restricts attention to these problems for uniformity. However, the results can easily be generalized, since: i) there is a recognition version of the problem for every optimization problem, and ii) any complexity results for the recognition version also hold for the original problem. Please also note, the theoretical background for computational complexity relating to this research can be found in Appendix F.)

Let us consider the following problem [136]:

**Definition 2.4.4 (PARTITION problem).** Given \( n \) positive integers \( w_1, \ldots, w_n \). Is there a subset \( S \subseteq \mathbb{N} = 1, \ldots, n \) such that \( \sum_{j \in S} w_j = \sum_{j \in \mathbb{N} \setminus S} w_j \)?

This problem is \( \mathcal{NP} \)-complete problem, originally treated in Karp’s 21 \( \mathcal{NP} \)-complete problems [137].

**Theorem 2.4.5 ([138]).** \( \text{SUBSET-SUM} \) is \( \mathcal{NP} \)-hard.

**Proof.** Consider \( R(\text{SUBSET-SUM}) \), i.e.: given \( n + 2 \) positive integers \( w_1, \ldots, w_n, c \) and \( a \), is there a subset \( S \subseteq \mathbb{N} = 1, \ldots, n \) such that \( \sum_{j \in S} w_j \leq c \) and \( \sum_{j \in \mathbb{N} \setminus S} w_j \geq a \)? Any instance I of Partition can be polynomially transformed into an equivalent instance I’ of \( R(\text{SUBSET-SUM}) \) by setting \( c = a = \sum_{j \in \mathbb{N}} w_j / 2 \) (the answer for I is "yes" iff the answer I’ is "yes").

**Theorem 2.4.6 ([138]).** \( \text{BINARY KNAPSACK} \) is \( \mathcal{NP} \)-hard.

**Proof.** \( \text{SUBSET-SUM} \) is the particular case of \( \text{BINARY KNAPSACK} \) when \( p_j = w_j \) for all \( j \in \mathbb{N} \).

For the multiple problems (\( \text{BINARY MULTIPLE KNAPSACK}, \text{BINPACKING} \)) no pseudo-polynomial algorithm can exist, unless \( \mathcal{P} = \mathcal{NP} \). Since these problems can be proved to be \( \mathcal{NP} \)-hard in the strong sense. Consider the following recognition problem [138]:

**Definition 2.4.7 (3-PARTITION problem).** Given \( n = 3m \) positive integers \( w_1, \ldots, w_n \) satisfying \( \sum_{j=1}^{n} w_j / m = B \) integer and \( B/4 < w_j < B/2 \) for \( j = 1, \ldots, n \). Is there a partition of \( \mathbb{N} = 1, \ldots, n \) into \( m \) subsets \( S_1, \ldots, S_m \) such that \( \sum_{j \in S_i} w_j = B \) for \( i = 1, \ldots, m \)? (Notice that each \( S_i \) must contain exactly three elements from \( \mathbb{N} \).)

This problem is the firstly discovered to be \( \mathcal{NP} \)-complete in the strong sense [140].

**Theorem 2.4.8 ([138]).** \( \text{BINARY MULTIPLE KNAPSACK} \) is \( \mathcal{NP} \)-hard in the strong sense.

**Proof.** Consider \( R(\text{BINARY MULTIPLE KNAPSACK}) \), i.e.: given \( 2n + m + 1 \) positive integers: \( p_1, \ldots, p_n; w_1, \ldots, w_n; c_1, \ldots, c_m \), and \( a \). Are there \( m \) disjoint subsets \( S_1, \ldots, S_m \) of \( \mathbb{N} = 1, \ldots, n \) such that \( \sum_{j \in S_i} w_j \leq c_i \) for \( i = 1, \ldots, m \) and \( \sum_{i=1}^{m} \sum_{j \in S_i} p_j \geq a \)? Any instance I of 3-PARTITION can be pseudo-polynomially transformed into an equivalent instance I’ of \( R(\text{BINARY MULTIPLE KNAPSACK}) \) by setting \( c_i = B \) for \( i = 1, \ldots, m \), \( p_j = 1 \) for \( j = 1, \ldots, n \) and \( a = n \) (which implies that \( \bigcup_{i=1}^{m} S_i = \mathbb{N} \) in any "yes" instance).

**Theorem 2.4.9 ([138]).** \( \text{BINPACKING} \) is \( \mathcal{NP} \)-hard in the strong sense.
2.5 Scheduling Problems

Proof. Consider \( R(\text{BINARY KNAPSACK}) \), i.e.: given \( n + 2 \) positive integers: \( w_1, ..., w_n; c, \) and \( a \). Is there a partition of \( \mathbb{N} = \{1, ..., n\} \) into \( a \) subsets \( S_1, ..., S_a \) such that \( \sum_{j \in S_i} w_j \leq c \) for \( i = 1, ..., a \)?

Any instance \( I \) of 3-PARTITION can be pseudo-polynomially transformed into an equivalent instance \( I' \) of \( R(\text{BINPACKING}) \) by setting \( c = B \) and \( a = m \).

2.5 Scheduling Problems

Scheduling problems constitute an important part of the combinatorial optimization problems. Scheduling theory first appeared in the mid 1950s [141], since then their complexity heavily increased. Scheduling problems are encountered in variety types of systems, as it is often necessary to organize and/or distribute the work between many entities. The following definition is often cited [142]:

**Definition 2.5.1 (Scheduling [142]).** Scheduling concerns the allocation of limited resources to tasks over time. It is a decision-making process in which the goal is the optimization of one or more objectives.

In this definition, the task (or job or activity) is the entity to schedule. (Please note, tasks are usually broken down into a series of operations (smaller tasks). In this research I deal with only mono-operation tasks when all tasks contain only one operation.)

2.5.1 Some Areas of Application

Scheduling problems are encountered at all levels and in all sectors of our life. Basically, we can distinguish four main branches of problems [141, 143, 144, 145, 142, 146]:

1. **Production related problems.** These problems are broadly covered in the literature and most often in a well defined application class. They are encountered in assembly shops where certain equipment must be assembled. They include Flexible Manufacturing Systems (FMS), Robotic Cell Scheduling (RCS), Scheduling of Automated Guided Vehicles (AGV), and Car Sequencing Problems (CSP).

2. **Computer systems.** These problems are studied in different forms by considering mono- or multi-processor systems with the constraints of synchronization of operations and resource sharing.

3. **Timetable scheduling problems.** They concern all educational establishments or universities, since they involve timetabling of courses assuring the availability of teachers, students and classrooms.

4. **Project scheduling problems.** They comprise a vast literature. They include problems of scheduling operations that use several resources simultaneously (money, personnel, equipment, raw materials, etc.), and these resources being available in known amounts.

In this research (more specifically in the Contribution 2 – Chapter 4), I focus on how to allocate tasks to personnel or groups of people. Therefore, we are interested in the subproblem of the last problem class – namely the resource allocation in project scheduling.
2.5.2 Project Scheduling within the Life-cycle of a Project

Project scheduling problems have been extensively studied in the literature. They are usually separated from problems occurring in general scheduling environments, since they have their own particularities. In project scheduling jobs are called activities. In the following, the definition of a project is given [146, 143]:

Definition 2.5.2 (Project). Project is a major one-time undertaking that is dedicated to some well-defined objective and involving considerable money, personnel and equipment. It is initiated either by parent organization or by a customer.

The life cycle of a project can be constructed as five consecutive phases involving specific managerial tasks [146]:

1. Project conception usually involves some proposals, several preliminary studies (such as a feasibility study), economic and risk analysis. The aim of this phase is to decide whether or not the corresponding project will be performed.

2. Project definition defines the objectives of the project, selects the type of project organization, assigns resources to the project, and different tasks are identified with associated milestones.

3. Project planning decomposes each task into activities by means of a structural analysis (1) of the project. Then with time and resource estimations (2) it provides duration and resource requirements for each activity as well as temporal constraints between activities. The result of the structural analysis and the time and resource estimations can be represented with network modeling methods. Next, the temporal scheduling (3) of the project provides the earliest and latest start times of the activities – without regarding the resource availability. Finally allocating resources (4) over time to the execution of the activities are performed.

4. Project execution includes the implementation of the project. It is usually controlled by monitoring the progress against the schedule which has been established in the planning phase. In case of significant deviations from schedule, the resource allocation has to be performed again.

5. Project termination evaluates and documents the project after its completion to facilitate the management of future projects.

Figure 2.2 visualizes the previously listed phases and their relationships.

In this research, the focus is on the project planning phase, more specifically on the resource allocation subphase. This subphase is the most complex part of project planning consisting in allocating the scarce resources over time to the execution of the activities. Actually, RESOURCE ALLOCATION problems are made up of two subproblems: SEQUENCING problems and TIME-CONSTRAINED PROJECT SCHEDULING problems [146, 143].

Indeed, a very prominent and interesting class of combinatorial optimization problems is the class of SEQUENCING problems. A SEQUENCING problem is defined as follows [146]:

Definition 2.5.3 (SEQUENCING problem [146]). Given a finite set I = 1, 2, ..., n of items and a cost function c which assigns costs to per permutations of I, and the goal is to compute a minimum cost permutation of I.
2.5. Scheduling Problems

One of the most famous sequencing problems is the traveling salesman problem (TSP). The problem of computing good variable orders of binary decision diagrams (BDD) (which has to be solved for circuit verification and logic synthesis) and scheduling problems are representatives of this class [141, 143]. The pure SEQUENCING problem is a specialized scheduling problem, in which an ordering of the activities – can be expressed with temporal constraints – completely determines a schedule. The simplest SEQUENCING problem has a single resource. This SEQUENCING problem forms the core problem of project scheduling [143].

The other part of resource allocation is the time-constrained project scheduling that can be defined as follows:

**Definition 2.5.4 (Time-constrained project scheduling [146]).** Time-constrained project scheduling is concerned with computing the project schedule such that all temporal constraints – predetermined by the structural analysis (i.e. decomposition) or arisen from sequencing – are observed and some objective function reflecting the managerial goal of the project is optimized.

Please note, temporal constraints can be due dates, deadlines, precedences etc. In the resource allocation methods, sequencing and time-constrained project scheduling are usually performed jointly in an iterative manner.

### 2.5.3 Some Fundamental Notions

In the following we review some basic notions that are important my research’ point of view [146].

**2.5.3.1 Activities, and Temporal Relations**

A project can be considered to be a set of interacting tasks requiring time and resources for their completion. The structural analysis of the project provides a decomposition of the tasks into a set \( V \) of activities...
2.5. Scheduling Problems

and a set $E$ of precedence relationships among them i.e. ordering of the activities [146, 143]. This leads to the following

**Definition 2.5.5** (Activity, precedence relation, duration, and time lag [146]). Set $V$ consists of $n$ activities $i = 1, \ldots, n$ to be scheduled and two auxiliary activities 0 and $n + 1$, representing the project beginning and the project termination, respectively. The precedence relationships can be represented as activity pairs $(i, j)$ where $i \neq j$, saying that the start time of activity $i$ affects the earliest start time of activity $j$. Thus, $E \subseteq V \times V$ is some irreflexive relation in set $V$. The time estimation associates a duration (processing time) $p_i \in \mathbb{Z}_{\geq 0}$ with each activity and a time lag $\delta_{i,j} \in \mathbb{Z}$ with each pair $(i, j) \in E$.

**Remark 2.5.6.** An activity $i \in V$ is referred to as event (or fictitious activity) if $p_i = 0$. Otherwise, we speak of a real activity. The project beginning and termination, the receipt of materials, or milestones are examples of events. $V^a$ and $V^e$ respectively denote the sets of real activities and events of the project.

This immediately results the next

**Definition 2.5.7** (Start time of activity [146]). Let $S_i$ denote the start time of activity $i$, which has to be determined when scheduling the project in the temporal scheduling and resource allocation steps.

Since we assume that activities must not be interrupted when being in progress. The completion time of activity $i$ is defined as

**Definition 2.5.8** (Completion time of activity [146]). Let $C_i$ denote the completion time of activity $i$, which is calculated with the following formula

$$C_i \triangleq S_i + p_i.$$  \hspace{1cm} (2.12)

### 2.5.3.2 Temporal Constraints

Complexity of scheduling arises from the interaction between the activities by explicit dependencies and implicit dependencies, which may be the subject to some degree of uncertainty. Explicit dependencies are usually given by precedence relations between activities emanating from technological or organizational requirements typically. In this case, the tasks and their precedence constraints define the routing of this task.

The time lags between start time of activities give rise to the

**Definition 2.5.9** (Temporal constraints [146]). Temporal constraint is a relation denoted by $\delta_{i,j} \in P$, which means that if $(i, j) \in E$, activity $j$ cannot be started earlier than $\delta_{i,j}$ units of time after the start of activity $i$. This fact is expressed with

$$S_j - S_i + p_i \geq \delta_{i,j} \text{ for } (i, j) \in E.$$  \hspace{1cm} (2.13)

**Remark 2.5.10.** (a) Relation $P$ has irreflexive, transitive properties, therefore it is strict partial order (SPO). (b) The temporal constraints connect the start times of activities $i$ and $j$.

It results in the following

**Definition 2.5.11** (Minimum time lag, precedence constraint [146]). A nonnegative value of $\delta_{i,j}$ corresponds to the minimum time lag $d_{i,j}^{\text{min}} = \delta_{i,j} \geq 0$ between activities $i$ and $j$. If $d_{i,j}^{\text{min}} = p_i$, inequality 2.5.9 is referred to as a precedence constraint between activities $i$ and $j$.  

54
2.5. Scheduling Problems

Remark 2.5.12. Temporal constraint \((\delta_{i,j})\) denotes relational (ordering) information, time lag \((d_{i,j})\) contains numeric information, and finally precedence constraint is a special case of the minimum time lag.

The SEQUENCING problem (see Def. 2.5.3) consists in assigning time intervals to the execution of the activities while taking into account the prescribed temporal constraints (precedence constraint and activity duration).

Some other constraints occur frequently in practice that can be modeled by minimum and maximum time lags between activities:

- **Release date** \(r_i\) for the start of activity \(i\) (head of \(i\)): \(d_{0i}^{\min} = r_i\).
- **Deadline** \(\bar{d}_i\) for the completion of activity \(i \in V\): \(d_{0i}^{\max} = \bar{d}_i - p_i\).
- **Quarantine time** \(q_i\) after the completion of activity \(i\) (tail of \(i\)): \(d_{i,n+1}^{\min} = p_i + q_i\).
- **Fixed start time** \(t_i\) for activity \(i\): \(d_{0i}^{\min} = d_{0i}^{\max} = t_i\).

Remark 2.5.13. (a) The project is started at time 0 and must be completed by a prescribed deadline \(\bar{d}\), i.e., \(S_0 = 0\) and \(S_{n+1} \leq \bar{d}\). The deadline is represented as a maximum time lag \(d_{0,n+1}^{\max} = \bar{d}\) between the project beginning 0 and the project termination \(n + 1\). (b) **Deadline**, **Due date** and **Availability date** are unary, and **Precedence relations** (Def. 2.5.11) are binary temporal constraints.

If temporal constraints are given, we speak **TIME-CONSTRAINED PROJECT SCHEDULING problem** (TCPSP) – see its definition in Sec. 2.5.3.3.

### 2.5.3.3 Time-constrained Project Scheduling Problem

A project is uniquely given by specifying the respective start times \(S_i\), that is why we will always represent solutions to project scheduling problems by a vector of activity start times. It leads to the following

**Definition 2.5.14** (Schedule [146]). The vector \(S = (S_0, S_1, ..., S_{n+1})\) of start times for the activities, where \(S_i \geq 0 : i \in V\), and \(S_0 = 0\) – is called a schedule. In this sequence the starting time of \(S_i\) is equal to the value of the maximal path from \(S_0\) to \(S_i\) in graph \(G(V, E)\), and \(S_{n+1}\) is equal to the value of the critical path (in other words it is called as makespan).

In schedule, in general, parameters are assumed to be integer-valued. A special, important subset of schedules is

**Definition 2.5.15** (Time-feasible schedule [146]). Schedule \(S\) is said to be time-feasible if it satisfies the temporal constraints 2.5.9. The set of all time-feasible schedules is denoted by \(S_T\).

Obviously, set \(S_T\) represents an integer polytope in \(\mathbb{R}_{\geq 0}^{n+2}\). Assume that \(S_T \neq \emptyset\). It is well-known that the partially ordered set \((S_T, \leq)\) possesses exactly one minimum \(ES\), where \(S \leq S'\), more precisely if \(S_i \leq S'_i\) for all \(i \in V\). I refer to \(ES\) as the earliest schedule. Furthermore, by Remark 2.5.13 \((S_T, \leq)\) possesses exactly one maximum \(LS\), which is termed the latest schedule. This means that there is no time-feasible schedule \(S\) such that \(S_i < ES_i\) or \(S_i > LS_i\) for any \(i \in V\). The interval \([ES_i, LS_i]\) is termed the time window (for the start) of activity \(i\) [146].
2.5. Scheduling Problems

**Definition 2.5.16 (TIME-CONSTRAINED PROJECT SCHEDULING problem (TCPSP) [146]).** Let \( f : S_T \rightarrow \mathbb{R} \) be an objective function assigning a value \( f(S) \) to each time-feasible schedule \( S \). Without loss of generality I assume that the objective function has to be minimized. The basic TIME-CONSTRAINED PROJECT SCHEDULING problem can then be stated as follows [146]:

\[
\begin{align*}
\text{Minimize } & \quad f(S) \\
\text{subject to } & \quad S \in S_T
\end{align*}
\]

**Definition 2.5.17 (Time-optimal schedule [146]).** A time-feasible schedule \( S \) solving the TIME-CONSTRAINED PROJECT SCHEDULING problem 2.14 is called time-optimal.

The TCPSP problem can be solved in \( \mathcal{O}(mn) \) time by the Bellman’s label-correcting procedure shown in [147]. Although in [148] it is showed that it cannot be decided in polynomial time (i.e. it can be decided in \( \mathcal{NP} \) time) whether or not there exists a feasible schedule.

### 2.5.3.4 Resource-constrained Project Scheduling Problem

To perform the activities of a project, different types of resources are required. Basically, we may distinguish between renewable (like manpower or machinery) and non-renewable (like project budget, materials) resources. I aim at renewable resources only, to which the RESOURCE-CONSTRAINED PROJECT SCHEDULING problems (RCPS) are related [149, 145]. The resource assignments express explicit, and the scarcity of resources forms implicit dependencies (see Sec.2.5.3.2).

**Definition 2.5.18 (Project resource [146]).** Let \( \mathcal{R} \) be the set of (renewable) resources \( k \). Resources have capacities: \( R_k \in \mathbb{N} \cup \{\infty\} \) – set have been assigned to the project during definition phase. The \( R_k = \infty \) means that the availability of resource \( k \) is not explicitly bounded from above, but can be adapted at a certain cost, to any usage over time.

Now let \( S \) be some schedule and let \( t \) be some point in time. Then let \( \mathcal{A}(S, t) \triangleq \{ i \in V \mid S_i \leq t \leq S_i + p_i \} \) be the active set of activities being in progress at time \( t \). The corresponding requirement for resource \( k \in \mathcal{R} \) at time \( t \) is given by \( r_k(S, t) \triangleq \sum_{i \in \mathcal{A}(S, t)} r_{i,k} \). For given schedule \( S \), function \( r_k(S, \cdot) : \mathcal{R} \rightarrow \mathbb{Z}_{\geq 0} \) is termed the loading profile for renewable resource \( k \). Obviously, we have \( r_k(S, t) = 0 \) for all \( t < 0 \).

The renewable-resource constraints can be treated as follows:

\[
\begin{align*}
r_k(S, t) & \leq R_k : k \in \mathcal{R}, \ 0 \leq t \leq d
\end{align*}
\]

The scarcity of resources is used to establish an implicit dependencies between activities and they formulated by resource constraints. The resource allocation problem consists in assigning time intervals to the execution of the activities while taking into account the prescribed temporal and resource constraints [146].

**Definition 2.5.19 (Resource-feasible schedule [146]).** A schedule \( S \) satisfying the renewable-resource constraints 2.15 is called resource-feasible with respect to renewable resources \( k \in \mathcal{R} \).

**Definition 2.5.20 (Feasible schedule [146]).** The set of all resource-feasible schedules is denoted by \( S_R \).

\( S \triangleq S_T \cap S_R \) is the set of all feasible schedules.
2.5. Scheduling Problems

As we will see later on, unlike the polytope of time-feasible schedules \( S_T \), set \( S_R \) represents a finite union of polytopes which is generally not connected. Resource allocation consists in assigning start times \( S_i \) (and thus execution time intervals \([S_i, C_i]\)) to the activities of the project such that the corresponding schedule \( S = (S_i)_{i \in V} \) is feasible and minimizes the objective function on set \( S \) [146].

**Definition 2.5.21** (RESOURCE-CONSTRAINED PROJECT SCHEDULING (RCPSP) problem [146]). The basic RESOURCE-CONSTRAINED PROJECT SCHEDULING problem with renewable resources reads as follows:

\[
\begin{align*}
\text{Minimize} & \quad f(S) \\
\text{subject to} & \quad S \in S_T \cap S_R
\end{align*}
\]

(2.16)

Recall that we have assumed objective function \( f \) to be lower semi-continuous. The compactness \( S \) then implies that there exists an optimal solution to problem 2.16 precisely if \( S \neq \emptyset \). Note, however, that due to the presence of maximum time lags it may happen that \( S_T \neq \emptyset \) and \( S_R \neq \emptyset \) but \( S = \emptyset \).

**Definition 2.5.22** (Optimal schedule [146]). A feasible schedule \( S \) solving the RESOURCE-CONSTRAINED PROJECT SCHEDULING problem 2.16 is called optimal.

By replacing the set \( S = S_T \cap S_R \) of feasible schedules with the set of resource-feasible schedules \( S_R \) we obtain the temporal relaxation of the RESOURCE-CONSTRAINED PROJECT SCHEDULING problem 2.16. Since we have assumed that \( r_{ik} \leq R_k \) for all \( i \in V^a \) and all \( k \in R^a \), each schedule carrying out the activities one after another is resource-feasible. The resource relaxation of problem 2.16 arises from deleting the resource constraints 2.15. The resource relaxation coincides with the basic TIME-CONSTRAINED PROJECT SCHEDULING problem 2.14.

When dealing with the project duration problem, we may drop the assumption that there is a deadline \( \bar{d} \) on the project termination because the objective is to maximize the slack \( \bar{d} - S_{n+1} \) of the deadline constraint. The construction of a feasible schedule then turns into an easy problem if there are no maximum time lags given. In that case, project network \( N \) is acyclic, and the activities can be scheduled consecutively according to any topological ordering of the nodes \( i \in V \) [146].

### 2.5.3.5 Optimality Criteria

In order to evaluate schedules, we can use a certain number of criteria. Occasionally, we want a criterion to be close to a certain reference value. There is a frontier between the notions of criteria and constraints. If a constraint represents a fact which definitely must be respected, optimizing a criterion allows rather a certain degree of freedom. From a practical point of view, the difference between a criterion and a constraint is only apparent to the decision maker who initiates a schedule calculated by an algorithm. Some scheduling problems have no criterion to be optimized. In this case, we are dealing with a feasibility problem, also called a decision problem: *Does a solution which satisfies the constraints exist?* [149, 145]

We can classify criteria into two large families: *minimax criteria*, which represent the maximum value of a set of functions to be minimized, and *minisum criteria*, which represent a sum of functions to be minimized. From now on we are only dealing with completion time minimax criteria which is most used in project scheduling environments. The minimax criteria give rise to the
2.5. Scheduling Problems

**Definition 2.5.23 (Completion time of the project [146])**. Let us define $C_{\text{max}} = \max_{i=1, ..., n}(C_i)$, with $C_i$ being the completion time of activity $i$. This criterion is also called *makespan*, which is the total length or duration of a schedule, i.e. it is the completion time of the last scheduled activity.

Definition 2.5.24 introduces the notion of regular criterion in the case of a minimization problem.

**Definition 2.5.24 (Regular criterion [146])**. Let $S$ be the set of solutions. A criterion $Z$ is a regular criterion if and only if $Z$ is an increasing function of the completion times of jobs, i.e. if and only if:

$$\forall x, y \in S, C_i(x) \leq C_i(y), \forall i = 1, 2, ..., n \Rightarrow Z(C_1(x), ..., C_n(x)) \leq Z(C_1(y), ..., C_n(y))$$ (2.17)

For the criteria presented in Def. 2.5.23, I deduce the following result:

**Corollary 2.5.25.** The criteria $C_{\text{max}}$ is regular.

2.5.3.6 Scheduling Rules

Several scheduling rules, optimal or heuristic, have been proposed in the literature. They are very often used in heuristic applications to industrial problems, given their simplicity and the little calculation time which they require. The most popular rules are the following [149, 145]:

**Definition 2.5.26 (Scheduling rules, SPT, WSPT, EDD [146])**. Let us define the following scheduling rules:

- *Shortest processing time first* (SPT): sequences the jobs in increasing order of their processing time.
- *Weighted shortest processing time first* (WSPT): sequences the jobs in increasing order of their ratio $p_i/w_i$.
- *Earliest due date first* (EDD): sequences the jobs in increasing order of their due date $d_i$.

The rule SPT, WSPT, EDD which enable us to compute an optimal active schedule for the singular resource with average completion time ($\bar{C}$), average weighted completion time ($\bar{C}^w$), maximum lateness of activities ($L_{\text{max}}$).

2.5.3.7 Visualization of Project Networks

$G(V, E)$ denotes the graph of precedence constraints (transitively reduced), while single precedence constraints are denoted alternatively by $i \rightarrow j$ or $i \leftarrow j$. $\text{Pred}(j)$ denotes the set of direct predecessors while $\text{Succ}(j)$ is the set of direct successors of activity $j$. As previously stated, the processing time of activity $j$ is given by $p_j$.

I will show how the activities $i \in V$ and the temporal constraints $S_j - S_i \geq \delta_{i,j}$ for $(i,j) \in E$ can be represented by a *project network*. Basically, there are three different types of project networks [145, 146]. *Activity-on-Arc (AoA) network* or *critical-path method (CPM) network* associates an arc $(u, v)$ with each activity $i$, where the nodes $u$ and $v$ represent events and arc $(u, v)$ is weighted by the duration $p_i$ of the corresponding activity $i$. 
2.5. Scheduling Problems

In event-on-node (EoN) network each real activity \( i \) is represented by two events \( i^s \) and \( i^c \) in node set \( V \). \( i^s \) corresponds to the start and \( i^c \) to the completion of activity \( i \). Both nodes are linked by two arcs \((i^s, i^c)\) and \((i^c, i^s)\) with weights \( \delta_{i^s,i^c} = p_i \) and \( \delta_{i^c,i^s} = -p_i \).

In this research, I represent projects by Activity-on-Node (AoN) network that is defined as follows. In Activity-on-Node networks, the nodes are identified with the activities. For each time lag \( \delta_{i,j} \), the network contains one arc \((i,j)\) with initial node \( i \) and terminal node \( j \), i.e., \( V \) is the node set and \( E \) is the arc set of the network. An arc \((i,j) \in E \) is weighted by \( \delta_{i,j} \) (time lag). Activity-on-Node networks belong to the class of metra-potential method (MPM) networks. Obviously, Activity-on-Node networks can cope with general temporal constraints. In addition, due to the one-to-one correspondence between precedence relationships and arcs, there is a unique Activity-on-Node representation of the project.

Example 2.5.1 (AoN Network). We consider a project with four real activities \( i = 1, 2, 3, 4 \) for which we assume that activities 3 and 4 cannot be started before activities 1 and 2 have been completed. The project must be completed by a prescribed deadline \( \bar{d} \). Figure 2.3 shows the corresponding Activity-on-Node project network.

![Figure 2.3: Example AoN Project Network.](image)

2.5.4 PS\[prec\]|C\(_{\text{max}}\) Project Scheduling Problem

The PS\([prec\)|C\(_{\text{max}}\) \([143]\) model forms the core problem among the class of RESOURCE-CONSTRAINED PROJECT SCHEDULING problems. Basically, while minimizing the project’s (PS) makespan (\( C_{\text{max}} \)), we have to observe precedence (prec) and resource constraints. Recently, a couple of papers have contributed new solution procedures, however the problem is still rather challenging from a computational point of view. In this case 1) the project network is assumed to be acyclic and depicted by an Activity-on-Node (AoN) network (see Sec. 2.5.3.7) with nodes as activities and arcs as precedence relations; 2) preemption is not allowed; 3) there are scarce renewable resources; and 4) all data are assumed to be integer-valued. The objective is to find a makespan-minimal schedule that meets the constraints imposed by the precedence relations and by limited resource availabilities \([149, 145]\).

2.5.5 Complexity of PS\[prec\]|C\(_{\text{max}}\) Problem

The RCPSP problem consists in assigning time intervals to the execution of the activities while taking into account the prescribed temporal constraints and resource scarcity. Next, I present the following theorem which relates to the problems investigated in the Contribution 2.
Theorem 2.5.27 ([141]). The $PS|prec|C_{\text{max}}$ RCPSP problem is $NP$-hard.

The proof of the theorem 2.5.27 is rather complex, please see the details in [146].

2.6 Graph Partitioning and Clustering

This section provides an overview of graph partitioning and clustering which theoretical backgrounds are applied in Contribution 3. Graphs are structures formed by a set of vertices (or nodes) and a set of edges (or links) that are connections between pairs of vertices. Graph partitioning is the task of identification in terms of dividing the connected elements into pieces. Partitioning algorithms are typically used as divisive algorithms: partitioning is the task of dividing the vertices of the graph into groups considering the edge structure of the graph in such a way that, there should be many edges within each group and relatively few between the groups [150, 151].

Graph clustering is a similar process to partitioning. In this case, the task is to separate the vertices of the graph taking into consideration the edge structure of the graph. Clustering is carried out by such a way that there should be many edges within each group and relatively few between the groups. The grouping procedure is based on some kind of similarity measure on the elements [152]. Clustering is closely related to unsupervised learning in pattern recognition systems, where the basic task is to classify a data set into groups based on a similarity measure over the data, without any a priori information on how the classification should be done.

2.6.1 Some areas of application

Partitioning and clustering have many applications. For example, in biology, clustering is used to build groups of genes with related expression patterns. In social networks, it may be used to recognize communities within large groups of people. In image segmentation, it also can be used to divide a digital image into distinct regions for object detection (e.g., face recognition). Cluster analysis is widely used in market research when working with multivariate data from surveys and test panels. Application of the graph partitioning arise from many diverse areas such as scientific computing, engineering, operations research, optimization, data mining, geographical information systems, VLSI design, parallel processing, task scheduling and load balancing for parallel computing. [153, 151]

2.6.2 Some Fundamental Notions

In the following I review some basic notions that are important my research point of view [150].

Definition 2.6.1 (Graph, number of vertices, number of edges, degree of a vertex $v$). A graph is a pair $G = (V, E)$ of sets such that $E = |V|^2$. Thus the elements of $E$ are 2-element subsets of $V$. The elements of $V$ are called vertices (or nodes, or points) and elements of $E$ are called edges (or links or lines). The vertex set of a graph $G$ is referred to as $V(G)$ and its edge sets as $E(G)$. The number of vertices of a graph $G$ is its order and written as $|G|$; its number of edges is denoted by $|G|$. A degree $\deg(v)$ of a vertex $v$ is the number $|E(v)|$ of edges at $v$, in other words, it is equal to the number of neighbors of $v$. 
2.6. Graph Partitioning and Clustering

Definition 2.6.2 (Path). A path in a non-empty graph \( P = (V, E) \) of the form \( V = \{x_0, x_1, \ldots, x_k\} \); \( E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\} \), where \( x_i \) are distinct. The vertices \( x_0 \) and \( x_k \) are linked by \( P \).

Definition 2.6.3 (Connected graph, and component). A non-empty graph \( G \) is called connected if any two of its vertices are linked by a path in \( G \). Let \( G = (V, E) \) be a graph. A maximal connected subgraph of \( G \) is called a component of \( G \).

Remark 2.6.4. 1) A component being connected is always non-empty. 2) An alternative way to define connected components involves the equivalence classes of an equivalence relation that is defined on the vertices of the graph. In an undirected graph, a vertex \( v \) is reachable from a vertex \( u \) if there is a path from \( u \) to \( v \). In this case, reachability is an equivalence relation, since it is reflexive, symmetric and transitive.

Definition 2.6.5 (Adjacent of a vertex, neighbor of a vertex, and neighborhood of a vertex \( v \)). An adjacent vertex of a vertex \( v \) in a graph is a vertex that is connected to \( v \) by an edge. The neighborhood of a vertex \( v \) in a graph \( G \) is the induced subgraph of \( G \) consisting of all vertices adjacent to \( v \) and all edges connecting two such vertices. The neighborhood is often denoted \( N_G(v) \) or (when the graph is unambiguous) \( N(v) \).

Remark 2.6.6. The neighborhood described above does not include \( v \) itself, and is more specifically the open neighborhood \( \text{open neighborhood} \) of \( v \). It is also possible to define a neighborhood in which \( v \) itself is included, called the closed neighborhood \( \text{closed neighborhood} \) and denoted by \( N_G[v] \).

Definition 2.6.7 (Distance between \( v \) and \( w \) and diameter). A distance \( d(u, w) \) in \( G \) of two vertices of \( v \) and \( w \) is the length of a shortest \( v \) \( - \) \( w \) path in \( G \). If no such a path exist, we set \( d(u, w) = \infty \). The greatest distance between any vertices in \( G \) is the diameter of \( G \) – written as \( \text{diam}G \).

2.6.3 Identifying Clusters

There are two main approaches for identifying clusters. Computing some values for the vertices and then classify the vertices into clusters based on the values obtained (vertex similarity measures). The other method is to compute a fitness measures over the set of possible clusters and then choose among the set of cluster candidates those that optimize the measure used [153, 151]. In this research, only the vertex similarity measure-based approach is applied, therefore I only present this kind of method.

2.6.3.1 Vertex Similarity Measures

There are many clustering algorithms based on similarities between the vertices [152]. The higher the similarity, the stronger the need to cluster the vertices together. Computing such similarities is not necessarily simple, and in some cases evaluating the similarity of two vertices may turn out to be a task even more complex than the clustering of the graph once the similarities are known.

In this case I use a distance measure to determine the cluster boundary. With distance measure it is desirable to cluster together vertices that have small distances to each other. Defining or selecting an appropriate similarity or distance function depends on the task at hand. The number of similarity measures used in the literature has been very high for various decades [154].

Given a data set, a distance measure \( \text{dist}(d_i, d_j) \), should fulfil the following criteria:
2.6. Graph Partitioning and Clustering

The distance from a datum to itself is zero: \( \text{dist}(d_i, d_j) = 0 \).

The distances are symmetrical: \( \text{dist}(d_i, d_j) = \text{dist}(d_j, d_i) \).

The triangle inequality holds: \( \text{dist}(d_i, d_j) = \text{dist}(d_i, d_k) + \text{dist}(d_k, d_j) \).

For points in an \( n \) dimensional Euclidean space, possible distance measures for two data points \( d_i = (d_{i,1}, d_{i,2}, ..., d_{i,n}) \) and \( d_j = (d_{j,1}, d_{j,2}, ..., d_{j,n}) \) include:

- \( \text{dist}(d_i, d_j) = \sum_{k=1}^{n} \sqrt{(d_{i,k} - d_{j,k})^2} \) (Euclidean distance or \( L_2 \) norm)
- \( \text{dist}(d_i, d_j) = \sum_{k=1}^{n} |(d_{i,k} - d_{j,k})| \) (Manhattan distance or \( L_1 \) norm)
- \( \text{dist}(d_i, d_j) = \max_{k \in \{1, n\}} |(d_{i,k} - d_{j,k})| \) (\( L_{\infty} \) norm)

Once the vectors are computed, a similarity measure can be applied.

2.6.3.2 Methods for Graph Clustering

Basically, there are two kinds of methods for clustering: global and local. In a global clustering, each vertex of the input graph is assigned a cluster in the output of the method. Whereas in a local clustering, the cluster assignments are only done for a certain subset of vertices, usually only one vertex. For large graphs, global clustering becomes computationally demanding. For massive data sets, the running time of a clustering algorithm should not grow faster than \( O(n) \) in order to be scalable. A brief survey on some global clustering methods is given in [155].

The minimum k-clustering problem is the combinatorial optimization problem where a finite data set \( D \) is given together with a distance function \( d : D \times D \to \mathbb{N} \), where \( d \) satisfies the triangle inequality \( (2.18) \). The task is to group \( D \) into \( k \) clusters \( C_1, C_2, ..., C_k \), where \( C_i \cap C_j = \emptyset \) for \( i \neq j \), such that the maximum inter-cluster distance is minimized (i.e. the maximum distance between two points assigned to the same cluster). This problem is approximable within a factor of two, but not approximable within \( (2 - \epsilon) \) for any \( \epsilon > 0 \) [156]. A popular algorithm for clustering vector data with respect to a distance function is the k-means method [151]. The basic idea of the k-means method is to cluster a set of points in some metric space into \( k \) clusters by iteratively improving \( k \) cluster centers and grouping each point to the cluster with the closest center; the centers are chosen to minimize the sum of squares of the intra-cluster distances. Unfortunately, k-means is \( \mathcal{NP} \)-hard even for \( k = 2 \) [151].

Other variants of global clustering are (that can be mixed together) 1) iterative (assigning one element at a time to an appropriate cluster), 2) online (process a group of elements at a time that are available), 3) incremental (incorporating newly added data into the existing clustering), and 4) hierarchical clustering (clusters are defined as a hierarchical structure).

Divisive clustering algorithms are a class of hierarchical methods that work top-to-down, recursively grouping the graph into clusters. Naturally, in addition to dividing the graph top-to-down into clusters, one may also work bottom-to-up merging singleton sets of vertices iteratively into clusters. Such methods are called agglomerative clustering algorithms [152].
2.6.4 Identifying Partitions

In general, computing the optimal partitioning is an \( \mathcal{NP} \)-complete problem, which means that the best known algorithms take time which is an exponential function \([140]\). Therefore we need to use heuristics to get approximate solutions for problems where \(|V|\) is large. There are two classes of heuristics, depending on the information available about the graph \( G \). In the first case, we have coordinate information with each node. This is frequently available if the graph comes from triangulating or otherwise discretizing a physical domain. Typically, such graphs have the additional property that only nodes which are spatially close together have edges connecting them. We may use this property to develop two efficient geometrical algorithms for graph partitioning.

The second kind of graph we want to partition is coordinate free, in other words, there is no identification of a node with a physical point in space. Additionally, edges have no simple interpretation such as representing physical proximity. These kinds of graphs require combinatorial rather than geometric algorithms to partition them. (Please note, obviously combinatorial algorithms may also be applied to graphs with coordinate information.)

2.6.4.1 Methods for Graph Partitioning

The earliest algorithms used a graph algorithm called breadth first search. It works well for some finite element meshes. The Kernighan-Lin (the same who designed the language C) algorithm takes an initial partitioning and iteratively improves it by trying to swap groups of nodes between \( x_1 \) and \( x_2 \), greedily picking the group to swap that best minimizes the number of edge crossings at each step. In practice, it converges quickly to a (local) optimum if it has a good starting partition. As of this writing, Kernighan-Lin combined with the multilevel technique is the fastest available technique, although the geometric methods are also fast and effective on graphs with appropriate properties. \([157]\) (Please note, the idea in multilevel technique is to replace the graph by a coarser graph with many fewer nodes partition the coarser (and so easier) graph. Then the coarse partition is used as a starting guess for a partition of the original graph. The coarse graph may in turn be partitioned by using the same algorithm recursively: a yet coarser graph is partitioned to get a starting guess, and so on.)

2.6.4.2 Minimum K-Cut Problem

The following problem is important concerning the Contribution 3. The MINIMUM K-CUT, is a graph partitioning problem that requires finding a set of edges whose removal would partition the graph to \( k \) connected components. These edges are referred to as \( k \)-cut. More formally, I can formulate the MINIMUM K-CUT problem as follows.

**Definition 2.6.8 (MINIMUM K-CUT).** Let \( G = (V, E) \) be an undirected graph (where node set is \( V = v_1, ..., v_n \) and edge set is \( E \)) with nonnegative edge weights \( l(e), e = (v_i, v_j) \in E \), and let \( K \) be a set of \( p \) positive integers \( k_i \) such that \( \sum_{i=1}^{p} k_i \leq |V| \). A K-cut is a collection of disjoint node sets \( P_i \) such that \( \forall i \in 1, ..., p |P_i| = k_i \). The MINIMUM K-CUT PROBLEM is to find a k-cut such that \( \sum_{i=1}^{p-1} \sum_{j=i+1}^{p} l(P_i, P_j) \) (where \( l(P_i, P_j) = \sum_{u \in P_i} \sum_{v \in P_j} l(u,v) \)) is minimized. \([77]\)
2.6. Graph Partitioning and Clustering

Actually, finding the optimal solution for the edge cutting minimization problem of graph partitioning for non-trivial cases is a complex task. Next, I present the following theorem which relates to the problems investigated in the *Contribution 3*.

**Theorem 2.6.9 ([158])**, *The Minimum K-Cut problem is NP-hard.*

The proof of the theorem 2.6.9 is rather complex, please see the details in [158]. A variety of algorithms were developed to solve this graph partitioning problem class [159].
Chapter 3

Optimized Agile Release Scheduling

RELEASE SCHEDULING aims at the selection and assignment of features to a sequence of consecutive product releases while several constraints – such as technical, resource, temporal and budget – are fulfilled. Although agile software development represents a major approach to software engineering there is no well-established conceptual definition and sound methodological support of agile release scheduling. In this chapter I introduce my first contribution (C1) that is a novel method (namely Agile Release Scheduling (ARS)) for agile release planning to provide improved efficiency and effectiveness in release-centered decisions.

3.1 Synopsis

The growing pressure to reduce costs, time-to-market and to improve quality catalyzes transitions to more automated methods and tools in software engineering to support project planning, scheduling and decisions [127]. In agile environments, scheduling features into the upcoming delivery stages is a complex, informal and mainly manual process. In 2008, an Agile Tools survey [160] showed that many developers-focused tools appeared (including JUnit testing, sub versioning, auto build, etc.) in the last decade, but most companies (> 52%) are still using old-fashioned project management tools like [74] or generic tools like spreadsheets. Surprisingly, 18% of the respondents do not use any tool for project scheduling and tracking at all – although many commercial (such as [161]) and open source (e.g. [162]) agile release planning tools are available.

The following chapter draws up my novel concepts, models and algorithms for agile release planning to supplement the deficiency of traditional planning. In the chapter, I present 1) a conceptual model (C1.1), and 2) a novel multiple knapsack-based optimization model (C1.2) with 3) a branch & bound global optimization algorithm (C1.3) for agile release scheduling as a solution. The elaborated method shows more smooth and fully padded iterations (> 80% of the total iterations are optimal), and prevents resource overload comparing to the traditional approaches.

Up to my best knowledge, the elaborated staged-delivery global optimized model formulation is novel in the field. Although, there are some tenets to manual scheduling [34, 67] algorithmic solution could not be found.
3.1. Synopsis

3.1.1 Related Work

Projects usually have more candidate requirements than that can be realized within the time and resource constraints of the release. Prioritization helps to select the more important requirements from the set of candidates to support release-centered decisions. Basically, there are two different techniques to prioritize requirements. The absolute prioritization technique (e.g. MoSCoW rules [67]) determines the importance of requirements without comparing to any other requirement. While the relative prioritization technique is based on the expressed relative importance among requirements. Relative techniques tend to be more accurate and informative than absolute ones [106]. One relative technique is the $100-test presented in [84]. It assigns a fixed amount of units among all requirements by different users and/or customers in order to construct relative weighted list of requirements. An other popular method is the Wiegers’ method. It calculates the priority of a requirement by dividing the value of a requirement by the sum of the costs and technical risks associated with its implementation [107].

The Karlsson's pair-wise comparison technique [106] is based on the Analytical Hierarchy Process (AHP) [108, 109]. In this technique, all requirement pairs are compared according to their importance on the same scale that is employed in AHP: 1 (equal); 3 (moderate difference); 5 (essential difference); 7 (major difference), and 9 (extreme difference). Then these comparisons lead to the understanding of each requirement value regarding to the total value of the requirements. In [110], an improved technique of the Karlsson’s pair-wise comparison technique is introduced. It uses cost and value as high-level factors in the comparisons also. Finally, in [112], values to each requirement is assigned then it formulates selection of requirements as binary knapsack problem (see Sec. 2.4.2).

Compared to the extensive research on requirements prioritization only few researches dealt with requirements release planning. In [90], release planning as an integer linear programming problem (ILP) is formulated. In [113], release planning is formulated as a binary knapsack problem where requirement dependencies are treated as constraints. The ILP planning approach with users’ and/or customers’ opinions is extended in [75]. This approach was also extended with some managerial steering mechanism that enabled what-if analysis [114]. The scheduling of requirements to provide both requirement selection and on-time-delivery project plan is introduced in [115]. In [E9] a case study showed that integration of requirements and planning how significantly (> 50%) can accelerate UML-based release scheduling. In [E13] a bin-packing-based suboptimal release scheduling solution is presented. In [C5] the iteration scheduling is formulated as a special form of resource-constrained project scheduling problem (RCPSP) to provide semi-automatic schedule generation and what-if-analysis. There are other techniques that deal with the case when several users and/or customers are involved, such as EVOLVE [116] and Quantitative WinWin [117]. In [118], utilized Bayes belief network is presented to take uncertainty of effort for resource decisions into consideration. In [119] a method is proposed to focus on schedule level minimization but did not examine different productivity level of developers. In [120] a two-phased optimization approach is proposed that combines the planning and resources allocation tasks of one release. This method assumes that each feature is decomposed into a sequence of tasks (e.g. design, implementation and testing) in release planning time. The second phase, which can be used alone in small problems, performs an unfocused search to allocate tasks to developers by formulating the problem as a special case of job shop scheduling problem (JSSP) and this problem is solved with a genetic algorithm. The optional first phase is used to reduce the search space by formulating the problem as a binary knapsack decision problem. The advantage of the first phase becomes considerable when the feature count is relatively large (> n * 10).
3.1. Synopsis

Due to the characteristics of agile approaches, agile release planning is significantly different from the previously proposed methods. As a consequence of the characteristics of agile approaches, agile planning is usually executed in two steps to support the required flexibility [67]: 1) release planning – a coarse-grained level plan that considers features (essential abstractions that both customers and developers understand), team resource (human resources in a cumulative way) and several delivery stages (i.e. iterations) (see Sec. 1.4.2), and 2) iteration planning – a fine-grained level plan that considers technical tasks (working units derived from features), human resources and one delivery stage (i.e. the iteration) (see Sec. 1.4.3). So, an agile release is made up of some iterations, and agile planning is carried out in an iterative way by defining: 1) an early coarse-grained level release plan where the major characteristics of the development are known (e.g. deliverable features, the size of the team); and 2) several fine-grained level iteration plans where the detailed expectations and development situation are known (e.g. technical tasks, task dependencies, available developers). After execution of the release the whole process is usually repeated.

3.1.2 Problem Statement and Analysis

The lack of penetration of modern agile planning tools (see Sec. 3.1) can be explained by the weak embedded support of traditionally important project scheduling functions such as task assignments and what-if analysis. Their implemented methods provide ‘quick and dirty’ scheduling solutions [161, 162]: the team can distribute deliverables among releases and iterations in planning meetings – while all planning constrains are taken into account informally. Considering release planning, typical informally managed planning factors are: C1P1 resource capacities (resource demands during iterations), C1P2 priorities (importance of each requirement delivery), C1P3 dependencies (relations between requirements), C1P4 staged-delivery (delivery time of iteration timeboxes), and C1P5 maximal value (to choose the high priority one from different plans). The principles of agile development (see [35]) emphasize communication instead of rigorous planning. This characteristic can be explained also by the lack of easily applicable algorithmic solutions. Informal approaches work well in smaller projects. But the role of development coordinator is necessary in larger projects. As the size and complexity increase scheduling becomes a very complex process and advocates tool support [75, 76, 120]. The usual manual approach makes difficult to construct different plans to perform what-if analysis in order to select the best plan from the potential candidates. As a consequence, in traditional approaches the optimality of plans (i.e. delivering maximal business value) is heavily based on the manager’s right senses. Nevertheless, optimized project plans are crucial issues from the economic considerations of both customer and developer sides. Although, there can be found some general techniques to release planning and scheduling (see Sec. 3.1.1), they do not consider the staged-delivery (a release is made up of several iterations) characteristics of agile environments. However, these time factors strongly influence the selection and assignment of requirements to iterations. This staged-delivery approach requires in-depth investigation of feature dependencies that heavily influence on the results and this investigation cannot be carried out without proper tool support.
3.2 Decision Support in Release Scheduling

3.1.3 Objectives

To provide a systematic approach without omitting the human expert from the process, I elaborate a formal agile release planning problem description and algorithmic solution. The elaborated method considers the previous factors (C1P1-5), while the deficiencies of the informal and intuitive approach are mitigated. The main contribution can be seen as to make agile release planning process more objective and more qualified in terms of the results.

Firstly, my intention is 1) to provide a detailed conceptual model of agile planning and scheduling (C1.1). Then I elaborate a release scheduling method which intends to mitigate previous problems (C1P1-5) by 2) formulating release scheduling as a BINARY MULTIPLE KNAPSACK-based optimization model (C1.2). Finally, I provide 3) a solution to this model by a branch & bound algorithm (C1.3) to easily produce optimized release scheduling.

3.1.4 Outline

The rest of the chapter is arranged as follows: Sec. 3.2 introduces the elaborated i) conceptual model, ii) optimization model and iii) algorithm for agile release scheduling; Sec. 3.3 describes experiences; Sec. 3.4 discusses my solution; and finally Sec. 3.5 concludes the chapter.

3.2 Decision Support in Release Scheduling

In this section, first, I introduce the two general approach to release planning – namely date-driven and scope-driven planning (Sec. 3.2.1). Then I detail the interdependencies between features (Chapter 2 Sec. 2.5.3.2) that heavily influencing the scheduling process. After that, I present an elaborated conceptual model (Sec. 3.2.3) (C1.1) to provide a global view of agile release and iteration scheduling. Then I point out that agile release scheduling (RS) can be characterized as a specialized multiple knapsack problem, and I formulate a general optimization model for this problem (Sec. 3.2.6) (C1.2) – or in other words, a model for the schedule function (Alg. 1). Then, I present a solution to this optimization model in the form of a branch & bound algorithm (Sec. 3.2.7) (C1.3). Finally, a prototypic tool, and the analysis of my elaborated solution is presented.

3.2.1 Date-driven and Scope-driven Process of Release Scheduling

Release scheduling usually addresses two kinds of typical customer questions: 1) Fixed-time question: ‘How much of the features can be delivered by a given date?’ and/or 2) Fixed-scope question: ‘When can the selected (or all) features be delivered?’ The Fixed-time question mandates date-driven, while Fixed-scope one requires scope-driven scheduling approach. In fact, the former one can be interpreted as a temporal constrained version of the latter one.

In the following, a high-level algorithmic formulation of the scheduling process is presented. The pseudo-code of the two scheduling approaches is shown in one algorithm to be concise (Alg. 1). The lines in parentheses and marked with † mean that they are excluded from the scope-driven and included in the
3.2. Decision Support in Release Scheduling

date-driven case. While mark ‡ denotes the opposite case. In the program listing lowercase/uppercase letters with indices denote vectors/matrices (e.g. \( p_i, D_{j,j'} \)) and without indices, they mean scalars (e.g. \( l^R \)). While bold-faced types show concise (without indices) forms (e.g. \( D \)), and Gothic types denote sets (e.g. \( R \)).

In the require section the preconditions are given. At the beginning of the process, the release constraints – such as required release duration (\( l^R \): in days), available developers (\( R^* \) and their effectiveness factors (\( e_i \): proportion of his/her daily work on this release), and dependencies between features (\( D_{j,j'} \) – and business factors – priorities (\( p_j \) and efforts (\( w_j \): in person days) – are defined. The ensure section prescribes the post condition on the return value (\( X \)): every feature \( j \) has to be assigned to maximum one or exactly one iteration \( k \).

Algorithm 1 Date-driven and scope-driven agile release scheduling process

<table>
<thead>
<tr>
<th>Require:</th>
<th>/∗ Release duration /∗</th>
</tr>
</thead>
<tbody>
<tr>
<td>((l^R \in \mathbb{N}))†</td>
<td>/∗ Available developers and their effectiveness factors /∗</td>
</tr>
<tr>
<td>(i \in R^*, e_i \in [0,1])</td>
<td>/∗ Priorities, efforts and dependencies /∗</td>
</tr>
<tr>
<td>(j \in W, p_j \in N, D_{j,j'} \in {0,1})</td>
<td>/∗ Selecting developers /∗</td>
</tr>
<tr>
<td>Ensure:</td>
<td>/∗ Calculating the velocity of selected resources /∗</td>
</tr>
<tr>
<td>(\forall j \exists! k \left( X_{k,j} \in {0,1}\right) \text{xor} \left( X_{k,j} = 1 \right))</td>
<td>/∗ Assessment of iteration count and length of each iteration /∗</td>
</tr>
<tr>
<td>1: repeat</td>
<td>/∗ Calculating iteration velocities /∗</td>
</tr>
<tr>
<td>2: (i \in R \subseteq R^*)</td>
<td>/∗ Select and assign features to iterations /∗</td>
</tr>
<tr>
<td>3: (v^R \leftarrow \sum_i e_i : i \in R)</td>
<td></td>
</tr>
<tr>
<td>4: (k, l^R_k)</td>
<td></td>
</tr>
<tr>
<td>5: (e_k \leftarrow v^R \cdot l^R_k : k \in I(\land \sum_k l^R_k \leq l^R))†</td>
<td></td>
</tr>
<tr>
<td>6: (X \leftarrow \text{schedule}(p, w, D, c))</td>
<td></td>
</tr>
<tr>
<td>7: until (X) is satisfying</td>
<td></td>
</tr>
<tr>
<td>8: return (X)</td>
<td></td>
</tr>
</tbody>
</table>

The main block of the algorithm shows that the release scheduling is an iterative process. Several schedule alternatives – typically due to what-if-analysis – compete with each other until the decision-makers are satisfied (line 1 – 7). During scheduling, first the developers are selected (line 2) and their velocity (line 3) is calculated. Next, the iteration count of the release, the length of each iteration (in days) is assessed (line 4) and the iteration velocity (i.e. how many features can be delivered by the team in a given iteration) is calculated (line 5). The most complex and critical part of the process is to find a schedule (line 6) where the objective (delivering features with the highest priorities) is maximum while the constraints (i.e. \( p, w, D, c \)) are satisfied (line 7). From now on, I name the function – that the algorithm realizes – \( \Theta_{AR} : (W, I, D, R, c, p, w, l^R, l^R_k) \rightarrow X \) as agile release scheduling function. In short (see Sec.1.4.2):

\[
\Theta_{AR} : (W, C^R) \rightarrow X
\]  

In this process, the settings of \( i, k, \) and \( l^R_k \) factors remained informal, their values mainly depend on the development situation and local agile practices. Their typical values are between 4 to 12 people, 2 to 4 iteration counts, 1 to 4 weeks respectively [34, 67, 66]. It is appropriate to calculate team velocities (\( v^R \)) from developers’ effectiveness factors if historical values are not available, but as soon as they are available, usage of historical values provides more reliable plans [67].
3.2 Decision Support in Release Scheduling

3.2.2 Dependencies between Requirements

The complexity of scheduling arises from the interaction between requirements by implicit and explicit constraints. While the previous is given by the scarcity of resources, the latter one is emerged from different dependencies between requirements.

3.2.2.1 Mapping Requirement Dependencies to Feature Dependencies

Considering release scheduling two important types of dependencies should be identified [75]: 1) coupling \((C)\): requirements connected with AND dependency should be released jointly because they expect each other to serve a unit of functionality, 2) precedence \((P)\): requirements connected with \texttt{REQUIRES}, \texttt{CVALUE}, \texttt{ICOST} or \texttt{TEMPORAL} dependencies should be released in given temporal order (i.e. \(j'\) before \(j\)). Dependency XOR only influences preliminary requirement selection during the requirement’s specification phase and does not influence on release scheduling. Table 3.1 summarizes the previous mappings.

<table>
<thead>
<tr>
<th>Requirement dependency group</th>
<th>Type</th>
<th>Influence on selection (see [90])</th>
<th>Influence on RS</th>
<th>Feature dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality-related dependency</td>
<td>AND</td>
<td>✓</td>
<td>✓</td>
<td>(C)</td>
</tr>
<tr>
<td></td>
<td>REQUIRE</td>
<td>✓</td>
<td>✓</td>
<td>(P)</td>
</tr>
<tr>
<td></td>
<td>XOR</td>
<td>✓</td>
<td>×</td>
<td>–</td>
</tr>
<tr>
<td>Value-related dependency</td>
<td>CVALUE</td>
<td>✓</td>
<td>✓</td>
<td>(P)</td>
</tr>
<tr>
<td></td>
<td>ICOST</td>
<td>✓</td>
<td>✓</td>
<td>(P)</td>
</tr>
<tr>
<td>Time-related dependency</td>
<td>TEMPORAL</td>
<td>×</td>
<td>✓</td>
<td>(P)</td>
</tr>
</tbody>
</table>

Besides requirements’ implementation, defect repairs should also be ordered during scheduling thus I interpret these two dependencies (i.e. coupling and precedence) and call them feature dependencies (cf. Sec. 1.4.2).

3.2.2.2 Constructing Singular Feature Dependencies

Feature dependencies can be modeled as binary relations. Precedence dependency can be defined as partial order relation (PO) [150] (usually written as ‘\(\leq\)’) and it possesses the Eq.3.2a, 3.2c, and 3.2d properties from the following:

\[
\begin{align*}
\text{If } j & \in W, \text{ then } j \sim j. \quad \text{(reflexive)} \\
\text{If } j & \sim j', \text{ then } j' \sim j. \quad \text{(symmetric)} \\
\text{If } j' & \sim j \text{ and } j \neq j', \text{ then } \neg(j \sim j'). \quad \text{(anti-symmetric)} \\
\text{If } j' & \sim j \text{ and } j'' \sim j', \text{ then } j'' \sim j. \quad \text{(transitive)}
\end{align*}
\]

Analogically, coupling can be interpreted as equivalence relation [150] (usually written as ‘\(=\)’) so it splits the set of features up (i.e. partition) into strict subsets which group jointly releasable items. Therefore coupling possesses the Eq.3.2a, 3.2b, and 3.2d properties from the Eq.3.2.
3.2. Decision Support in Release Scheduling

There can be not just singular, but multiple dependencies between two features. It is obvious that similar multiple dependencies can be expressed by one singular relation since the properties of the dependencies are the same. Although, multiple different dependencies may result in conflicts in dependencies – though release scheduling deems necessary the unanimous instructions to the implementation order. Problematic cases are those where 1) anti-symmetric properties are in opposite directions (i.e. \( j' \leq j \) and \( j' \geq j \)) and/or 2) both anti-symmetry and symmetry properties are expected (e.g. \( j' \leq j \) and \( j' = j \)). Nevertheless, if we consider the fact that these relations express temporal dependencies on the time line (since the aim is to order items), then for both previous cases the proper singular relation is '\( = \)'.

3.2.2.3 Algorithm for Feature Package Graph Transformation

I have developed a `produceCCPrec` algorithm (Alg. 2) to produce feature package graphs \( G_{FP} \) from feature graphs \( G_F \). The algorithm first determines equivalence classes then draws the real precedence relation to produce \( G_{FP} \). In the program listing letters are interpreted as in Alg. 1.

**Algorithm 2** `produceCCPrec` algorithm

---

**Require:**

\[
\begin{align*}
&j \in W, \quad P_{j',j} \in 0, 1 \land P_{j,j} = 0 \land P \text{ is DAG} \quad \text{/* items */} \\
&C_{j',j} \in 0, 1 \land C_{j,j} = 0 \land C \text{ is SCG} \quad \text{/* precedences */} \\
&j \in W, \quad P_{j',j} \in 0, 1 \land P_{j,j} = 0 \land P^C \text{ is DAG} \quad \text{/* coupling */} \\
\end{align*}
\]

**Ensure:**

\[
\begin{align*}
&P^C_{j',j} \in 0, 1 \land P_{j,j} = 0 \land P^C \text{ is DAG} \quad \text{/* precedences on } W^C */ \\
\end{align*}
\]

1: \( \text{sets} \leftarrow \text{createSingletonSets}(C) \)
2: \( [\text{rows}, \text{cols}] \leftarrow \text{findListOfEdges}(C) \)
3: \( \text{for } i \leftarrow 1, \text{length(rows)} \) do
4: \( [a, b] \leftarrow \text{findSetOf}(\text{sets, rows}(i), \text{cols}(i)) \)
5: \( \text{if } \neg \text{isMember(sets\{a\}, sets\{b\})} \text{ then} \)
6: \( \text{sets\{a\}} \leftarrow \text{sets\{a\}} \cup \text{sets\{b\}} \)
7: \( \text{sets\{b\}} \leftarrow \emptyset \)
8: \( \text{end if} \)
9: \( \text{end for} \)
10: \( P^C \leftarrow [0]_{\text{sets}} \)
11: \( [\text{rows}, \text{cols}] \leftarrow \text{findListOfEdges}(P) \)
12: \( \text{for } i \leftarrow 1, \text{length(rows)} \) do
13: \( [a, b] \leftarrow \text{findSetOf}(\text{sets, rows}(i), \text{cols}(i)) \)
14: \( \text{if } a \neq b \land P^C_{a,b} \neq 1 \text{ then} \)
15: \( P^C_{a,b} \leftarrow 1 \)
16: \( \text{end if} \)
17: \( \text{end for} \)
18: \( \text{return } P^C \)

---

In the **Require** section the preconditions are given. Each \( j \) denotes features, and precedences between features can be represented by a precedence matrix where \( P_{j',j} = 1 \) means that feature \( j' \) precedes feature \( j \), otherwise \( P_{j',j} = 0 \). Both conditions \( P_{j,j} = 0 \) (no loop) and \( P \) is Directed Acyclic Graph (DAG) ensures that temporal constraints are not trivially unsatisfiable. Coupling is represented by a coupling matrix where \( C_{j',j} = 1 \) means that feature \( j' \) has to be delivered with feature \( j \), otherwise \( C_{j',j} = 0 \). Condition \( C_{j,j} = 0 \) expresses (no loop) and the single connected (SCG) graph property ensures that it is a simple graph. The **Ensure** section prescribes the postconditions on the return value \( (P^C) \): the resulted graph has to fulfill the properties of precedence relation as in the \( P \).
3.2. Decision Support in Release Scheduling

In the course of operation, first singleton sets (each one has only one element) and list representation (row and column pairs) is produced from coupling matrix $C$ (line 1, 2). The cycle iterates from 1 to the number of coupling relations in $C$ (line 3 – 9). In the cycle, at first the indexes ($a$ and $b$) of row and column pair are determined in the set $sets$ (line 4). Then if any of the two sets (indexed by $a$ and $b$) is not member of the other set the $isMember$ function returns $false$ (line 5), and the set indexed by $b$ is incorporated into set indexed by $a$ (line 6 – 7). By the end of the cycle, set $sets$ contains the equivalence classes of $G_F$.

As for the second part of the algorithm, firstly $P^C$ is initialized with zeros (no connection) in size of identical with the elements of the $sets$. Similarly to the previous part of the algorithm, list representation is produced from precedence matrix $P$ (line 10), then a same cycle is constructed (line 12 – 17). If the two indexes ($a$ and $b$) is not equal (not in the same set) and $P^C_{a,b}$ has not been set (line 14) then precedence is declared between sets $a$ and $b$ (line 15).

After termination $P^C$ defines precedences over $W^C$, where $length(P^C) = n^* (n^* \leq n : length(P) = n)$. The $createSingletonSets$, $findListOfEdges$ and $isMember$ functions need $O(n)$ computation time, however, the $findSetOf$ function needs $O(n * \log n)$ time. Thus, the complete algorithm requires $O(n * \log n)$ time.

Besides the previous listing (See Alg. 2), the Matlab version of the algorithm can be found in Appendix A Sec. C.1.

3.2.2.4 Visualization of Feature Dependencies

Since precedence and coupling relations have strong influence on release planning proper visualization can support human decisions in the course of the planning process. By representing requirements and their relations by nodes and edges in a kind of graph form may help to draw conclusions connected with release planning at a glance. In my presented visualization, precedence is denoted with the traditional directed edge notation, although to stress the difference between coupling and precedence I did not applied the usual non-directed edge notation for coupling, but draw nodes in cluster. The Figure 3.1 shows two examples of visualization. For instance, on the left, there is a coupling relation between nodes 36, 37, and there is a precedence relation between nodes 34, 35.

![Figure 3.1: Visualization of Release Planning Dependencies: Coupling with Cluster and Precedence with Directed Edge Notation.](image)

The notation in Figure 3.1 focuses on only the relations and the nodes show just the identifiers of requirements. In practical situations other data (such as denomination, priority, cost) may also be useful to be included.
3.2. Decision Support in Release Scheduling

In order to formulate the release scheduling optimization model, first, I have to identify the main concepts of agile scheduling. Next, I do not only present release but iteration scheduling concepts to provide a global view of agile scheduling. Thus, these concepts not only help to identify the objects and the subject of the optimization model but with the precise relationships it can also be used as database schema definition for agile release and iteration scheduling applications.

My elaborated conceptual model of agile release and iteration schedule (called Agile Release and Iteration Scheduling Model (ARISM)) is visualized with UML notation in Figure 3.2 and in Figure 3.3, and presented in Table 3.2, 3.3 [67, 163]. While the previous figure points out the specialization of the concepts (with inheritance relationships) to provide a general overview, the latter figure details the relationships between the notions.
### 3.2: Concepts of Agile Release and Iteration Planning Model (ARISM)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeliveryStage</td>
<td>is an abstract concept of delivery as in the agile environments the software is rolled out in stages (i.e. Releases and Iterations).</td>
<td></td>
</tr>
<tr>
<td>Project</td>
<td>is a planned endeavor, usually with specific Workproducts that are rolled out in several deliverable stages i.e. Releases by some Resources.</td>
<td></td>
</tr>
<tr>
<td>Release</td>
<td>produces (usually external) selected deliverable Features for the customer by a selected Team, and it usually contains 1-4 iterations. In case of date-driven planning the length (or deadline) of release is defined (l[\text{R}]) in working months/weeks/days.</td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td>is a development timebox in which intermediate deliverables i.e. Features are implemented. It is characterized by available developers and iteration length (ha\text{Length}: L[i[\text{I}]) – often expressed by iteration capacity (or velocity) has\text{Capacity}: e [c_k]: how many features can be delivered by the Team in a given iteration index (has\text{Index} \in \mathbb{K}) within the release.</td>
<td>L</td>
</tr>
<tr>
<td>Resource</td>
<td>is an abstract concept of human manpower selected for a given Project. A resource can be a Developer or a Team. Each resource is characterized its effectiveness factor (has\text{Effectiveness}: e[i]) that gives how effectively can take part in the project beside its other activities (e.g. other projects, support tasks). In case of developers its value is between in [0, 1], while in case of team the effectiveness factors of individual members are aggregated (i.e. e = \sum e[i]).</td>
<td>R</td>
</tr>
<tr>
<td>Developer</td>
<td>is the unit of human manpower. In iteration planning, developers (its index is has\text{Index}[\text{D}]) are allocated to (inverse of hasAl-locat\text{ion}[\text{D}]) low level workproducts i.e. TechnicalTasks.</td>
<td></td>
</tr>
<tr>
<td>Team</td>
<td>is a group of developers (\text{D}) that are selected to the realization of a Release from available Developers. During release planning, high level workproducts i.e. Features are realized by the Team.</td>
<td>T</td>
</tr>
<tr>
<td>Workproduct</td>
<td>is an abstract concept of deliverables. At the release planning level, deliverables are Features, while at the iteration planning level they are TechnicalTasks. Every workproduct (its index is has\text{Index}[\text{W}]: j) is characterized with its value for the customer which is denoted by priority (has\text{Priority}: p [p_j]).</td>
<td>W</td>
</tr>
<tr>
<td>Feature</td>
<td>is an abstract concept of deliverable. It is selected for a given Release and they can be classified into two kinds of set of elements according to its type: RequirementFeature, and DefectRepairFeature. Its realization usually needs several working days (Wd) manpower that is estimated by developers or some methods (has\text{Weight}: w [w_j]).</td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>is a deliverable that is value for the customer. A requirement can be new or changed (including functional and non-functional ones). In most cases requirements mandate several realization steps that may include cooperation of some developers.</td>
<td></td>
</tr>
<tr>
<td>DefectRepair</td>
<td>is a deliverable that fixes defects in former product variants, and in some cases it may include cooperation of some developers.</td>
<td></td>
</tr>
<tr>
<td>FeaturePackage</td>
<td>is a set of deliverable software module which contains the implementation of a given Feature.</td>
<td>M</td>
</tr>
<tr>
<td>FeatureChunk</td>
<td>holds together some Features (W) that must be delivered by one of the distributed team to provide minimal coordination and communication intensities. Its resource demand is aggregated value of its parts (\sum w_j). (See Sec. 3.2.7.1)</td>
<td>W^*</td>
</tr>
<tr>
<td>TechnicalTask</td>
<td>is a fundamental working unit accomplished by one developer. Proper coordination requires individually realizable working units thus each RequirementFeature and DefectRepairFeature should be broken down into several technical tasks. They usually require some working hour (Wh) manpower that is estimated by developers and denoted by duration (has\text{Duration}: d). Additionally, every technical task is characterized with its start/complete dates (has\text{StartDate}: S and has\text{CompletionDate}: C) to precise scheduling.</td>
<td>A</td>
</tr>
<tr>
<td>Product</td>
<td>is an abstract concept of the product elements. A product element can be a Software or SoftwareModule.</td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td>is a deliverable product which is made up of some SoftwareModule.</td>
<td></td>
</tr>
<tr>
<td>SoftwareModule</td>
<td>holds together some Features (W) that must be delivered by one of the distributed team to provide minimal coordination and communication intensities. Its resource demand is aggregated value of its parts (\sum w_j). (See Sec. 3.2.7.1)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.3: Relations of Agile Release and Iteration Planning Model (ARISM)

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasSubStage</td>
<td>denotes parent-child relationship between different kinds of DeliverStage.</td>
<td></td>
</tr>
<tr>
<td>hasMember</td>
<td>denotes parent-child relationship between Team and Developers.</td>
<td></td>
</tr>
<tr>
<td>hasModule</td>
<td>denotes parent-child relationship between Software and SoftwareModules.</td>
<td></td>
</tr>
<tr>
<td>hasPart</td>
<td>denotes parent-child relationship between Feature and TechnicalTasks.</td>
<td></td>
</tr>
<tr>
<td>isAssignedTo</td>
<td>signifies that a set of FeaturePackages are assigned to a given Iteration (X). This assignment is the result of the release scheduling ((\mathcal{S}_{AR})).</td>
<td></td>
</tr>
<tr>
<td>hasAllocationOf</td>
<td>means that the given TechnicalTask is allocated for a Developer (S). This allocation is the result of the iteration scheduling ((\mathcal{S}_{AI})). (It is the inverse relation of the usual resource allocation to work products.)</td>
<td></td>
</tr>
<tr>
<td>hasPreAllocationOf</td>
<td>means that the given TechnicalTask - before scheduling - maybe allocated for a Developer (a).</td>
<td></td>
</tr>
<tr>
<td>hasTeamPartitionTo</td>
<td>means that the given set of Features are partitioned to the given set of distributed Team. This partitioning is the result of the distributed agile release planning ((\mathcal{A}_{F})).</td>
<td></td>
</tr>
<tr>
<td>isCoupledWith</td>
<td>is a joint realization ((C_{j',j})) prescription between Features.</td>
<td>(C \subseteq D)</td>
</tr>
<tr>
<td>isPrecedencedBy</td>
<td>is a realization precedences ((P_{j',j})) between Features and between TechnicalTasks.</td>
<td>(P \subseteq D)</td>
</tr>
<tr>
<td>isImplementedIn</td>
<td>denotes that a given Feature in which SoftwareModule will be implemented ((\otimes)).</td>
<td></td>
</tr>
<tr>
<td>isSelectedTo</td>
<td>signifies that a set of Features are selected to a given Release.</td>
<td>(\otimes)</td>
</tr>
</tbody>
</table>
3.2. Decision Support in Release Scheduling

Generally, scheduling mandates defining who will realize what and when. Team, Feature and Release concepts answer to these questions in agile release scheduling (cf. Sec. 1.4.2). In this case, a Feature can be a Requirement or a DefectRepair; and a Release consists of several Iterations. Whereas, Developer, TechnicalTask and Iteration concepts answer to the previous questions in agile iteration scheduling (Sec. 1.4.3). Due to the structural similarity of release and iteration scheduling, these concepts are expressed with Resource and Workproduct abstract concepts. Additionally, schedule constraint must also be defined: resource constraints can be expressed by iteration velocity or the number of developers (Sec. 3.2.1), temporal constraints can be asserted as attributes (e.g. deadlines – Sec. 3.2.1) or Precedence and Coupling dependencies (Sec. 3.2.2.1). The distributed extension of agile release planning (see Sec. 5.3) introduces the Module concept with the related Implemented in relation mean a workproduct is implemented in a module.

In Figure 3.3, shaded objects (like Developer and TechnicalTask) pertain to iteration scheduling only – therefore the non-shaded objects constitute to the Agile Release Scheduling Model (ARSM) or the Distributed Extension of Agile Release Scheduling Model (DEARMS). Objects in italics point out abstract objects (cannot be instantiated), and the lower compartments of entities with double compartment give the list of attributes of each object.

3.2.4 A Prototype for Collaborative Agile Release Planning Data Collection

Previously presented ARSM conceptual model is realized by our MS SharePoint-based website1 (named SERPA™). SharePoint is browser-based collaboration and a document-management platform, and its capability includes creating different lists (as database tables). The previously constructed agile planning information model (see Figure 3.3) were implemented as SharePoint lists. In Appendix E the Figure E.7 shows the visual appearance of the prototype.

3.2.5 Mapping to Multiple Knapsack Problem

The optimized version of the release planning problem can be derived by selecting the extreme-valued plan from the potentially feasible alternatives. In this section, I specify agile release scheduling as a combinatorial optimization problem. In such a problem, we have to find the maximum valued solution from a finite but very large number of solutions. Generally, the BINARY KNAPSACK problem (BKP) (see Sec. 2.4.2) instance is specified by a knapsack capacity and a set of items (features) where each item has profit (business priority) and weight (required effort). The objective is to fill up the knapsack by selecting from among possible items those which give maximum profit while the total size of items does not exceed the knapsack capacity [166, 138]. An important generalization of the BINARY KNAPSACK problem is the BINARY MULTIPLE KNAPSACK problem (see Sec. 2.4.3) which arises when more than one knapsack is given with possible different capacities. The analogy between release scheduling and multiple knapsack problem can be explained as follows.

The team’s iteration velocity (cf. \( v^R \) in Sec.3.2.1) in an iteration stands for a knapsack capacity, while a feature’s resource demand (cf. \( w_j \) in Sec.3.2.1) represents an item weight. We can view each iteration (cf. \( k \) in Sec.3.2.1) within a release as a knapsack into which we can select and assign different deliverable features. Without loss of generality, we can ensure that the resource demand of each feature is less than

1This website is employed at Multilogic [164, 165].
3.2. Decision Support in Release Scheduling

team’s iteration velocity (i.e. knapsack capacity). I apply priority instead of profit to express values of features.

The two release scheduling question types (cf. Sec. 3.2.1) can be formed as:

1. **Fixed-time**: We have a fixed release date to which we determine an iteration count using the iteration velocity. In this case, the goal is to maximize the delivered value in the determined iterations (knapsacks).

2. **Fixed-scope**: We have a fixed scope to which we determine how many iterations are needed. In this case, the goal is to minimize the number of iterations (knapsacks) used.

It is obvious that the former one is similar to BMKP. Although, the latter one should be formulated with flexible packing approach: we are packing features until all selected requirements are assigned to iterations (not fixed iteration count). At the point of scheduling we assume that the number of candidate features is larger than what can be developed with the available resources. Therefore, an important common characteristic of both approaches is that – when we pack deliverables into iterations – we usually select the highest valued deliverables first in order to maximize the delivered values. It has an important practical advantage in the Fixed-scope case: it makes that possible to mitigate the emerging problems on the most valued deliverables early.

I extend the ordinary BMK problem and interpretation with the following elements to provide further computational capabilities for wide-ranging scheduling situations (cf. Sec. 3.1.2: C1P1-5): 1) flexible packing approach, 2) dependencies between features (cf. Chapter 2Sec. 2.5.3.2). From now on I call this extended problem as Agile Release Scheduling Problem (ARSP).

### 3.2.6 Formulating ARSP Optimization Model

Let us given a set of deliverable features \( j \) \((j \in W : |W| = n)\) with resource demands \( w_j \) and iterations \( k \) \((k \in I : |I| = o)\) with different iteration velocities \( c_k \) within a release. In case of Fixed-scope scheduling \( o \leq n \) (considering \( n \) features the iteration count cannot be more than \( n \)), and in case of Fixed-time scheduling \( o \triangleq \lceil FT/ \sum l_k \rceil \leq n \), where \( FT \) is the Fixed-time duration of the release and \( l_k \) is the length of iteration \( k \). Let assign each feature into one iteration so that the total required effort in iteration \( k \) does not exceed \( c_k \), and the number of iteration used as a minimum – while both precedence relations (matrix) \( P_{j,j'} \in \{0,1\} \) (where \( P_{j,j'} = 1 \) if \( j \) precedes \( j' \), otherwise \( P_{j,j'} = 0 \)) and coupling relations (matrix) \( C_{j,j'} \in \{0,1\} \) (where \( C_{j,j'} = 1 \) if \( j \) is coupled with \( j' \), otherwise \( C_{j,j'} = 0 \)) hold.

A possible mathematical formulation can be found in 3.3 (see definitions of \( p_j, w_j \) in Sec. 3.2.1). The equation marked with \( \dagger \) means the date-driven (some tasks may be omitted due to lack of resource) case while \( \ddagger \) denotes the scope-driven (every task must be done in some iteration) case.
Maximize  
\[ z = \sum_{k=1}^{m} \sum_{j=1}^{n} p_j x_{k,j} \]  
(3.3a)

subject to  
\[ \sum_{j=1}^{n} w_j x_{k,j} \leq c_k y_k \quad : k = 1 \ldots m \]  
(3.3b)

\[ k' - k \geq P_{j',j} \quad : x_{k',j'} = x_{k,j} = 1 \]  
(3.3c)

\[ \frac{1}{\lfloor |k' - k| + 1 \rfloor} = C_{j',j} \quad : x_{k',j'} = x_{k,j} = 1 \]  
(3.3d)

\[ (\sum_{k=1}^{m} x_{k,j} \leq 1) \oplus (\sum_{k=1}^{m} x_{k,j} = 1) \quad : j = 1 \ldots n, \forall j \exists! k \]  
(3.3e)

where \( y_k = 0 \) or \( 1 \), and \( x_{k,j} = 0 \) or \( 1 \), and

\[ x_{k,j} = \begin{cases} 
1 & \text{if } j \text{ is assigned to iteration } k \\
0 & \text{otherwise}
\end{cases} \]  
(3.4a)

\[ y_k = \begin{cases} 
1 & \text{if iteration } k \text{ is used} \\
0 & \text{otherwise}
\end{cases} \]  
(3.4b)

The equations denote: 3.3a) deliverable value maximization of release (i.e. summing up values of delivered features \( p_j \) in each iteration \( k \)), 3.3b) resource constraints (i.e. resource demands of delivered features \( w_j \) in iterations cannot be greater than the total effort in different iterations \( c_k \)), 3.3c) temporal constraints (i.e. if \( j \) precedes \( j' \) (\( P_{j',j} = 1 \)) then iteration index \( k' \) must be greater than or equal to \( k \)), 3.3d) coupling constraints (i.e. if \( j \) is coupled with \( j' \) (\( C_{j',j} = 1 \)) then iteration index \( k' \) must be equal to \( k \)), and finally 3.3e) item \( j \) can be assigned to exactly one (scope-driven case) or maximum one (date-driven case) iteration depending on the goal of the decision maker (cf. Sec.3.2.1).

I will suppose, as is usual, that the efforts \( w_j \) are positive integers. Without loss of generality, I will also assume that

\[ c_k \text{ is a positive integer} \]  
(3.5a)

\[ w_j \leq c_k \text{ for } \forall k, j \]  
(3.5b)

If the assumption (3.5a) is violated, \( c_k \) can be replaced by \( \lfloor c_k \rfloor \). If an item violates the assumption (3.5b), then the instance is treated as trivially infeasible. For the sake of simplicity, I will also assume that, in any feasible solution, the lowest indexed iterations are used, i.e. \( y_k \geq y_{k+1} \) for \( k = 1, 2, \ldots, n - 1 \).

### 3.2.7 Solving the ARSP Optimization Problem

For the previously formulated optimization model, I developed a BINARY MULTIPLE KNAPSACK-based (see Sec. 2.4.3) scheduling algorithm (Alg. 3). It is a branch & bound algorithm, which iteratively
3.2. Decision Support in Release Scheduling

selects and schedules an item (feature) into an iteration.

3.2.7.1 Constructing Feature Packages

The structural analysis of the ARSP model leads to the decomposition of the release schedule into a set $W$ of features, a set $P$ of precedence and a set $C$ of coupling relations. These elements can be represented as a feature graph $G_F = (W; P, C)$, where the precedence and the coupling relations are drawn as directed and non-directed edges respectively. Therefore, $G_F$ is a multigraph as the same two vertices may have two edges. One can notice that $G_F = G_P \cup G_C = (W; P) \cup (W; C) = (W; P \cup C)$, so $G_P$ and $G_C$ are spanning subgraphs of $G_F$ [150].

According to the coupling definition (cf. Chapter 2 Sec. 2.5.3.2) coupled features should be delivered together. Therefore, deliverable items may be interpreted as feature packages, where coupled features are joined. As a consequence, a graph $G_F$ has to be transformed into a feature package graph $G_{FP}$ where precedences are preserved. This $G_{FP}$ can be constructed in the following way.

For each $j \in W$ I define an equivalence class containing $j$ – denoted by $[j]$, where $j$ refers to the equivalence class representative element – to be the set of those elements which are accessible from $j$ through coupling edges. So, each item $j$ of $W$ belongs to some equivalence class ($j \in [j]$) and the union of all equivalence classes is $W$. Then I can say that the equivalence class (coupling relation) partitions $W$. As a consequence, I can define coupled items as $[j]$ and the set of them as $W_C \triangleq \bigcup[j]$. As for precedence relation, we have to construct a new precedence relation – namely $P^C$ – on $W_C$ considering $P$ on $W$. As for construction, we have to think of the following two consequences of the previous coupling: 1) some precedences are between items that are in the same partition $[j]$, and 2) some ones are in different partitions – let us say $[j]$ and $[j']$. The former one produces self (not valid) precedences (i.e. $P_{j',j} : j', j \in [j]$) in $W_C$, therefore we have to leave them out. The latter one, however, can produce possible multiple – valid precedences (i.e. $P_{j',j} : j' \in [j'], j \in [j]$). In case of multiple precedence we need to preserve just one of them.

Figure 3.4 shows a feature package construction example using post-mortem analysis results of a real life development [165]. In this figure, precedence is denoted with directed edge notation and coupling is drawn in a cluster. For instance, on the left, there is a coupling relation between nodes indexed with 17, 18, and there is a precedence relation between nodes 6, 11. In this case, after graph transformation (i.e. $G_F \rightarrow G_{FP}$), 13 dependent nodes are produced from the 19 – as the coupled nodes are merged. (Notice, 1) transformation reduces the indices in the resulted graph so nodes 1, 5 became 1, 2) node 17, 18 and 19, 20, 21 on the left have no precedence after transformation, so they are – 16 and 17 respectively – not shown on the right.)

![Figure 3.4: Illustration of Feature Package Construction.](image-url)
3.2.7.2 Algorithm for Feature Package Scheduling

Nowadays, branch & bound algorithms are the most common way to effectively find the optimal solution of knapsack problems. In our case, a BKP branch & bound algorithm was embedded into an iterative frame to solve the previously described ARSP problem. Up to now several algorithms proposed to solve the BKP, although the Horowitz-Sahni algorithm is one of the most effective [138]. I extended this algorithm with 1) multiple knapsack capabilities, 2) flexible packing approach (cf. Sec. 3.2.5), and 3) precedences between items (Chapter 2 Sec. 2.5.3.2). The first extension is realized by embedding the Horowitz-Sahni algorithm into an iterative frame, while the second and third ones are carried out by interpreting them as additional constraints.

The algorithm assumes that the items are ordered according to decreasing values of the priorities per resource demands, i.e. \( p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n \). The course of the elaborated algorithm is outlined as follows. It solves the BINARY MULTIPLE KNAPSACK problem as the solution of several BINARY KNAPSACK subproblems (see first while in the Alg. 3). In each subproblem solving, first the schedulable (not scheduled and not precedented) items are selected. Two kinds of moves produce the optimal solution of each subproblem: forward move inserts the largest possible set of new consecutive items into the current solution; backward move removes the last inserted item from the current solution. Whenever a forward move performed, an upperbound of the current solution is computed and compared to the best solution so far to check whether a new forward move may lead to a better solution otherwise backtracking performs. The algorithm stops when no further backtracking is possible.

An important feature of the algorithm is that it aims at precedences in a soft way: if not every item is packed and room is remained in a given iteration, it tries to put consecutive items (that are directly preceded by packed items) into the given iteration. Therefore, for example, even if there is precedence relation between two items, both may be put to the given iteration.

In the program listing letters are interpreted as in Alg. 1.
Algorithm 3 mksched algorithm (2/1)

Require:
\[ w_j \in W \] /* weights of each feature \( j \) */
\[ P_{j,j'} \in 0, 1 \land P_{j,j} = 0 \land P \text{ is DAG} \] /* precedences */
\[ p_j \in N, c_k \in N \] /* priority values and iteration velocities */

Ensure: \( \forall j \exists k \left( (X_{k,j}) \in \{0, 1\} \right) \) xor \( (X_{k,j} = 1) \)

1: \( X \leftarrow [0]_{\text{length}(w)}, [0]_{\text{length}(c)} \) /* assignment matrix initialization */
2: \( rlist \leftarrow \emptyset, slist \leftarrow \emptyset \) /* 'ready list' and 'scheduled list' initialization */
3: \( idx \leftarrow 1 \) /* iteration index initialization */
4: \( remcap \leftarrow \text{false} \) /* remained velocity is not utilized */
5: while \( |\text{findNotSchedItems(slist)}| \neq 0 \land |\text{length}(c) \geq idx| \) do
6: \( ns \leftarrow \text{findNotSchedItems(slist)} \) /* find not scheduled items */
7: \( rlist \leftarrow \text{findNotPrecItems(ns, P)} \) /* construct ready list */
8: if \( rlist = \emptyset \) then
9: \( \text{print} '\text{Infeasible schedule!}', \return \emptyset \)
10: end if
11: \( pn \leftarrow p(rlist), wn \leftarrow w(rlist) \) /* current priorities and resource demands */
12: \( n \leftarrow \text{length}(rlist) \) /* items in the 'ready list' */
13: \( x \leftarrow [0]_n, xn \leftarrow [0]_n \) /* best and current assignment vector */
14: \( z \leftarrow 0, zn \leftarrow 0 \) /* best and current values */
15: if \( remcap \) then
16: \( cn \leftarrow cnrem \) /* let current residual velocity equal to remained velocity */
17: else
18: \( cn \leftarrow c(n) \) /* let current residual velocity equal to iteration velocity */
19: end if
20: \( backtrack \leftarrow \text{false}, \text{opt} \leftarrow \text{false}, j \leftarrow 0 \) /* backtracking, optimum, search index */
21: while \( \neg \text{opt} \) do
22: \( u \leftarrow \text{computeUBound}(j) \) /* computing upper bound from \( j \) */
23: if \( z \leq (zn + u) \) then
24: \( \text{while } wn(j) \leq cn \) do
25: \( cn \leftarrow cn - wn(j), zn \leftarrow zn + wn(j), xn(j) \leftarrow 1 \) /* insert the largest set of consecutive items */
26: \( j \leftarrow j + 1 \) /* forward move may lead to better solution */
27: end while
28: if \( j \leq n \) then /* take away it from the current solution */
29: \( xn(j) \leftarrow 0 \)
30: \( j \leftarrow j + 1 \)
31: end if
32: else
33: \( backtrack \leftarrow \text{true} \) /* forward move doesn’t lead to better solution */
34: end if
35: end while
36: if \( zn > z \) then /* forward move produced better solution */
37: \( x \leftarrow xn \)
38: end if
39: \( xnidx \leftarrow \text{findAssigned}(xn) \)
40: \( k \leftarrow 0 \)
41: if isEmpty(xnidx) then /* backtracking is no more possible */
42: \( \text{opt} \leftarrow \text{true} \)
43: \( X(rlist(\text{findAssigned}(x))) \leftarrow 1 \) /* store scheduled items */
44: \( slist(rlist(\text{findAssigned}(x))) \leftarrow 1 \) /* update scheduled list */
45: end if
Algorithm 4 mksched algorithm (2/2)

46: if ¬isEmpty(findNotSchedItems(slist)) then /* not scheduled items */
47:     minnsi ← minWeightOfNotSchedItems(slist)
48:     cnrem ← cn - sum(findWeightOfAssigned(x)) /* remained vel. */
49:     if cnrem < minnsi then
50:         remcap ← false
51:         idx ← idx + 1 /* increment iteration */
52:     else
53:         remcap ← true /* remained velocity utilization */
54:     end if
55: else
56:     idx ← idx + 1 /* increment iteration */
57: end if
58: end while
59: else
60:     k ← max(xnidx) /* find the last inserted item */
61: end if
62: if ¬opt ∧ (backtrack ∨ j > n) then /* – backtracking – */
63:     cn ← cn + wn(k), zn ← zn - pn(k), xn(k) ← 0
64:     j ← k + 1
65:     backtrack = false
66: end if
67: end while
68: end while
69: return X

In the Require section the preconditions are given. Each \( w_j \) is the resource demand for item \( j \). Precedences between items are represented by a matrix where \( P_{j',j} = 1 \) means that item \( j' \) precedes item \( j \), otherwise \( P_{j',j} = 0 \). Both conditions \( P_{j,j} = 0 \) (no loop) and \( P \) is directed acyclic graph (DAG) ensures that temporal constraints are not trivially unsatisfiable. Additionally, priorities \( p_j \) and iteration velocities \( c_k \) are given. The date-driven case can be controlled with the length of \( c_k \) (cf. Sec. 3.2.1). The Ensure section prescribes the two kind of postconditions on the return value (\( X \)) depends on scheduling question (cf. Sec. 3.2.1, 3.3e).

During scheduling steps, first the initial values are set (line 1 – 4). The assignment matrix \( X \) contains the feature package assignments to iterations – initially it is set to a zero matrix (items × iteration count) (line 1). The algorithm uses a ready-list (rlist) and a scheduled-list (slist) to keep track of schedulable and scheduled items – these lists are set to empty sets (line 2). Finally, the iteration index (\( idx \)) is initialized (line 3) and the signal of remained velocity usage for soft precedence handling is initialized (line 4).

The first while iterates as long as there is any schedulable item and iteration (line 5). The algorithm selects unassigned items (ns) (line 6) then constructs the actual ready-list (rlist) from which the algorithm can choose in the current control step without violating any precedence constraint (line 7). If the ready list doesn’t contain any item the schedule is infeasible (line 9).

The second while iterates until the actual iteration is filled optimally (line 21). Before the cycle, the current list of priorities (\( pn \)) and resource demands (\( wn \)) are determined using the ready list (line 11). Then the current and the best solutions (assignment vector in the given iteration) and their values are initialized (line 13 – 14). If all the items are not packed to the given iteration then the current residual iteration velocity (\( cn \)) is equal to the velocity of an iteration (line 18) else it is equal to the remained velocity of the iteration (line 16) – see it later (line 48).
3.2 Decision Support in Release Scheduling

The third while (line 22) represents forward moves and iterates while the search index is not out of scope \((j > n)\) and backtrack is not necessary. Whenever a forward move performed, an upperbound from index \(j\) relating to the current solution is computed and compared to the best solution so far to check whether a new forward move may lead to a better solution (line 23 – 24) otherwise backtrack performs (line 34). If a forward move could lead to a better solution than the algorithm inserts the largest set of consecutive items into the solution (line 25 – 28). If the resource demands of the last item, where \(j \leq n\), exceeds the remained iteration velocity then it is set to 0 in the current solution (line 29 – 32).

If forward move produced a better current solution than the best solution so far, I update the best solution (line 37 – 39). If backtracking is no more possible (line 42) then the optimal solution of the subproblem was found. As a consequence, it stores the assigned items (line 44), actualizes the scheduled list (line 45), and deletes the no longer valid precedences (lines 58). Although, if there are not scheduled items (line 46), and if the remained velocity of the iteration – which is calculated by initial velocity minus weights of scheduled items (line 48) – is greater than the minimum weight of schedulable items (line 47, 49) then remained velocity utilization is set (line 53) else it is annulled and the iteration count is incremented (line 50 – 51). If the current solution is not empty, it calculates the last inserted item’s index (line 60) which is used during backtracking (line 62 – 66).

After termination, \(X\) contains the item assignments to iterations, where the number of nonzero columns denotes the packed iterations (cf. 3.3a):

\[
z \leftarrow \text{length} \left( \text{nonZeros} \left( \sum_{j=1}^{n} w_j x_{k,j} \right) \right)
\]

(3.6)

3.2.7.3 Solution Analysis

The presented branch & bound approach produces global optimal solutions. In the course of computation it builds a tree, where each node corresponds to the inclusion or omission of an item. As we have \(n\) items and \(o\) iterations, there will be \(2^n\) nodes and \(o\) trees so the time complexity is \(O(o*n^2)\). Since several packing combinations are not feasible, the tree has less nodes in practice so it provides sufficient results for practical applications \([138]\). As previously stated, it assumes that the items are sorted, if it is not the case, they can be re-indexed in \(O(n*\log n)\) (see Appendix F) time through an efficient sorting algorithm (e.g. Quicksort).

I need to underline that the backtracking capability of the presented algorithm ensures finding the global optimum. The precedences determine the selectable items set (see the ready list in line 7) at a given time, therefore the selection of a given item from this set may lead to suboptimal results as it determines the further search space. So, in other to correct the previous selection which did not lead to optimal solution, we have to backtrack and rethink our decision by select an other one. This process is realized in this algorithm as follows: if forward move does not lead to a better solution (line 34), it finds the last inserted item (line 60), and this item is taken out from the current solution (line 63), then it continues to find better solution with forward move (line 22). During this process the precedences of the given solution are maintained according to the scheduled items and the precedences (line 6 – 7).

Figure 3.5 illustrates the scheduling concept continuing the previous example (see Sec. 3.2.7.1). It shows the post-mortem release scheduling result based on a real life development situation using the Algorithm 3.
3.3. Experimental Results

Figure 3.5 shows the histogram of schedulable feature packages. The $x$-axis enumerates the estimated working days, while $y$-axis shows how many feature packages fall into these categories. Figure 3.5b depicts the scheduling results produced by mksched algorithm in stacked bar chart form: the previous feature packages are assigned to five iterations ($x$-axis) with capacities 30, 30, 30, 29.5, and 18 ($y$-axis). Bar colors on the Figure 3.5a point out how feature packages are distributed on Figure 3.5b, and in each bar, the values denote indices and the working days (in brackets) of the given feature package.

3.3 Experimental Results

In order to validate my contribution, first I carried out a post mortem analysis on seven real-life representative data sets extracted from a software development company [165], then I investigated my solution with the numerous representative generated data sets. The latter validation is described in Sec. 3.3.4, the former one is presented in the following.

The seven past release data sets were compared against the results of simulations applying the same inputs [167, 168]. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical in terms of organization size (6 developers), applied agile methods (Scrum-like development process) and techniques (XP development practices). At this organization, the release scheduling process is made up of the typical agile planning steps (see Sec.1.4.2).

Additionally, I also carried out simulations on 120 generated representative data sets – by varying parameters of the scheduling problem (see Sec. 1.4.2: resource capacities ($\mathcal{R}$) iteration capacity ($c$), release length ($l^R$), iteration length ($l^I_k$) and dependencies ($\mathcal{D}$)) – to get an insight into the performance and quality of the presented approach and to filter out the statistical staggering of different agile planning problems.

Finally, to obtain a proof-of-concept, I implemented a prototype as a scheduling toolbox in Matlab [169]. The Matlab version of the mksched algorithm (see Sec. 3) can be found in Appendix C Sec. C.2.
3.3. Experimental Results

3.3.1 Context and Methodology

IRIS is a client risk management system (approx. 2 million SLOC) for credit institutions for analyzing the non-payment risk of clients. It has been a continual evolution since its first release in the middle of 90s. The system was written in Visual Basic and C# the applied methodology was a custom agile process. The release scheduling process was made up of the following steps. First, the development coordinator used intuitive rules for selecting features from the backlog into a release. Then the team estimated on every feature and determined the number and the length of iterations – based on iteration velocity. Finally, the team distributed features into iterations considering priorities, resource demands, and dependencies. The team usually spent some hours on this last step.

3.3.2 Data Collection and Results

Seven data sets (Collateral evaluation, Risk assumption, Ukrainian deal flow I-II, Romanian deal flow I-III – respectively $R_A$ to $R_G$) were selected to make a comparison between the algorithmic method and the manual release scheduling carried out previously. These data sets were results of typical agile processes and as a consequence they were appropriate for analysis. The $R_C$ data set is used to present the concept in the previous example (Figure 3.4, 3.5a, 3.5b). All releases had same project members (6 developers), iteration length (2 weeks), iteration velocity (30 Story point), domain, customer, and development methodology, but they were characterized by different feature counts ($FC$), iteration counts ($IC$), buffer per releases ($BpR$) (for contingencies), and feature size (in working hours) per iteration ($FS_i$). Table 3.4 summarizes the variables $R_A$ – $R_G$ collected from the company’s backlog.

**Table 3.4: Historical release schedule values ($R_A - R_D$)**

<table>
<thead>
<tr>
<th></th>
<th>FC</th>
<th>IC</th>
<th>BpR</th>
<th>$FS_1$</th>
<th>$FS_2$</th>
<th>$FS_3$</th>
<th>$FS_4$</th>
<th>$FS_5$</th>
<th>$\sum_{i=1}^{5} FS_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A$</td>
<td>33</td>
<td>4</td>
<td>3.0</td>
<td>28.0</td>
<td>35.0</td>
<td>24.0</td>
<td>30.0</td>
<td>0.0</td>
<td>117.0</td>
</tr>
<tr>
<td>$R_B$</td>
<td>25</td>
<td>3</td>
<td>4.5</td>
<td>33.0</td>
<td>34.5</td>
<td>18.0</td>
<td>0.0</td>
<td>0.0</td>
<td>85.5</td>
</tr>
<tr>
<td>$R_C$</td>
<td>27</td>
<td>5</td>
<td>12.5</td>
<td>31.5</td>
<td>33.0</td>
<td>23.0</td>
<td>26.0</td>
<td>24.0</td>
<td>137.5</td>
</tr>
<tr>
<td>$R_D$</td>
<td>27</td>
<td>4</td>
<td>3.5</td>
<td>29.5</td>
<td>33.0</td>
<td>27.0</td>
<td>27.0</td>
<td>0.0</td>
<td>116.5</td>
</tr>
<tr>
<td>$R_E$</td>
<td>53</td>
<td>4</td>
<td>-6.5</td>
<td>32.0</td>
<td>34.0</td>
<td>26.5</td>
<td>34.0</td>
<td>0.0</td>
<td>126.5</td>
</tr>
<tr>
<td>$R_F$</td>
<td>26</td>
<td>4</td>
<td>0.0</td>
<td>36.0</td>
<td>27.0</td>
<td>26.0</td>
<td>31.0</td>
<td>0.0</td>
<td>120.0</td>
</tr>
<tr>
<td>$R_G$</td>
<td>53</td>
<td>5</td>
<td>-10.0</td>
<td>34.5</td>
<td>35.0</td>
<td>30.0</td>
<td>33.5</td>
<td>27.0</td>
<td>160.0</td>
</tr>
</tbody>
</table>

To determine the usefulness of my elaborated method, I used the historical data as input of the mksched algorithm (Alg. 3). This method made it possible to compare characteristics of the algorithmic approach against the manual one. Computed values ($R^*_A - R^*_G$) are shown in Table 3.5 (since $FC$, $BpR$ were the same as in Table 3.4 they are not shown).

**Table 3.5: Optimized release plan values ($R^*_A$ to $R^*_G$)**

<table>
<thead>
<tr>
<th></th>
<th>$IC$</th>
<th>$FS_1^*$</th>
<th>$FS_2^*$</th>
<th>$FS_3^*$</th>
<th>$FS_4^*$</th>
<th>$FS_5^*$</th>
<th>$\sum_{i=1}^{5} FS_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A^*$</td>
<td>4</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>27.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_B^*$</td>
<td>3</td>
<td>30.0</td>
<td>29.5</td>
<td>26.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_C^*$</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>29.5</td>
<td>18.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_D^*$</td>
<td>4</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>26.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_E^*$</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>29.0</td>
<td>26.0</td>
<td>5.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_F^*$</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>6.5</td>
<td>0.0</td>
</tr>
<tr>
<td>$R_G^*$</td>
<td>6</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>10.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
3.3. Experimental Results

3.3.3 Analysis

The analysis goal was to compare the manual and the optimized approaches using the same input variables. The following key questions were addressed: **C1Q1**: What are the staffing requirements over time?; **C1Q2**: How many iterations do we need per release?; and **C1Q3**: How buffers for contingencies are allocated?

I constructed $Y_k \triangleq \sum_{j=1}^{n} w_j x_{k,j}$ (i.e. sum of assigned features to iteration) as a result variable to answer our questions (C1Q1-3). First, I carried out Exploratory Data Analysis (EDA) [169] to get an insight into the data sets, then I performed descriptive statistical analysis to quantitatively compare the main properties of the two approaches. The results of the analysis are presented as follows.

3.3.3.1 Qualitative Analysis

The following EDA techniques (called 4P EDA) are simple, efficient, and powerful for the routine testing of underlying assumptions [169]:

1. run sequence plot ($Y_k$ versus iteration $k$)
2. lag plot ($Y_k$ versus $Y_{k-1}$)
3. histogram (counts versus subgroups of $Y$)
4. normal probability plot (ordered $Y$ versus theoretical ordered $Y$)

where $Y_k \triangleq \sum_{j=1}^{n} w_j x_{k,j}$ (i.e. sum of assigned features to iteration) were identified as result variables to test or questions (C1Q1-3).

The four EDA plots are juxtaposed for a quick look at the characteristics of the data (Figure 3.6). The assumptions (A1-4) are addressed by the graphics (Figure 3.6):

**A1**: The run sequence plots indicate that the data do not have any significant shifts in location but have significant differences in variation over time.

**A2**: The upper histogram depicts that the data are skewed to the left, there there is no significant outliers in the tails, and it is reasonable to assume that the data are from approximately a normal distribution. Contrary, lower one shows asymmetricity (skewed to the left heavily), data are more peaked than the normal distribution. Additionally, there is a limit (30) in the data of the algorithmic case that can be explained by the subject of the optimization.

**A3**: The lag plots do not indicate any systematic behavior pattern in the data of the historical case. Though, in the optimized case, there is a dense area at the surrounding of point (30, 30) which indicates efficient assignments: considerable amount of iterations are packed with the limit value (30) roughly. The horizontal and vertical lines can be explained by the fact that every release is ended with partial assignments (last iterations contain contingencies).

**A4**: In the upper diagram, the normal probability plot approximately follows straight lines from the first to the third quartiles indicating normal distributions. In contrast, the normality assumption is not reasonable in the lower case.
3.3. Experimental Results

Interpreting the above plots, I can statistically conclude that there is no correlation among the historical data (A3), while it follows approximately a normal distribution (A4), and the optimized approach points out correlation (A3), and yields more smooth release padding and less variance (A1, A2).

These statistical conclusions point out some important differences between the optimized and the historical release planning cases. On the one hand, apart from the last iterations in each release, all iterations were (almost) smoothly and fully padded that can be explained by the effective packing capability of the algorithm which yielded economically well exploited iterations (resources are not underloaded). Additionally, the algorithmic case also avoided resource overloading (exceeding the resource limit) which overload may lead to increasing delivery risk. In contrast, in the manual method some iterations were over- and others were underloaded (the paddings were considerable smaller or larger than the limit value).
3.3. Experimental Results

3.3.3.2 Quantitative Analysis

The qualitative analysis pointed out an important aspect of the algorithm usage in scope-driven situations (see Sec. 3.2.1): applying the algorithm on a given input set, without considering the scheduling parameters, the algorithm may result in ineffective paddings in the last iterations which constitute non-economical resource utilizations. This observation can be explained by the fact that the algorithm had to add one more iteration to the releases $R_E^*$ to $R_G^*$ to avoid resource overload (cf. Table 3.5 and Table 3.4).

(This effect cannot be observed in date-driven cases since it is usually assumed that there are more deliverable features than that can be delivered in a given time period – so all iterations are well padded.) So, it suggests that the algorithm yields an automatic overload protection due to the iteration velocity constraints ($c_k$), but underload protection requires human intervention by adjusting the input parameters of scheduling.

Actually, in real life situations, the project manager usually iteratively adjusts the parameters of scheduling to deliver the most valued features with as few resources (i.e. cost) as possible. The latter one can be realized with resource underload avoidance. Practically, this endeavor can be implemented by 1) decreasing the scope of the release to get rid of the last ineffective iteration, 2) increasing the scope of the release to fully pad the last iteration, or 3) decreasing the length of the last iteration. To simulate this project manager’s endeavor, I also constructed an other optimized case – denoted by $R^*_A-G$ – by leaving out the last iteration from the release in order to get rid of the ineffective iteration. I consider this case as a more realistic resource utilization than $R_A-G$ since it avoids resource underload. As scope reduction does not affect answering the C1Q1 and C1Q3, I used this case in the comparisons also.

The previous data sets were analyzed with descriptive statistics (D1-3) (Table 3.6) to point out the quantitative differences between the historical and optimized cases. These statistics (D1-3) not just point out the need of human intervention, but also support the suggested conclusions of the qualitative analysis:

<table>
<thead>
<tr>
<th>Case</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Variance</th>
<th>Range</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{A-G}$</td>
<td>18.0</td>
<td>36.0</td>
<td>29.7</td>
<td>19.7</td>
<td>18.0</td>
<td>-0.61</td>
<td>2.79</td>
</tr>
<tr>
<td>$R^*_{A-G}$</td>
<td>5.0</td>
<td>30.0</td>
<td>27.5</td>
<td>39.5</td>
<td>25.0</td>
<td>-2.40</td>
<td>7.28</td>
</tr>
<tr>
<td>$R^{**}_{A-G}$</td>
<td>26.0</td>
<td>30.0</td>
<td>29.9</td>
<td>0.1</td>
<td>1.0</td>
<td>-3.64</td>
<td>15.24</td>
</tr>
</tbody>
</table>

**D1:** despite the iteration velocity was 30, the release schedule – in the historical case – prescribed 36, which resulted in 20% resource overload (see Max column). The previously mentioned fact, namely the algorithm had to add one more iteration to the releases $R_E^*$ to $R_G^*$ in other to avoid resource overload, explains the 5 value in the Min column of the $R^*_{A-G}$ case. Contrary, in the $R^{**}_{A-G}$ case the worst resource utilized iteration (Min column) is economically better than the historical case by $(26-18)/30 = 27\%$.

**D2:** relatively large (4 times) skewness (measure of the asymmetry of the data around the sample mean) of the $R^*_{A-G}$ case (histogram in Figure 3.6) can be interpreted by the capacity constraints (see Eq.3.3b) and the objective (see Eq.3.3a). Additionally, comparing the historical case with the $R^{**}_{A-G}$ case it shows an even larger (6 times) skewness, that can be explained by the fact that the algorithm can pack the iterations in even better (cf. minimum values) if the manager considers the size of the development scope.
3.3. Experimental Results

D3: relatively large kurtosis (measure of how outlier-prone a distribution) of the $R^*_A$ case (histogram in Figure 3.6) can also be explained by the capacity constraints (see Eq.3.3b) and the objective (see Eq.3.3a) of the optimization. Considering the $R^*_A$ case, the higher kurtosis means less frequent extreme deviations which is economically better also.

Referring back to the Table 3.4, there were 4 iterations in the historical case where the resources were underloaded more than or equal to 20% ($FS_i \leq 24$), there were 7 iterations where the underloads were more than or equal to 10% but less than 20% ($24 < FS_i \leq 27$), and there were 2 iterations where the underloads were less than 10% but the paddings were not optimal ($27 < FS_i$ and $FS_i \neq 30$). Considering overload, there was 1 iteration in the historical case where the resources were overloaded more than or equal to 20% ($FS_i \geq 36$), there were 10 iterations where the overloads were more than or equal to 10% but less than 20% ($36 > FS_i \geq 27$), and there were 3 iterations where the overloads were less than 10% but the paddings were not optimal ($33 > FS_i$ and $FS_i \neq 30$). Contrary, in the $R^*_A$ case, the count of the underloaded iterations were 0, 1, and 3 in the same intervals respectively, and there was not any overloaded iteration due to the iteration velocity constraint ($c_k$) of the optimization model. The Table 3.7 summarizes the distances ($\Delta$) from the optimal value (30) expressed in percentages (how many iterations from the total iterations of the given case fell in the defined intervals).

<table>
<thead>
<tr>
<th>$\Delta \geq 20%$</th>
<th>$20% &lt; \Delta \leq 10%$</th>
<th>$10% &lt; \Delta \leq 0$</th>
<th>$\Delta = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^*_{A-G}$</td>
<td>17%</td>
<td>59%</td>
<td>17%</td>
</tr>
<tr>
<td>$R^<em>_{A-G</em>}$</td>
<td>0%</td>
<td>4%</td>
<td>12%</td>
</tr>
</tbody>
</table>

So, in the historical case, 17%, 59%, 17% of the total 29 iterations were over- or underloaded more than or equal to 20%, more than or equal to 10% but less than 20%, less than 10% respectively. Only 7% of the total iterations were optimal. In contrast, in the $R^*_{A-G*}$ case, 0%, 4%, 12% of the total 25 iterations were over- or underloaded more than or equal to 20%, more than or equal to 10% but less than 20%, less than 10% respectively. But 84% of the total iterations were optimal. Comparing these values, it can be stated that the algorithmic approach with human intervention could provide a considerably better resource utilization than the manual approach. The better padding of $R^*_A$ cannot be explained by the scope reduction since historical paddings showed normal distribution: we could expect similar padding in the historical case if the scope was smaller (Sec. 3.3.3.1).

As a consequence, the staffing requirements (cf. C1Q1) showed more smooth and fully padded iterations (Figure 3.6) in the algorithmic cases (both $R^*_{A-G}$ and $R^*_{A-G*}$). It means that the algorithm strives to 1) prevent resource overload – which often yields increasing delivery risks, and 2) underload – which captivates economically badly utilized iterations. It is important to note that, totally padded iterations cannot be achieved in every situation due to the dependencies, and/or the relationship between the size of features and the iteration capacities. Iteration counts per releases (cf. C1Q2) of the algorithmic case $R^*_A$ exhibited slight differences: in some situations the algorithm had to add one more iteration to prevent resource overload. Therefore, in real life situations, the algorithm should be used in an iterative manner by altering the schedule parameters – which leads to economically better schedules – cf. $R^*_{A-G*}$.

Finally, if we consider the buffer per releases ($BpR$) that is used for contingencies in practice, we can realize major differences also. In the historical case, they were allocated sporadically: they were considered in some cases (positive $BpR$ values), and in other cases they were not considered (negative $BpR$ values) – see Table 3.4. Their allocations in time were also occasional – see Run sequence plot in Figure 3.6.
3.3. Experimental Results

The $R^*_{A\rightarrow G}$ case points out the buffer allocation property of the algorithm: time buffers (cf. C1Q3) are moved to the end of releases due to the optimality criteria (packing as many items into the iteration as possible). So, if the project manager wants to allocate a time buffer to a given release he/she has to decide on the size of the buffer by determining the deliverable feature set and iteration count, and the algorithm automatically allocate it to the end of the release. (Please note, a simple reordering of the iterations is not a satisfactory solution because the dependencies must be considered.) This characteristic indicates that the contingencies are moved to the end of the release while dependencies are considered by the algorithm, which is more advisable to mitigate risks of delivery slippage [170].

3.3.4 Computational Benchmarking

Problem solving time of the historical data sets, which can be considered as medium-sized problems, took less than one second with my prototypic tool. Therefore, to give some orientation about the performance and the generated plan qualities of the mksched algorithm on larger problems, I carried out simulations with the guidance of [139].

To get a more nuanced picture of the algorithm I considered several groups of randomly generated instances of release scheduling to reflect special properties that may have influenced the solution process. In all instances the resource demands ($w_j$) were uniformly distributed in the data set of $\{0.5, 1, 2, 3, 5, 8\}$ (reflected the typical resource demands in Story points [67]). The values ($p_j$) were expressed as a function of the resource demands – yielding typical properties of each group such a way that $p_j$ was chosen randomly from $[w_j - w_j/10, w_j + w_j/10]$ interval. It can be explained by the fact that, typically the value differs from the resource demand by only a few percents, since it is well-known that the return of investment (ROI) is generally proportional to the sum invested within some variations (they are highly correlated) [36]. To enlarge the problem size, I chose total effort in each iteration ($c_k$) equal to 50 which can be considered as a typical team velocity of a large agile team (cca. 10-people) [67]. I also generated precedence and coupling relations between the items – 30% and 10% of the feature count in a given iteration respectively – analogically to the collected data sets (Sec. 3.3.2).

3.3.4.1 Solving Time

The behavior of the algorithm was considered for different problem sizes $n = \{50, 100, 200, 500\}$, and it was run 30 times to calculate the mean solving time. The size 50 is similar to the historical data sets (Sec. 3.3.2), but 500 practically never occurs since it would require to plan 10 iterations ahead for a large agile team (the typical value is 4 [67]). Additionally, the algorithms were run in three cases: 1) without dependencies (\((P \cup C)\)), 2) without couplings (\(C\)), and 3) with dependencies (\(P \cup C\)) in other to get insight into their effects on the solving time. All tests were run on an Intel Pentium 4, 2.2 GHz, 4GB memory, MS Windows XP SP3. The results are presented in Table 3.8, where $I$ points out how many iterations were needed to compute the solution, and $ST$ denotes running time (in milliseconds) of the algorithm.

Based on the results in Table 3.8, we can conclude that: 1) the more realistic case ($ST^{(P\cup C)}$) was acceptable since the computation time was less than 4 seconds even in case of large instances (500 features); 2) the $\\< (P \cup C)$ and $\\< C$ cases were exponential to the item count (roughly $ST \propto n^{1.1}$ to $1.5$) as it was expected (see the $O(o \ast n^2)$ asymptotic upper bound in Sec. 3.2.7); 3) the $\\< C$ case was harder
3.3. Experimental Results

Table 3.8: Simulation results

<table>
<thead>
<tr>
<th>n</th>
<th>I(P\cup C)</th>
<th>ST(P\cup C)</th>
<th>I(C)</th>
<th>ST(C)</th>
<th>I(P\cup C)</th>
<th>ST(P\cup C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>234</td>
<td>90</td>
<td>16545</td>
<td>1742</td>
<td>6443</td>
<td>651</td>
</tr>
<tr>
<td>100</td>
<td>1690</td>
<td>208</td>
<td>53020</td>
<td>5565</td>
<td>6883</td>
<td>711</td>
</tr>
<tr>
<td>200</td>
<td>2058</td>
<td>684</td>
<td>72880</td>
<td>7653</td>
<td>17090</td>
<td>1812</td>
</tr>
<tr>
<td>500</td>
<td>33164</td>
<td>4413</td>
<td>273987</td>
<td>29392</td>
<td>34232</td>
<td>4116</td>
</tr>
</tbody>
</table>

than the \((P \cup C)\) case – roughly one order of magnitude, which can be explained by the complexity of precedence constraint handling; and finally 4) the \(P \cup C\) case was roughly linear to the problem size, which can be interpreted by the fact that constructing feature packages resulted in smaller problems (fewer item counts). Therefore it required less computation power.

As a consequence, the hardest problems are those ones which have many precedences (as precedence negatively affects the performance). Contrary, coupling relations positively influence the performance, since they decrease the number of schedulable items – not considering that extreme cases when there are too many coupling relations and the constructed feature packages exceed the total capacity of the given iteration, so the problem becomes infeasible (see 3.5b). This extreme situations may be handled by partitioning the packages into smaller ones by diminishing the count of coupling relations. As a consequence, I can state that, my elaborated algorithm could compute schedules very quickly (within some seconds) both on the historical and on the hypothetical agile release instances.

Considering the 'simple' binary multiple knapsack case (\((P \cup C)\)), which does not contain my additional constraints (see Sec.3.2.7), my algorithm’s solving time was worse by one or two magnitudes than the solving time of the state-of-the-art public knapsack algorithms that can be found in [139] (Chapter 5). Specifically, the elaborated algorithm’s solving time was worse by one magnitude than the solving time of the Horowitz-Sahni algorithm. This slowing-down can be explained by two things. On the one hand, my elaborated algorithm requires additional iterations and administration to aim at the introduced extensions – particularly the multiple knapsack capabilities and the precedence handling (see Sec. 3.2.7). The additional iterations, that are emanated from the multiple knapsack capabilities, are manifested in the external loop of the algorithm (see line 5 – 68 in Alg. 3.2.7.2). Actually it is also expressed in the \(o\) parameter of the algorithm time complexity (Sec. 3.2.7). (Please note, the time complexity of the Horowitz-Sahni algorithm and the elaborated algorithm is \(O(n^2)\) [138] and \(O(o*n^2)\) (Sec. 3.2.7) respectively.) On the other hand, the additional administration can be explained by the precedence handling, namely maintaining ready-list and scheduled-list (see line 6 – 7), of the elaborated algorithm.

Revert to the comparison of my algorithm and the state-of-the-art public knapsack algorithms, I have to underline the fact that I implemented my algorithm on the Matlab platform, which executes the code in an interpreted way, therefore, compiled versions (e.g. implemented in C) are expected to be faster with one order of magnitude at least. Even if the worst comes to the worst and my algorithm would not be accelerated by the compilation in that degree, my algorithm is able to handle the introduced constraints of agile release scheduling while it computes the results within an acceptable response time.

3.3.4.2 Quality of Plans

The objective function (see 3.3a) points out the maximal cumulative business value delivery endeavor of the ARS optimization model. This model tries to deliver maximal cumulative business value in each iteration: the greatest value in the first iteration, the second greatest one in the second iteration and so on.
3.3. Experimental Results

Let us given a set of deliverable features \( j \ (j \in W : |W| = n) \) with priorities \( p_j \) (reflect the business value for the customer), resource demands \( w_j \), and iterations \( k \) with different iteration velocities \( c_k \) within a release (see Sec. 3.2.6). Assignments of features to iterations depend on the iteration velocity (\( c_k \)), the precedence (\( P \)), and the coupling (\( C \)) constraints of the model while the objective function is considered. In spite of the fact that the algorithm produces a global optimal solution, in Section 3.3.3.2 (see Table 3.7) it is revealed that the algorithm could not achieve complete utilization of resources (i.e. fully packed iterations) in every scheduling case owing to the constraints of the model. Thus, in this section, we investigate this distance (\( \Delta \)) from the maximal value expressed in percentages (how many iterations from the total iterations of the given case fell in the defined intervals) as a function of the adequate scheduling parameters (i.e. \( n, c_k, C \) and \( P \)). (Please note, \( w_j \) and \( p_j \) parameters are not considered. As the former one interacts with \( c_k \) and the latter one does not play a role in capacity filling.)

Let us define the quality of release plans in terms of the distances (\( \Delta \)) from the theoretical maximal resource utilization (i.e. from the \( c_k \)). This distance characterizes the underload of iterations only, since overloads are avoided by the \( c_k \) constraint of the model (see Sec. 3.3.3.2).

The previous randomly generated instances of release data were analyzed to measure the quality of the release plans. I investigated the release plans considering the previously defined item counts (\( n \in \{50, 100, 200, 500\} \)), and the \( c_k \in \{30, 40, 50\} \) instances to analyze the influences of different iteration capacities (small, normal, large capacity – respectively) on the quality of the plans. The algorithms were run in three cases as before: 1) without dependencies (\( \langle P \cup C \rangle \)), 2) without couplings (\( \langle C \rangle \)), and 3) with dependencies (\( P \cup C \)) in other to get insight into their effects on the quality. The algorithms were run 30 times to calculate the average distances of the defined intervals. The Table 3.9 summarizes the distances (\( \Delta \)) in percentages. Please note, to save space in the Table, the previously defined distance intervals (see Table 3.7) are abbreviated as \( \delta_1 \triangleq (\Delta \geq 20\%) \); \( \delta_2 \triangleq (20\% > \Delta \geq 10\%) \); \( \delta_3 \triangleq (10\% > \Delta > 0) \) and \( \delta_4 \triangleq (\Delta = 0) \). Additionally, the average distance in an interval is denoted with \( \bar{\Delta} \). Considering these intervals along with the investigated parameters (\( n, c_k, C \) and \( P \)), we can observe that:

<table>
<thead>
<tr>
<th>n</th>
<th>( c_k )</th>
<th>( \langle P \cup C \rangle )</th>
<th>( \langle C \rangle )</th>
<th>( \langle P \cup C \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>50</td>
<td>0% 0% 2% 98%</td>
<td>5% 0% 0% 95%</td>
<td>0% 1% 0% 99%</td>
</tr>
<tr>
<td>40</td>
<td>0% 0% 0% 100%</td>
<td>6% 5% 4% 85%</td>
<td>0% 3% 5% 92%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0% 0% 0% 4% 96%</td>
<td>10% 0% 0% 82%</td>
<td>3% 0% 11% 86%</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>40</td>
<td>0% 0% 0% 100%</td>
<td>2% 0% 2% 96%</td>
<td>0% 0% 2% 98%</td>
</tr>
<tr>
<td>30</td>
<td>0% 0% 0% 2% 98%</td>
<td>3% 0% 3% 94%</td>
<td>0% 0% 2% 98%</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>50</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
</tr>
<tr>
<td>40</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 1% 99%</td>
<td>0% 0% 0% 100%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0% 0% 0% 100%</td>
<td>1% 0% 1% 99%</td>
<td>0% 0% 0% 100%</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>0% 0% 0% 100%</td>
<td>1% 0% 0% 99%</td>
<td>0% 0% 0% 100%</td>
</tr>
<tr>
<td>40</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
<td>0% 0% 0% 100%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.9: Simulation results of plan quality

PQ1: looking at the \( \bar{\Delta} \) values in the table, one can realize that \( \langle P \cup C \rangle \) case produced the highest value in the \( \delta_1 \) column and the lowest values in the \( \delta_{1-3} \) ones, therefore, it provided the best resource utilization. Comparing the remained cases, the \( (P \cup C) \) case is better than the \( \langle C \rangle \) case. This order can be explained by the complexity of precedence handling and the different feature sizes. The \( \langle P \cup C \rangle \) case did not aim at any dependencies, so it enabled considerably larger search space for the algorithm. (Generally, the larger search space leads to greater variations of packing and
3.4 Discussion

often may lead to a better solution.) The \((P \cup C)\) case addressed precedences and couplings where the latter one constituted fewer precedences and larger item sizes (see Sec. 3.3.4.1). Fewer precedences positively affect the finding of better packing due to the larger search space. Whereas, the larger item sizes negatively influence on the packing, since it results in fewer item counts therefore smaller search space. Finally, the \((P \cup C)\) case dealt with the greatest number of precedences that constituted the worst packing.

**PQ2:** pondering the item counts \((n)\), we can notice that the more items the better packing. The most significant difference can be seen between the \(n = 50\) and \(n = 500\) instances. In \(n = 50\) instances (including \(\setminus (P \cup C), \setminus C\) and \((P \cup C)\) cases), the value of the interval \(\delta_4\) was between 82\% and 100\%. While in \(n = 500\) instances, the value of the interval \(\delta_4\) was between 99\% and 100\%. This phenomenon can be explained by the fact that greater item counts constitute larger search space.

**PQ3:** considering the iteration capacity \((c_k)\), we can realize that the capacity positively influences on the packing under a given item count (i.e. \(n \leq 100\)): the larger the capacity the better the packing. This observation can be explained by two things. Firstly, the larger the capacity the greater the variation of packing due to the greater search space. Secondly, the greater item counts – above a certain level (cca. \(n \geq 100\)) – can compensate the negative effect of the capacity constraint.

Interpreting the observation PQ1-PQ3 we can statistically conclude that 1) the number of precedences and couplings (PQ1) negatively correlate with the degree of resource utilization – therefore, the degree of release plan quality. On the other hand, both the number of items (PQ2) and the extent of iteration capacity (PQ3) positively correlate with the degree of plan quality. As a consequence, these parameters should be considered by the project manager during release planning to provide a satisfactory quality level.

One can notice that the \((P \cup C)\) case with \(n = 50\) and \(c_k = 30\) parameters is similar to the post-mortem optimized case \((R^*_{A.C})\) case – see Table 3.7). I think of this type of problem as a small-medium-sized agile release scheduling problem. Actually, I also tried to carry out simulations of the \((P \cup C)\) case with \(n = 50\) and \(c_k = 20\) case to simulate small sized problems. But in a number of cases the algorithm met with infeasible scheduling problems due to the relatively large size of coupled items (i.e. in many cases the schedulable items were greater than the capacity of release). This event very rarely happened at parameter \(c_k = 30\) and therefore, in this case, I considered them as outliers. In real life situations, this problem may be tackled with either partitioning of the items or increasing the iteration time period that yields greater capacities. As a consequence, I can conclude that the feature size may influence on the plan quality also: the smaller items the better quality. This observation is particularly important in small-sized agile release scheduling problems.

Comparing all the differently parameterized simulations with the historical case, despite the simulation problems were more complex than the historical one, the algorithmic approach could outperform the manual approach in terms of release plan quality (cf. Sec. 3.3.3.2).

3.4 Discussion

Release planning is an activity concerned with the implementation of the selected features (what aspect) in the next version of the software. Complexity of planning arises from the interaction between features
3.4. Discussion

by implicit and explicit constraints. While the previous is given by the scarcity of resources (available developers), the latter one is emerged from different dependencies (cf. Chapter 2 Sec. 2.5.3.2) [115, 90]. Without dependencies, release planning can be considered as an BINARY KNAPSACK problem, which is \( \mathcal{NP} \)-hard [138] (see Appendix F.2.1). With dependencies planning becomes a more complex problem. Though, there are some proposed techniques to release planning, they do not consider the staged-delivery (when) characteristics of agile environments – however, this timing factor strongly influences the selection and assignment of features to iterations. In fact, when both what and when factors are considered, the problem becomes scheduling so I called it – agile release scheduling.

In agile environments, all scheduling factors (C1P1-5 in Sec. 3.1.2) are managed informally in planning meetings. Informal approaches work well in smaller projects, however as the size and complexity increase planning becomes a very complex, time consuming process and advocates tool support [75, 76]. To address this situation, I elaborated a combinatorial optimization model and algorithm as a solution.

First, I identified two release scheduling dependencies (i.e. coupling and precedence) between features (cf. Chapter 2 Sec. 2.5.3.2) that affect implementation sequences. I concluded that five dependencies can be interpreted as release scheduling dependencies from the six types of dependencies identified in [90]. Whereas XOR is heavily used in practice during requirement selection, I ignore it from my solution since release planning and scheduling practically happen after the scope is specified so when the selection among alternatives is previously done. Goal models have been found to be effective for determining the scope of the scheduling [171, 172, 173]. They can help to identify variability at the early requirement’s phase by capturing alternatives by which users and/or customers can achieve their goals.

Secondly, to formulate the agile scheduling model, I identified the main concepts of agile release and iteration scheduling (ARISM) (cf. Sec. 3.2.3) (C1.1). Although, there can be found partial conceptual models in [E13, C5], but a full-scaled model could not be found. So, I did not only present release but iteration scheduling concepts to provide a global view of agile scheduling. These concepts not only helped to identify the objects and subject of the optimization but with the precise relationships it can also be used in the design of agile release and iteration scheduling applications.

I formulated agile release scheduling problem (ARSP) as an extension of the BINARY MULTIPLE KNAPSACK-based optimization problem [138] (C1.2) – considering the previous factors (C1P1-5). My elaborated solution covers wide-ranging release scheduling situations with the expression of: 1) date-driven/scope-driven scheduling (cf. Sec. 3.2.1), 2) dependencies between features (cf. Chapter 2 Sec. 2.5.3.2). I called this as agile release scheduling problem (ARSP). This interpretation made it possible to adapt efficient global optimization algorithms – I utilized the search concept of Horowitz-Sahni’s algorithm [138] – to solve ARS problems. The developed algorithm (mksched), in terms of 1) staffing requirements (cf. C1Q1): it showed more smooth and fully padded iterations that mean that the algorithm strives to 1) prevent resource overload – which often yields increasing delivery risks, and prevent resource underload – which captivates economically badly utilized iterations; 2) iteration counts per releases (cf. C1Q2): it pointed out that to avoid resource underload it should be used in an iterative manner by the project manager; finally, 3) release buffers (cf. C1Q3): it supports lower level risk of delivery slippage [170]. Moreover, the easy and fast computation – even on large problems (Sec. 3.3.4) – allows 1) generating alternative schedules by utilizing what-if analysis to tailor the best schedule for the specific project context and 2) considering the users’ and/or customers’ feedbacks by altering constraints and priorities; both lead to more informed and established decisions by selecting the best schedule from the alternatives.
3.4. Discussion

However, several optimization algorithms are publicly available (see in [138, 139]) and they can solve common knapsack models very efficiently, the previously presented ARS optimization model requires a custom-designed algorithm (an extended version of the Horowitz-Sahni’s algorithm [138]) due to the followings:

1. **Scheduling capability**: Common knapsack models can calculate the next releasable items in contrast with my elaborated solution, which can handle many (theoretically any) iterations immediately – due to the demand of staged-delivery approach of agile environments. As a consequence, my (ARS model and) mksched algorithm not just determines the what-aspect, but the when-aspect also (in which iteration the selected items will be delivered). Therefore, I named this model as agile release scheduling.

2. **Soft precedence**: Multi-knapsack algorithms generally do not handle any precedence, so I had to complement it. This extension can handle precedences internally in each iteration: even if there is precedence relation between two items both may be put into a given iteration – as far as there is free capacity in the iteration. Consequently, this solution often leads to a far better solution than the usual precedence constrained solutions (e.g. in [120] and in [E13]) which ones aim at precedences externally (i.e. if an item is put into iteration \( k \) then its successor must be put into iteration \( > k \)).

The presented ARS optimization model can be also implemented by several optimization tools (such as CPLEX [174], AIMMS [175]), and they may provide better solving times (but not better quality since my algorithm finds the global optimum). However, there may be some practical drawbacks using these packages: 1) optimization problem modeling with these tools requires special mathematical modeling and/or ILP tool knowledge in comparison to implementing the presented algorithm in any popular programming language, and the lack of this knowledge is often a barrier to put the elaborated models into practice; 2) integrating the callable tool libraries may be difficult (e.g. CPLEX connector is not available); finally, 3) buying a tool (e.g. CPLEX, AIMMS) that is made by professionals in other to solve just one problem is rather expensive (typically \( n \ast \$1000 \)).

We can look at the elaborated solution as an extension to the readily available agile planning tools (such as [161] or [162]) which helps collecting the planning data (features, required effort, team velocity, etc.), but they support only manual release planning. (The planning data were collected through a popular web-based site [164].) Therefore, with this extension, I believe that one can produce agile release schedule easily based on the collected data. We also expect that my elaborated method requires a little more effort (some minutes) in set up time in expressing dependencies between deliverable features, but it produces optimal schedule within seconds – in contrast to the manual scheduling which required some hours (see Sec. 3.3.1). But actually, the major difference is not the required effort but the quality of the schedule: the optimized approach provides a higher quality schedule (i.e. avoiding under- and overload) and makes it possible to re-schedule it any time within seconds in other to support what-if analysis or to adapt the schedule to a changed situation.

The results suggest that the project manager with this algorithm could realize less risky (not overloaded and better time buffer allocated) and more economical (rarely slightly underloaded) release schedules by altering the input parameters of the algorithm. The presented conceptual model of agile release and iteration scheduling (cf. Sec. 3.2.3) points out that this approach can be complemented with an iteration-level scheduling i.e. allocating TechnicalTasks to Developers. Additionally, since the presented solution pertains to single team release scheduling only, it can also be extended with multi-team scheduling.
3.5 Conclusions

In this chapter, I presented my novel concepts, models and algorithms for agile release planning. The elaborated method shows more smooth and fully padded iterations (> 80% of the total iterations are optimal), and prevents resource overload comparing to the traditional approaches.

I carried out a post mortem analysis on seven real-life representative data sets extracted from a software development company [165] to validate my contribution. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical in terms of organization size (6 developers), applied agile methods (Scrum-like development process) and techniques (XP development practices). At this organization, the release scheduling process is made up of the typical agile planning steps (see Sec.1.4.2).

Additionally, I also carried out simulations on 120 generated representative data sets – by varying parameters of the release planning problem (see Sec. 1.4.2: resource capacities ($R$), iteration capacity ($c$), release length ($l_R$), iteration length ($l_I$) and dependencies ($D$)) – to get an insight into the performance and quality of the presented approach and to filter out the statistical staggering of different agile release planning problems.

The results of experiments on representative real-life data sets and representative generated data sets indicate that my approach can provide practical value as a decision support method for agile release planning.

The summary of the Contribution 1 can be found in the following box:

<table>
<thead>
<tr>
<th>C1: I have elaborated a novel method (namely Agile Release Scheduling (ARS)) for agile release planning to provide improved efficiency and effectiveness in release-centered decisions. [C6, E13, B2, G18, I26]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1.1 I have constructed a consistent ontology-styled information model of agile release scheduling (namely Agile Release Scheduling Model (ARSM)) to specify the components of the agile release scheduling decision space. This model involves the main concepts and their relations, and can be applied as a conceptual model of agile release planning tools. [C6]</td>
</tr>
<tr>
<td>C1.2 I have formulated agile release scheduling as an optimization problem (namely Agile Release Scheduling Problem (ARSP)) to provide features’ allocation considering team’s resource capacity and minimized number of iterations as planning objective. The main novelty of my approach lies in the mathematical precise formulation of the problem. The elaborated problem formulation is solved with an extension of the BINARY MULTIPLE KNAPSACK optimization model to cover wide-ranging release planning situations</td>
</tr>
</tbody>
</table>
3.5. Conclusions

with the expression of: 1) date-driven/scope-driven scheduling, 2) dependencies between features, 3) team capacities, 4) feature priorities, 5) staged-delivery and maximal deliverable value to the customer. [C6, E13, B2]

C1.3 I have developed a branch-and-bound binary multiple knapsack algorithm, which finds the global optimum (time complexity is $O(o \times n^2)$, where $n$ and $o$ is the number of features and iterations respectively) [C6, E13].
Chapter 4

Optimized Agile Iteration Scheduling

I

TERATION SCHEDULING aims at the selection and assignment of resources to development
tasks of an iteration within a release while several constraints – such as resource, temporal and
precedences – are fulfilled. Although agile software development represents a major approach
to software engineering there is no well-established conceptual definition and sound methodological
support of agile iteration scheduling. In this chapter I introduce my second contribution (C2) that is a
novel method (namely Agile Iteration Scheduling (AIS)) for agile iteration planning to provide improved
efficiency and effectiveness in iteration-centered decisions.

4.1 Synopsis

During the past decades, advanced planning and scheduling solutions emerged which received consider-
able attention [146, 178, 179, E9]. In spite of the attractive research results, only some of them are found
application in the software industry. In agile environments, which recommends small and iterative soft-
ware releases, the decision is even more difficult due to the perpetual changes in requirements, constraints
and objectives. However, the growing pressure to reduce costs, time to market and to improve quality
catalyzes transitions to more automated methods in software engineering to support project planning,
scheduling and decisions [127].

The following section draws up my novel concepts, models and algorithms for agile iteration planning.
In the chapter, I present 1) a conceptual model for agile iteration scheduling (C2.1). Then, I introduce 2)
a novel resource-constrained project scheduling optimization model (C2.2) with 3) a heuristic algorithm
for agile iteration scheduling (C2.3). The elaborated method significantly improves load balancing of
resources (≈ 4 − 5×), significantly accelerates iteration scheduling production (> 50%), enables more
than 10% increase in project execution’s efficiency, and more than 50% growth in planning efficiency
comparing to the traditional approaches.

A requirement – conforming to the widely cited IEEE 610.12 1990 standard [180] – is a collection of
needs arising from various users and/or customers of a project/product/organization – all of which must be
satisfied [127]. Ideally, requirements are independent of both design (how to realize needs) and goals (why
are to be expected) and only define needs (what must be met) [181]. Requirements management refers
4.1. Synopsis

to the whole life-cycle of requirements, including elicitation, modeling, analysis, change management, tracing, verification and validation [182].

In software engineering, diagrams (such as UML) are more frequently used for specifying, constructing, and documenting system artifacts. In UML-based projects, project planning should be based on requirements expressed by UML models; nevertheless it is created separately by manual work. Therefore, I elaborate 4) an extension of UML models by constructing a domain-specific metamodel for iteration scheduling (C2.4). This extension can be used as a consistent iteration scheduling repository that enables schedule generation with the previous results (C2.2-3).

The increasing strategic importance of IT demands that system requirements need to be integrated with managerial methods, processes and tools to support their alignment with business objectives. Goals are intentions (why aspect) that are expected to be fulfilled by the system. It enables modeling business and system goals, (non-functional) requirements, alternatives, and decision rationales. Therefore, I elaborate 5) a method that integrates goals and scenarios modeling techniques into iteration scheduling (C2.5) to support that the delivered system requirements will be mostly aligned with business objectives.

Case studies of 4) and 5) are drawn from the lending sector. It demonstrates how the method can significantly accelerate project scheduling production (> 50%), project realization (> 10%) and can improve planning quality through automated schedule production.

4.1.1 Related Work

Up to now, in the literature, there has not been emphasis on differentiating the release and the iteration planning processes. It can be explained by the fact that traditional methods do not separate these two levels of planning. Actually, the need of separation is derived from the iterative characteristic of agile approaches (Sec. 1.1). According to this characteristic, deliverables are broken down into small delivery phases (iterations) to encourage face-to-face communication, frequent inspection and adaption. As a consequence, the most of the previously cited papers (c.f. 3.1.1) can be treated as the related work of iteration planning.

4.1.2 Problem Statement and Analysis

Referring to the Agile Tools survey (Sec. 3.1), the lack of penetration of the modern agile planning tools can be explained by the weak embedded support of traditionally important project scheduling functions such as resource allocations and what-if analysis. In iteration planning point of view, the typical constraints and objectives are C2P1 precedences (to express temporal precedences between realizations), C2P2 balancing resource workloads (to avoid resources overloading), and C2P3 optimality (to choose the best one from different plans). Informal approaches work well in smaller projects, however as the size and complexity increases scheduling becomes a very complex process and advocates tool support [75, 76].

In UML-based projects, despite the planning and development should be based on a common collection of agreed requirements, the requirements are usually separated into two repositories: informal and semi-formal sources (e.g. documents) for management, and different requirements models (e.g. Use case diagram) for development. This separation is the root of many problems, such as C2P4 difficult iteration
planning (collecting planning data and producing Gantt charts etc.), C2P5 poor decision support (weak tool support etc.), and C2P6 difficult tracing of requirements realization (due to requirements changes etc.). These problems lead to imprecise planning and improper management of requirements, and all of them result in lower quality, increased efforts and additional project risks.

Requirements from various users and/or customers are often expressed as objectives or goals. Directly modeling goals, rather than entities, allow searching for alternatives before deciding a particular solution. Although planning and development should be based on common collection of requirements that aligned with the business goals, they are separated in different repositories: goal definitions in business specifications and requirements in the system requirements specifications. This separation may lead to solutions that are not aligned with the business goals and it results in C2P7 longer iteration time periods (because of implementation rework), C2P8 more costly realization of the required solution, and C2P9 poor planning support (realizing not the mostly demanded requirements). These problems lead to improper satisfaction of business needs and results in increased efforts and additional project risks.

4.1.3 Objectives

My elaborated method intends to provide a sound decision support to the C2P1-P3 by constructing 1) conceptual model to specify data semantics of agile planning (C2.1), 2) formulating iteration scheduling as a RESOURCE-CONSTRAINED PROJECT SCHEDULING optimization problem (see Sec. 2.5.3.4) that considers all the previous factors (C2.2), and 3) an innovative heuristic scheduling algorithm for wide-ranging agile iteration scheduling problems (C2.3). This method not only supports making delivery decisions even in complex situations, but it provides a 'quick and clean' solution for agile iteration scheduling.

I also elaborate 4) a generic extension of UML-based specification with planning objectives by constructing a domain-specific metamodel (C2.4) to diminish the negative effects of the C2P4-P6. Additionally, I elaborate 6) a generic extension of UML-based specification with business objectives to form a common, consistent repository for both development and management (C2.5) to eliminate the previous problems C2P7-P9. These extensions can be used as a consistent iteration scheduling repository that enables schedule generation applying the previous results (C2.2-3).

4.1.3.1 Outline

The rest of the chapter is arranged as follows: Sec. 4.2 details i) conceptual model, ii) optimization model and iii) algorithm for agile iteration scheduling; Sec. 4.3 presents iteration scheduling from Use case-based specifications; Sec. 4.4 shows goal-oriented extension of iteration scheduling; Sec. 4.5 discusses my solution and findings; and finally Sec. 4.6 concludes the chapter.

4.2 Decision Support in Iteration Scheduling

In this section first, I construct an information model of agile iteration planning. It represents concepts, relations, constraints to specify data semantics for agile iteration scheduling (C2.1) (Sec. 4.2.1). Then I point out that iteration scheduling can be characterized as a special kind of RESOURCE-CONSTRAINED
4.2. Decision Support in Iteration Scheduling

PROJECT SCHEDULING problem (RCPSp) (c.f. Sec. 2.5.3.4 and Sec. 4.2.3) (C2.2). After that, an innovative heuristic scheduling algorithm is introduced for wide-ranging agile iteration scheduling problems (Sec. 4.2.4) (C2.3). Finally, a prototypic tool, and the analysis of my elaborated solution is presented.

4.2.1 Conceptual Model of Agile Iteration Planning

First, I have to identify the main concepts of agile planning to formulate the iteration scheduling model. My elaborated conceptual model of agile release and iteration scheduling (ARISM) is presented in Sec. 3.2.3.

A compact view of the iteration planning conceptual model (Agile Iteration Scheduling Model (AISM)) is presented in the following. These concepts not only help to identify the objects and the subject of the optimization model but with the precise relationships it can also be used as database schema definition for an agile planning and scheduling application.

![Figure 4.1: Agile Iteration Scheduling Model (AISM).](image)

Generally, scheduling mandates defining who will realize what and when. Developer, Feature and Start date (S) of TechnicalTask concepts answer to these questions in agile iteration scheduling (cf. Sec.1.4.3). Additionally, schedule constraint must also be defined: resource constraints can be expressed by the number of developers (Sec. 3.2.1), temporal constraints can be asserted as attributes (e.g. deadlines – Sec. 3.2.1) and Precedence dependencies (Sec. 3.2.2.1).

In Figure 4.1, shaded objects (like Team and Coupling) pertain to release scheduling only. Objects in italics point out abstract objects (cannot be instantiated). The lower compartments of entities (e.g. TechnicalTask) give the list of attributes of each object.
4.2. Decision Support in Iteration Scheduling

4.2.2 Mapping to RCPSp

The optimized version of the iteration scheduling problem can be derived by selecting the extreme-valued schedule from the potentially feasible alternatives. In the following, an analogy between iteration planning and RESOURCE-CONSTRAINED PROJECT SCHEDULING problem (RCPSp) (see Sec. 2.5.3.4) is presented [146]. Generally, scheduling concerns with the allocation of limited resources (manpower) to tasks over time in order to fulfill the predefined scheduling objective. In fact, many different objectives are possible – depending on the goals of the decision makers – but my aim is to ‘maximize users’ and/or customers’ satisfaction in the least time possible’. Thus, the makespan minimization (i.e. finding the minimum execution time) is the most adequate. As agile methods recommend collaborative teamwork – without any development role (such as analyst, programmer, tester) – I only identify one kind of resource: the developer. The complexity of scheduling arises from the interaction between tasks by implicit and explicit dependencies. While the previous is given by scarcity of resources, the latter is emerged from different precedences (Chapter 2 Sec. 2.5.3.2) between tasks that define the routing of the tasks [90].

I extended the ordinary RCPSp problem with i) pre-assignments (i.e. assigning certain tasks to resources before scheduling) and ii) timeboxed iteration duration control to provide suitable scheduling method for wide-ranging iteration scheduling situations. On the one hand, defect corrections and onward development of a formerly delivered functionalities legitimates pre-assignments. On the other hand, timeboxed iteration execution mandates an upper boundary control in time – which is not allowed to be exceeded otherwise schedule is treated infeasible.

4.2.3 Formulating AISP Optimization Model

Let \( R \) be the set of resources \( i \) and the following typical properties for scheduling be interpreted on technical tasks to schedule them (i.e. \( j \in A \)) [146]:

**Effort** : \( w_j \) – time estimation (in hours) is associated with each task. It is calculated by simple expert estimation (e.g. 2,4, or 8 working hour (Wh)).

**Pre-assignment** : \( a_j \) – in some cases resource pre-assignment is applied before scheduling. It is used by the scheduler algorithm during resource allocation.

Let the vector \( S = (S_0, S_1, \ldots, S_{n+1}) \) be start times for tasks’ realizations – where \( S_j \geq 0 : j \in A \) and \( S_0 = 0 \). The vector \( S \) is called a schedule of development. In this definition the 0 and \( n + 1 \) are auxiliary elements to represent iteration beginning and termination, respectively.

4.2.3.1 Temporal and Resource Constraints

Dependencies can be defined by precedence relations (Eq. 4.1):

\[
S_j - S_{j'} + d_{j'} \geq P_{j',j} : j', j \in A
\]  

(4.1)

Please note, this relation is equivalent to the Precedence relation \( (P) \) that was introduced in Sec. 4.2.1).
4.2. Decision Support in Iteration Scheduling

Let the \( R_i \in \mathbb{N} \) be a set of capacities of resources that have been assigned to the project. The effort estimation yields resource requirements \( r_{j,i} \in \mathbb{Z} \) for each task \( j \) and each resource \( i \). Now let \( S \) be some schedule and let \( t \) be some point in time. Then let \( A^* (S, t) \triangleq \{ j \in A | S_j \leq t \leq S_j + w_j \} \) be the active set of tasks being in progress at time \( t \). The corresponding requirement for resource \( i \in R \) at time \( t \) is given by \( r_i (S, t) \triangleq \sum_{j \in A^* (S, t)} r_{j,i} \). As a consequence, the resource constraints can be treated as follows (Eq. 4.2):

\[
r_i (S, t) \leq R_i \quad : i \in R
\]

### 4.2.3.2 Optimization Model

With the application of previous elements, RCPSP (Sec. 2.5.4, 2.5.3.4) for iteration scheduling can be formulated as follows:

Minimize \( z = S_{n+1} \) \hspace{1cm} (4.3a)

subject to

\[
S_j - S_{j'} + P_{j,j'} \geq 0 \quad : j, j' \in A \quad (4.3b)
\]

\[
r_i (S, t) \leq R_i \quad : i \in R \quad (4.3c)
\]

\[
S_{n+1} \leq l^I \quad (4.3d)
\]

where Eq. 4.3b, 4.3c are scheduling constraints (c.f. Eq. 4.1, 4.2), Eq. 4.3d is the timebox duration (length of the iteration), and Eq. 4.3a is the makespan minimization objective.

### 4.2.4 Solving the AISP Optimization Problem

For the previous optimization model, I developed an innovative scheduling algorithm. It is a constructive heuristic algorithm, which iteratively selects and assigns technical tasks to resources. In the program listing (Algorithm 5) lowercase/uppercase letters with indices denote vectors/matrices (e.g. \( r_i, P_{j,j'} \)). While bold-faced types show concise (without indices) forms (e.g. \( P \)).

In the require section the preconditions are given. The vector \( r \) indicates the available resources (developers) in the iteration. Each \( w_j \) is the planned effort (duration) for technical task \( j \) – both development and defect correction. Every element of vector \( a_j \) contains a reference to a resource index (\( a_j \in \{1..|r|\} \)) which indicates resource pre-assignment to task \( j \). The \( a_j = 0 \) means that task \( j \) is not pre-assigned, thus the algorithm will find the best resource to its realization. Precedences between tasks (c.f. Eq. 4.3b) can be represented by a precedence matrix where \( P_{j,j'} = 1 \) means that task \( j \) precedes task \( j' \), otherwise \( P_{j,j'} = 0 \). Both conditions \( P_{j,j} = 0 \) (no loop) and \( P \) is directed acyclic graph (DAG) ensures that temporal constraints are not trivially unsatisfiable. Iteration timebox is asserted by variable \( l^I \). It is used as an upperbound in resource allocation to prevent resources overloading. The result of the algorithm is a schedule matrix \( S \), where rows represent resources, and columns give an order of task execution. Thus \( S_{i,p} = j \) means that task \( j \) is assigned to resource \( i \) at position \( p \). The ensure section prescribes the postcondition on the return value (\( S \)): every task \( j \) has to be assigned to exactly one resource \( i \). From now
4.2. Decision Support in Iteration Scheduling

Algorithm 5 Icap algorithm with AF strategy

Require:
1: \( r_j \in \mathbb{N}, l^I \in \mathbb{N} \) /* resources and length of the iteration */
2: \( a_j \in \mathbb{N} : a_j \in \{1..|\mathbf{r}|\}, w_j \in \mathbb{R} \) /* pre-assignments and duration of tasks */
3: \( P_{j,j'} \in 0, 1 \wedge P_{j,j'} = 0 \wedge P \text{ is DAG} \) /* precedences */

Ensure:
5: \( S_{i,j} \in 0, 1 \wedge \forall j \exists \forall S_{i,j} = 1 \)

1: \( m \leftarrow \text{length}(\mathbf{r}), n \leftarrow \text{length}(\mathbf{d}) \) /* number of resources and tasks */
2: \( \mathbf{S} \leftarrow [0]_{m,n} \) /* assignment matrix initialization */
3: \( \text{rlist} \leftarrow \emptyset \), \( \text{slist} \leftarrow \emptyset \) /* ‘ready list’ and ‘scheduled list’ initialization */
4: for \( j \leftarrow 0 \) to \( n \) do
5: \( \text{pot} \leftarrow \text{findNotPrecedentedTasks} (\mathbf{P}) \) /* find potentially tasks */
6: \( \text{rlist} \leftarrow \text{pot} \setminus \text{slist} \) /* construct ready list */
7: if \( \text{rlist} = \emptyset \) then
8: \( \text{return} \emptyset \) /* No schedulable task */
9: end if
10: \( \text{return} \emptyset \) /* overloaded iteration */
11: \( l \leftarrow \sum (S_{i,1..n}) \) /* calculate load of resource \( i \) */
12: if \( (l + w_j) > l^I \) then
13: \( \text{return} \emptyset \) /* index for next task */
14: end if
15: \( \text{p} \leftarrow \text{findNextPos} (\mathbf{S}, i) \) /* assign task \( j \) to resource \( i \) at position \( p \) */
16: \( S_{i,p} \leftarrow j \) /* add task \( j \) to ‘scheduled list’ */
17: \( \text{slist} \leftarrow \text{slist} \cup \{ j \} \) /* delete precedence related to scheduled task */
18: \( P_{1..n,j} = 0 \)
19: end for
20: \( \text{return} \mathbf{S} \)

on, I name the function – that the algorithm realizes – \( \mathcal{S}_{\text{AF}} : (\mathbf{A}, \mathbf{R}, \mathbf{a}, \mathbf{w}, \mathbf{P}, l^I) \to \mathbf{S} \) as agile iteration scheduling function.

During scheduling steps, first the initial values are set (line 1 – 3). The iteration value \( n \) is equal to the number of technical tasks (line 1). The algorithm uses a ready list (rlist) and a scheduled list (slist) to keep track of schedulable and scheduled tasks. Potentially schedulable tasks (pot) are unscheduled tasks from which the algorithm can choose in the current control step without violating any precedence constraint (line 5). Previously assigned tasks are subtracted from pot to form the ready list (line 6). As long as the ready list contains schedulable tasks, the algorithm chooses tasks from it – otherwise the schedule is infeasible (line 7) and as a consequence the algorithm aborts (line 8).

To select the next task to schedule from concurrently schedulable tasks (i.e. ready list) I constructed the custom ‘Assigned First’ scheduling rule (line 10) (c.f. Sec. 4.2.2). This rule chooses from the preassigned tasks \( (a_j > 0) \) before the unassigned ones \( (a_j = 0) \). As the selection sequence is discretionary I applied the max function to the choice. After selection the minimal loaded (min summa duration) resource is allocated to the selected task unless the task is pre-assigned to a given resource (line 11 – 15) (c.f. Sec. 4.2.2). If the load of the resource \( i \) exceeds iteration timebox \( (l^I) \) then the schedule is treated infeasible (line 16 – 19).

104
4.2. Decision Support in Iteration Scheduling

The following step is to find the index of next task position \((p)\) (right after the previous task’s index) at resource \(i\) (line 20) for task \(j\) for assignment (line 21). Finally, scheduled list (slist), is updated with scheduled task (lines 22), and no longer valid precedence relations are also deleted from \(P\) (lines 23).

Iteration proceeds until all items are assigned to iterations (line 4–24). After termination, \(S\) contains the task assignments to resources and the makespan is \(z \Leftarrow \max_{i=1..m} \sum_{p=1}^{n} d_{S_i,p} - \text{c.f. Eq. 4.3a)}\).

4.2.4.1 Solution Analysis

This greedy strategy makes a series of local decisions, selecting at each point the best step without backtracking or lookahead. Thus local decisions miss the global optimal solution, but produce quick (time complexity is clearly \(O(n + m)\)) and usually sufficient results for practical applications.

Figure 4.2 illustrates the application of the algorithm on real application development data which was extracted from the backlog of EHKR [165]. The figure shows post-mortem scheduling result of an iteration – visualized by resource aspect Gantt diagram – where tasks’ realizations plotted against time. The diagram points out that 94 tasks (with 2, 4, and 8 working hours \((Wh)\)) are allocated to 6 resources, and the makespan is 78.

\[
\text{FIGURE 4.2: Generated Iteration Schedule.}
\]

4.2.5 Tool Support

I extended the SERPA\textsuperscript{TM} web application with conceptual model of iteration planning (c.f. C2.1) to promote experimental validation. Additionally, I also implemented an iteration planning algorithm (c.f. C2.2-3) – named lscap – as a part of my PROPAS\textsuperscript{TM} Matlab toolbox. These tools were used during the simulations. The implementation of lcap can be found in Appendix C.3.

4.2.6 Experimental Results

I carried out a post mortem analysis on four real-life representative data sets extracted from a software development company [165] to evaluate the C2.1, C2.2 and C2.3 parts of my contribution. Applying the historical iteration planning data, as an input for the scheduling algorithm, made it possible to compare them [167]. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical in terms of organization size (6–8 developers), applied agile methods (Scrum-like development process) and techniques (XP development practices). At this organization, the iteration scheduling process is made up of the typical agile planning steps (see Sec.1.4.3).
4.2. Decision Support in Iteration Scheduling

In this section, first I set research questions, then present necessary background information, and finally I present and interpret my findings.

4.2.6.1 Research Questions

My initial intend (see Sec. 4.1.2 C2P1-3) was to support decisions in agile iteration scheduling in the following aspects: 1) dealing with precedences, 2) tracking workloads, and 3) providing optimal (makespan minimized) delivery plan. To validate my elaborated method the next questions were addressed: How does optimization-based iteration scheduling compare with informal one in terms of C2Q1) resource workload over time, C2Q2) quality and C2Q3) feasibility of the plans.

4.2.6.2 Context and Methodology

EHKR is a client risk management system (approx. 2 million SLOC) for credit institutions for analyzing the non-payment risk of clients. It has been continual evolution since its first release in the middle of 90s. The system was written in Visual Basic and C# the applied methodology was a custom agile process.

The planning process were made up of the following steps. First, during release planning, the requirements were selected (expressed in User stories [67]) from the backlog – considering users’ and/or customers’ demands. Then every User story was estimated by the team and distributed into iterations taking resources, precedences and iteration timebox into account. Second, during iteration planning, each User Story was broken down into technical tasks and important defect corrections were also selected to the next product increment. Finally, resource allocation was determined intuitively by the team in intuitive way and the conflicts (precedences, resource overload) were managed during daily meetings (see Figure 1.4).

4.2.6.3 Data Collection and Results

Four data sets (four iterations (I₁A, I₂A, I₁B, I₂B) of two releases (R₁, R₂)) were selected to make a comparison between the algorithmic and the intuitive method. These data sets were results of typical agile processes and as a consequence they were appropriate for analysis. All iterations had same project members (6 developers /Dev./), iteration length (80 working hours (2 weeks) /IL/), domain, customer, and development methodology, but they were characterized by different number of technical tasks (development /DT/ and defect correction /CT/ with 2, 4 and 8Wh), User Stories /US/, precedences /Prec./, and pre-assignments /Ass./. Table 4.1 summarizes state variables that were used to capture facts that were likely affect the findings. These variables were collected from the SharePoint-based backlog.

<table>
<thead>
<tr>
<th>TABLE 4.1: Iteration Planning Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev.</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>I₁A</td>
</tr>
<tr>
<td>I₂A</td>
</tr>
<tr>
<td>I₁B</td>
</tr>
<tr>
<td>I₂B</td>
</tr>
</tbody>
</table>
I constructed Task effort ($T_{E_i}$) response variables to test C2Q1. This simple variable is computed by adding up estimated tasks’ efforts that were assigned to resources $i$. Explanations of C2Q2 and C2Q3 were produced with the utilization of the solution’s inherent properties.

### 4.2.6.4 Analysis

To answer the questions C2Q1-3 simulations were performed on the previously described input data to compare the characteristics of the two approaches. The simulation output is summarized in Table 4.2,4.3.

On the left the four historical iteration schedules are presented ($I_A^1$, $I_A^2$, $I_B^1$, and $I_B^2$). In the table $D_i$ denote resources (developers); 2, 4, and 8 values are estimated effort (instead of indices) of task realizations; and finally the previously introduced response variable ($T_{E_i}$) can be seen. On the right column simulation results ($^*I_A^1$, $^*I_A^2$, $^*I_B^1$, and $^*I_B^2$) are presented.

**TABLE 4.2: Intuitive Schedules**

<table>
<thead>
<tr>
<th>$I_A^1$</th>
<th>$T_{E_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>2 2 2 2 2 2 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 - - - 84</td>
</tr>
<tr>
<td>$D_2$</td>
<td>2 2 2 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 8 8 - - - 84</td>
</tr>
<tr>
<td>$D_3$</td>
<td>2 2 2 2 2 2 2 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 - - - 46</td>
</tr>
<tr>
<td>$D_4$</td>
<td>2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 70</td>
</tr>
<tr>
<td>$D_5$</td>
<td>2 2 2 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 - - - 102</td>
</tr>
<tr>
<td>$D_6$</td>
<td>2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 - - - 64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_A^2$</th>
<th>450</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 64</td>
</tr>
<tr>
<td>$D_2$</td>
<td>2 2 2 2 2 2 2 2 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 66</td>
</tr>
<tr>
<td>$D_3$</td>
<td>2 2 2 2 2 2 2 2 4 4 4 4 4 8 8 8 8 8 8 8 8 - - - 52</td>
</tr>
<tr>
<td>$D_4$</td>
<td>2 2 2 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 8 - - - 90</td>
</tr>
<tr>
<td>$D_5$</td>
<td>2 2 2 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 8 - - - 90</td>
</tr>
<tr>
<td>$D_6$</td>
<td>2 2 2 2 2 2 2 2 4 4 4 4 4 4 4 8 8 8 8 8 8 - - - 64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_B^1$</th>
<th>426</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>2 2 2 2 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 66</td>
</tr>
<tr>
<td>$D_2$</td>
<td>2 2 2 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 64</td>
</tr>
<tr>
<td>$D_3$</td>
<td>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 68</td>
</tr>
<tr>
<td>$D_4$</td>
<td>4 2 2 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 - - - 82</td>
</tr>
<tr>
<td>$D_5$</td>
<td>2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - 84</td>
</tr>
<tr>
<td>$D_6$</td>
<td>2 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 - - - 76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$I_B^2$</th>
<th>440</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 - - - - - - - - - - - - - 44</td>
</tr>
<tr>
<td>$D_2$</td>
<td>2 2 2 2 2 2 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 - - - - - - - - 74</td>
</tr>
<tr>
<td>$D_3$</td>
<td>2 2 2 4 4 4 8 8 8 8 8 8 8 8 8 - - - - - - - - - - - - - 78</td>
</tr>
<tr>
<td>$D_4$</td>
<td>4 4 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 - - - - - - - - 88</td>
</tr>
<tr>
<td>$D_5$</td>
<td>2 2 2 2 2 2 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 - - - - - - - - 82</td>
</tr>
<tr>
<td>$D_6$</td>
<td>2 2 2 2 4 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 8 8 8 - - - - - - - - 88</td>
</tr>
</tbody>
</table>

| $I_B^3$ | 454 |

To compare the intuitive and the algorithmic cases quantitative (statistical) analysis were performed on the two response variables ($T_{E_i}$ and $^*T_{E_i}$). The result is presented in Table 4.4 and summarized in boxplot (see Figure 4.3).

From these, I conclude that optimized case i) did not exceed the timebox limit ($^*Max = 78Wh < 80Wh < Max = 102$) which means lower level scheduling risk; ii) has less dispersion in total task allocation ($^*Std.dev = 3$ vs. $Std.dev = 14.6$); iii) yields more balanced workload on resources – while the means are similar ($^*Mean = 74 \approx Mean = 75$). As a consequence, in terms of coefficient variation (i.e. normalized measure of dispersion), the optimization-based scheduling provides $c_v/^*c_v =
4.2. Decision Support in Iteration Scheduling

<table>
<thead>
<tr>
<th>TABLE 4.3: Optimized Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I^A_1$</td>
</tr>
<tr>
<td>$D_1$</td>
</tr>
<tr>
<td>$D_2$</td>
</tr>
<tr>
<td>$D_3$</td>
</tr>
<tr>
<td>$D_4$</td>
</tr>
<tr>
<td>$D_5$</td>
</tr>
<tr>
<td>$D_6$</td>
</tr>
</tbody>
</table>

$I^A_2$ | $I^B_2$
| $D_1$ | 2 4 4 4 8 8 8 2 8 4 4 8 8 4 | - | - | - | - | 72 |
| $D_2$ | 2 4 4 4 4 8 8 2 2 4 8 4 2 2 4 8 | - | - | - | - | 74 |
| $D_3$ | 8 8 2 2 4 2 2 8 4 4 8 8 4 | - | - | - | - | 66 |
| $D_4$ | 4 8 8 8 2 2 4 2 8 8 8 - | - | - | - | - | 72 |
| $D_5$ | 8 8 8 2 2 2 2 8 4 4 4 2 2 2 4 2 2 8 | - | - | - | - | 70 |
| $D_6$ | 4 8 2 2 2 4 8 2 2 2 4 2 4 8 8 4 2 8 | - | - | - | - | 72 |

$I^A_3$ | $I^B_3$
| $D_1$ | 4 4 4 4 8 8 8 4 2 4 8 4 8 4 | - | - | - | - | 72 |
| $D_2$ | 2 4 4 4 2 4 8 2 2 4 4 2 2 4 8 | - | - | - | - | 76 |
| $D_3$ | 4 4 4 4 8 8 4 4 8 4 8 2 4 8 - | - | - | - | - | 76 |
| $D_4$ | 4 4 8 8 8 4 8 8 2 2 4 8 - | - | - | - | - | 76 |
| $D_5$ | 4 4 8 2 4 8 8 4 2 4 8 2 4 4 - | - | - | - | - | 70 |
| $D_6$ | 2 4 4 4 8 2 8 4 2 4 8 4 4 - | - | - | - | - | 70 |

$I^A_4$ | $I^B_4$
| $D_1$ | 4 4 4 8 2 2 4 4 4 | - | - | - | - | 72 |
| $D_2$ | 4 4 4 4 2 8 2 4 4 8 4 4 2 8 4 8 - | - | - | - | - | 78 |
| $D_3$ | 8 8 8 8 4 8 8 8 - | - | - | - | - | 76 |
| $D_4$ | 4 4 8 8 2 8 2 8 4 8 8 8 - | - | - | - | - | 76 |
| $D_5$ | 2 2 4 4 4 2 8 2 8 4 2 2 2 4 8 2 4 8 | - | - | - | - | 76 |
| $D_6$ | 2 4 8 8 8 4 8 8 4 8 2 4 8 - | - | - | - | - | 76 |

* Table 4.4: Comparison of Schedules

<table>
<thead>
<tr>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>Std.dev.</th>
<th>$C_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I^A_1$</td>
<td>74</td>
<td>75</td>
<td>44</td>
<td>102</td>
<td>14.6</td>
</tr>
<tr>
<td>$I^B_1$</td>
<td>74</td>
<td>74</td>
<td>66</td>
<td>78</td>
<td>3.0</td>
</tr>
</tbody>
</table>

$$\frac{\text{Std.dev}}{\text{Mean}} / \frac{\text{Std.dev}}{\text{Mean}} = \frac{0.1976}{0.0410} \approx 5 \times$$ more balanced resource workload over time contrary to the intuitive method (c.f. C2Q1).

**Figure 4.3**: Boxplots of Intuitive (above) and Optimized (below).

The algorithmic method easily resolves complex decision situation – as it handles precedences between tasks and avoids resource workloads – contrary to the intuitive case where these are managed intuitively during daily meeting. As a consequence these two capabilities of the algorithmic method ensure higher quality and lower-risk feasible plans in contrast to the intuitive case (c.f. C2Q2-3).
4.2. Decision Support in Iteration Scheduling

4.2.7 Computational Benchmarking

Problem solving time of the historical data set, which can be considered as medium-sized problem, took less than one second with my prototypic tool. Therefore, to give some orientation about the performance and the quality of the representative generated numerous plans (480 different cases) of my approach on larger problems, I carried out simulations with the guidance of [139].

To get a more nuanced picture of the algorithm I considered four groups of randomly generated instances of iteration scheduling to reflect special properties that influence the solution process. The four groups were made up of \{150, 300, 500, 1000\} number of technical tasks. The 1000 number of technical tasks have to be considered as an extreme case that hardly exists even in the greatest agile teams. In all instances the technical tasks (\(w_j\)) were uniformly distributed in the data set of \{2, 4, 8, 12, 16\} expressed in working hour (\(Wh\)) (reflected the typical technical task duration including development tasks /DT/ and correction tasks/CT/ [67]). To enlarge the problem size, I chose four different team sizes with \{5, 10, 15, 20\} people – the first three teams’ size were considered typical [67] but the fourth one is extremely large in an agile setting. I also generated precedences between the technical tasks to increase the complexity of the scheduling problem. The generated precedences were produced proportionally to the technical tasks in terms of percentage of \{5\% , 10\%, 20\%, 40\%\}. The 5\% is considered as an easy problem while the 40\% is an extremely complex one in agile planning point of view.

4.2.7.1 Solving Time

The algorithms were run 30 times to calculate the mean solving time in the four team settings (5 – 20 developers) and four precedence densities (5\% – 40\%). All tests were run on an Intel Pentium 4, 2.2 GHz, 4GB memory, MS Windows 7. The results are presented in Table 4.5, where \(ST\) denotes the algorithm’s solving time (in seconds).

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Prec.</th>
<th>Team size</th>
<th>(ST)</th>
<th>(\nabla \text{Prec}_i) | (ST)</th>
<th>(\nabla \text{Team}_i) | (ST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>5%</td>
<td>5</td>
<td>0.11</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>10</td>
<td>0.29</td>
<td>0.01</td>
<td>1.70</td>
</tr>
<tr>
<td>500</td>
<td>15</td>
<td>0.83</td>
<td>–</td>
<td>6.68</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>2.51</td>
<td>–</td>
<td>22.25</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>10%</td>
<td>5</td>
<td>0.12</td>
<td>0.10</td>
<td>–</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>0.29</td>
<td>0.01</td>
<td>1.48</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>15</td>
<td>0.84</td>
<td>0.01</td>
<td>6.07</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>2.66</td>
<td>0.06</td>
<td>21.49</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>20%</td>
<td>5</td>
<td>0.13</td>
<td>0.08</td>
<td>–</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>0.29</td>
<td>0.00</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>15</td>
<td>0.87</td>
<td>0.04</td>
<td>5.85</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>2.87</td>
<td>0.08</td>
<td>21.57</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>40%</td>
<td>5</td>
<td>0.13</td>
<td>0.04</td>
<td>–</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>0.30</td>
<td>0.01</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>15</td>
<td>0.90</td>
<td>0.03</td>
<td>5.76</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>20</td>
<td>2.88</td>
<td>0.00</td>
<td>20.73</td>
<td></td>
</tr>
</tbody>
</table>

Looking at the data within the \(ST\) column we can realize two things: 1) the densities of precedence has a little affect while 2) the team size has a great affect on the solving time. In order to measure their effects I constructed two variables. The first variable measures the effects of the number of precedences using the same team sizes. This variable is formulated as \(\nabla \text{Prec}_{i+1,i} \triangleq (ST_{\text{Prec}_i} + ST_{\text{Prec}_{i+1}})/ST_{\text{Prec}_i} – \) where the \(i\) denotes the index of precedence density which value spreads from 1 to 3 to indicate values.
4.2. Decision Support in Iteration Scheduling

of \{5\%, 10\%, 20\%, 40\%\} densities. Considering the values of the \(\nabla_{ST}^{Prec}\) variable in the Table 4.5, one can realize that the variable does not show any tendency and its values are between 0 and 0.10 (small values). As a consequence the effect of precedence had a value of \(\nabla_{ST}^{Prec} = 0.037\) – which means 3.7\% – on average. This can be explained by the fact that the precedence constraint is fulfilled with rearrangement of tasks that requires linear time complexity function (see \textit{findNotPrecededTask} function in Algorithm 5).

The goal of the second variable was to measure the effects of team size using the same precedence density. This variable was constructed as \(\nabla^{Team_{1,1}}_{ST} \triangleq (ST_{Team_{1}} + ST_{Team_{j}})/ST_{Team_{1}}\) – where \(j\) denotes the index of team size which value spreads from 1 to 4 to indicate values of \{5, 10, 15, 20\} team members. In the \(\nabla^{Team}_{ST}\) column of the Table 4.5, the changing of values shows definite tendencies: 1) \(\nabla^{Team_{1,1}}_{ST} = \text{avg}(1.70, 1.48, 1.30, 1.23) = 1.43\); 2) \(\nabla^{Team_{3,1}}_{ST} = \text{avg}(6.68, 6.07, 5.85, 5.76) = 6.09\); 3) \(\nabla^{Team_{4,1}}_{ST} = \text{avg}(22.25, 21.49, 21.57, 20.73) = 21.51\). These numbers reflect the before mentioned worst case \(O(n + m)\) – where \(n\) denotes the number of tasks and the \(m\) denotes the number of developers – time complexity of the algorithm. In the case of Team2,1 the change in developers was 300/150 = 2 and tasks was 10/5 = 2 that resulted in theoretical 4\times greater solving time. It the in case of Team3,1 the solving time required 15/5 * 500/150 = 10\times more time. Finally, in the in case of Team4,1 it required 20/5 * 1000/150 = 27\times more solving time.

Although there is \(O(n + m)\) relationship between the task numbers, developer numbers and the solving time, delivering 1000 technical tasks in an iteration with 20 team member – which practically never occurs in agile teams – requires cca. 3 seconds that is acceptable response time. Additionally, I have to underline the fact that I implemented my algorithms on the Matlab platform, which executes the code in an interpreted way. Therefore, compiled versions (e.g. implemented in C) are expected to be faster with one order of magnitude at least.

4.2.7.2 Quality of Plans

The previous randomly generated instances of iteration data were analyzed to measure the quality of the iteration plans. The algorithms were run 30 times to calculate the average. I investigated two cases: i) the \textit{naive} case and the ii) \textit{alg} case – which applied the before mentioned algorithmic approach.

In the \textit{naive} case I used a \textit{simple scheduling algorithm} that mimics the manual scheduling method which can be characterized as follows. Firstly, it considered the capacity of developers during task allocation: if the allocated task fills the developer capacity up then no more other task is allocated to him or her. In other words, the optimality of resource allocation is not considered – as in the practice. Secondly, it used the pre-assignment of tasks to developers functionality also: some tasks are allocated to given people as they have the skills or experience to solve those problems. These allocations were carried out using random task assignments to developers in this \textit{naive} approach. Finally, it did not consider the precedences between technical tasks since they are usually resolved during daily meetings in practice. The \textit{naive} and the \textit{alg} settings made it possible to measure the quality of my approach.

The previously detailed factors (number of technical tasks, precedences and developers) of scheduling (see Sec. 4.2.7.1) did not affect on the plan quality but the solving time (see Sec. 4.2.7.1). Therefore, the statistics of the different 480 cases are presented in one aggregated statistics. To compare the \textit{naive} and the \textit{alg} cases quantitative (statistical) analysis were performed on the response variables \(TE_{naive}\) and
4.3 Iteration Scheduling from Use Case-based Specification

The statistics of the simulation is presented in Table 4.6. The first row shows the result of naive scheduling and the second row contains the result of the my approach.

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>Std.dev.</th>
<th>(c_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TE_{\text{naive}})</td>
<td>76.18</td>
<td>79</td>
<td>71.01</td>
<td>88.25</td>
<td>4.26</td>
<td>0.0559</td>
</tr>
<tr>
<td>(TE_{\text{alg}})</td>
<td>78.23</td>
<td>78</td>
<td>74.45</td>
<td>79.24</td>
<td>1.15</td>
<td>0.0147</td>
</tr>
</tbody>
</table>

Comparing the naive approach with the previous historical manual approach (cf. Sec. 4.2.6.4), the naive one had considerably better scheduling value in all aspects. This result can be explained by the complexity of the iteration scheduling problem: even in the smallest problem (5 developers, 80-100 tasks, and precedences) the combination of possible schedules are large, therefore, to construct a ‘good enough’ plan from the possible planning values tool support is needed. Moreover, when the problem size increases – in terms of either developers, tasks or precedences – the tool support is even more important.

Comparing the naive and the alg cases, from the statistics of Table 4.6, I conclude that that the \(\text{alg}\) case i) did not exceed the timebox limit \(\text{Max}(TE_{\text{alg}}) = 79.24\text{Wh} < 80\text{Wh}\) vs. \(\text{Max}(TE_{\text{naive}}) = 88.25\text{Wh} > 80\text{Wh}\) which means even lower level scheduling risk; ii) has even less dispersion in total task allocation \(\text{Std.dev}(TE_{\text{alg}}) = 4.26\); iii) yields even more balanced workload on resources – while the means are similar \(\text{Mean}(TE_{\text{alg}}) = 78 \approx \text{Mean}(TE_{\text{naive}}) = 79\). As a consequence, in terms of coefficient variation (i.e. normalized measure of dispersion), the optimization-based \(\text{alg}\) scheduling case provides \(\frac{c_v(TE_{\text{alg}})}{c_v(TE_{\text{naive}})} = \frac{\text{Std.dev}(TE_{\text{alg}})}{\text{Mean}(TE_{\text{alig}})} / \frac{\text{Std.dev}(TE_{\text{alig}})}{\text{Mean}(TE_{\text{alg}})} = \frac{0.0559}{0.0147} \approx 3.8\times\) more balanced resource workload over time contrary to the naive approach (c.f. C2Q1).

The my algorithmic method easily resolves complex decision situation – as it handles precedences between tasks and avoids resource workloads – contrary to the naive or intuitive cases. As a consequence these two capabilities of my algorithmic method ensure higher quality and lower-risk feasible plans in contrast to the naive or intuitive cases (c.f. C2Q2-3).

4.3 Iteration Scheduling from Use Case-based Specification

In this section I introduce extension elements of Use case diagrams to formulate the previously detailed AIP (see Sec. 4.2.3) in Use Case-based specification (C2.4). Finally, a prototypic tool and the analysis of my elaborated solution is presented.

4.3.1 Mapping Use Case-based Specification to AISP Optimization Model

In the following, the extension of Use case-based specification is presented. Along with the resource constraint (see Sec. 4.2.3.1) the AISP problem (Sec. 4.2.3) can be formulated to easily generate iteration schedule.
4.3. Iteration Scheduling from Use Case-based Specification

4.3.1 Scheduling Properties of Use Cases and Actors

In my approach, requirements are expressed with use cases and actors. Similarly to the previously mentioned case in Sec. 4.2.3 – let \( R \) be the set of resources \( i \) and the following typical properties for scheduling be interpreted on both use cases and actor to schedule them (i.e. \( j \in A \)) [146]:

**Effort** : \( w_j \) – time estimation (in hours) is associated with requirement. It is calculated by some method (see 4.3.1.2), or it is a simple expert estimation.

**Pre-assignment** : \( a_j \) – in some cases resource pre-assignment is applied before scheduling. It is used by the scheduler algorithm during resource allocation.

Let the vector \( S = (S_0, S_1, ..., S_{n+1}) \) be start times for tasks’ realizations – where \( S_j \geq 0 : j \in A \) and \( S_0 = 0 \). The vector \( S \) is called a schedule of development. In this definition the 0 and \( n+1 \) are auxiliary elements to represent iteration beginning and termination, respectively.

4.3.1.2 Duration Calculation from Use Case Diagrams

As the relation between \( UCP \) and \( TPH \) is linear (see Eq. 1.7), the development duration (effort) calculation of each use case and actor can be counted with proportioning: 1) \( d_{actor_j} \propto AW_j \), and 2) \( d_{use\ case_j} \propto UCW_j \). So the following Eq. 4.4 holds (\( W_j = \{AW_j, UCW_j\} \)):

\[
w_j = W_j \ast TCF \ast EF \ast MR \tag{4.4}
\]

Therefore estimated duration for each use case/actor can be calculated from the given weight (\( W_j \)) multiplied by a project constant (\( = TCF \ast EF \ast MR \)).

In [96] it is demonstrated that linear relationship between UCP (size) and TPH (effort) estimation is a good approximation within the interval \([3, 30]\) person-months projects.

4.3.1.3 Precedence Constraints from Use Case Diagrams

The source of dependencies are defined in Chapter 2 Sec. 2.5.3.2. In iteration scheduling point of view, the precedence relation (\( P \)) is relevant (see Sec. 4.2.3.1 in Eq. 4.1).

The precedence relation (\( P \)) includes the following type of dependencies between requirements (Chapter 2 Sec. 2.5.3.2):

<table>
<thead>
<tr>
<th>Dependency name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRES ( j )</td>
<td>( j ) builds on functionality ( j' ) to function</td>
</tr>
<tr>
<td>CVALUE ( j )</td>
<td>( j' ) influences the customer value of ( j ), so rational to realize ( j' ) earlier</td>
</tr>
<tr>
<td>ICOST ( j )</td>
<td>( j' ) influences the implementation cost of ( j ), so rational to realize ( j' ) earlier</td>
</tr>
<tr>
<td>TEMPORAL ( j )</td>
<td>technological/organizational constraints between ( j ) and ( j' ), so it is rational to realize ( j' ) earlier</td>
</tr>
</tbody>
</table>

In Use case-based requirements specification, the Extend and Include relations [79] have the following definition:
4.3. Iteration Scheduling from Use Case-based Specification

- Extend means that base use case precedes extending use case realization.
- Include indicates that included use case precedes base use case realization.

As a consequence, the Extend and the Include relations can be interpreted as REQUIRES dependencies. Although, in order to cover the remained dependency types (CVALUE, ICOST TEMPORAL), we have to introduce a new kind of UML relation between use cases and actors. I name this new relation as precedes relation. As an illustration, the model in Figure 4.4 is evolved from the previous CCM example.

![Figure 4.4: Precedence Relations of Credit Controller Example.](image)

Figure 4.4 denotes that the communication interface with the outer Document handling system (DHS) must be implemented before the realization of displaying of documents (ViewScannedDocs) due to a TEMPORAL dependency – denoted by a precedes relation. Additionally, it shows the previously mentioned Include and Extend relations as REQUIRES dependencies.

4.3.1.4 Solution Analysis

I selected UML Use cases to be extended with planning objectives, since it has many advantages related to project planning: 1) expresses client-valued functions, 2) facilitates communication with users and/or customers, 3) provides traceability of realization, and 4) serves reliable effort estimations (UCPM) [102, 103]. I integrated planning information (such as resources, deadlines, dependencies) with the UML2 Profile mechanism, since this standard mechanism enables extending UML models that were created by any modeling tool that supports the popular XMI format [183]. The implemented UML2 Profile can be found in Appendix D.

I applied the popular UCPM method to estimate duration of each requirement’s realization. Up to now, several researches and case studies reported that the UCPM is a reliable method to estimate the required effort at the early requirements phase [102, 103]. In later phases when requirements uncertainly is not so high, these estimations can be refined with more accurate expert estimates [184].

A variety of algorithms were developed to solve the RCPSP problem class (see Sec. 2.5.3.4) [145, 146]. They include 1) heuristic (finds possible suboptimal), and 2) exact methods (finds optimal solutions for not so difficult problems). To provide suitable scheduling method for wide-ranging iteration scheduling situations, I applied the previously introduced lcap algorithm (see Sec. 4.2.4 and in Appendix C.3). This model enabled to take advantage of computing release plan by the algorithm semi-automatically. The computed values could be represented in AoN (where nodes are activities and arcs are predecessor relations) form to provide further analysis [146]. Additionally, export function makes possible to complete the plan with additional administrative project activities in a popular tool [74].
4.3.2 Tool support

Previously presented theoretical foundation is realized by a software tool on the Eclipse platform [185] – called PYTHIA PROJECT PLANNER™ [186]. This tool was used to support the case study. This prototype is implemented with the UML2 Profile technology [79], since my intend was to create a standard extension to the UML2 Use case diagram. Some screenshots of PYTHIA PROJECT PLANNER™ can be found in Appendix E.2.

The previous Figure 4.4 were produced by this tool. To continue the previous example, Figure 4.5 shows the computed release plan by PYTHIA PROJECT PLANNER™ as an Activity-on-Node activity graph [146] – where UCPM parameters were $MR = 20$, $TCF = 1.035$, $EF = 0.995$, and scheduling objective was makespan minimization.

In this figure dotted and straight lines denote precedence relations (c.f. Figure 4.4) and computed optimized realization orders, respectively. Each activity’s compartment shows estimated duration (EST), start date of realization (relative to the project start date – in working hour), priority (top right, c.f. Sec. 4.3.1.1), allocated resource (bottom), and ID (right).

4.3.3 A Case Study

A longitudinal experiment (case study) was carried out to evaluate the C2.4 part of my contribution [187]. The pilot project was the whole Credit controller module of the EHKR. The previously presented partial examples were extracted from this case study.

In this section, I set hypotheses, present necessary background information, describe study methodology, and finally my findings are presented, validated and interpreted.
4.3. Iteration Scheduling from Use Case-based Specification

4.3.3.1 Setting the hypotheses

My initial intend (see Sec. 4.1.2) was to mitigate the following problems: 1) difficult project planning, 2) poor decision support, and 3) difficult tracing of requirements realization. To validate my elaborated method I stated the following hypotheses:

C2H1: The elaborated method (and tool) with the formulation of software requirements metrics, planning constraints and objectives as RCPSP enables improved production (i.e. less manual work) in project release planning contrary to the traditional and mainly manual method.

C2H2: The elaborated method supports what-if analysis by different parameterizing of the scheduling algorithm which leads to improved decision support (i.e. choose from several alternatives) in release planning decisions.

C2H3: The elaborated method with the derivation of release plans from requirements specification leads to precise requirement’s level tracing.

4.3.3.2 Context and Methodology

EHKR is a medium size (approx. 1.5 million SLOC) client rating and client risk management system for credit institutions. It has been continual evolution since its first release in the middle of 90s. When the case study was performed, fairly small teams (three to eight people) worked on specific functional modules. Staff fluctuation and turnover was low. The system was written in Visual Basic and the applied methodology was a custom agile process.

A single project (the whole Credit Controller module) was selected to make a comparison between the elaborated method and two typical project carried out within the evolutional development of EHKR [165]. The two other past projects were Collateral Value Estimating/CVE/, and Simplified Deal Flow/SDF/. The motives of the selection of projects were same project members, domain, customer, development methodology, and organization structure – so it was possible to compare their results.

Development processes were made up of the following steps. Each project started with the description of customers’ business objectives. Analysts captured requirements with textual scenario descriptions, and the manager agreed with customers on prioritization, and temporal constraints (e.g. deadlines). Then development team was formulated, which interpreted and analyzed dependencies, and structured requirements with Use case modeling (Eclipse UML2 [185]) and textual documents. Gantt charts are manually prepared frequently to ensure project tracing. Two release candidates were deployed to the client’s site before the final release. Testers validated and verified deliverables with scenario-driven manual testing.

In the pilot project the prototype was applied to model requirements, to estimate effort, and to generate schedule. Since the prototype is built on Eclipse UML2 platform the learning time of tool was negligible. The generated schedule was exported into a popular tool [74] regularly.

4.3.3.3 Data collection and Results

The following state variables were used to capture facts that were likely affect the findings: Manager experience, Team experience, Number of developers, Product size, Project duration, and Development effort [102, 100]. These metrics were collected by post-mortar analysis: UCPs were calculated by PYTHIA
4.3. Iteration Scheduling from Use Case-based Specification

PROJECT PLANNER™ (past projects were imported in XML), and the duration were counted based on project’s timesheet records. The collected values are presented in Table 4.8.

<table>
<thead>
<tr>
<th>Variable</th>
<th>CVE</th>
<th>SDF</th>
<th>Pilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager experience [Yr.]</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Team experience [Yr.]</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Number of developers</td>
<td>7</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Size [UCP]</td>
<td>73</td>
<td>58</td>
<td>71</td>
</tr>
<tr>
<td>Duration [Month]</td>
<td>1.5</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>Effort [TPH]</td>
<td>1767</td>
<td>1378</td>
<td>1400</td>
</tr>
</tbody>
</table>

Table 4.8: Comparison of Projects’ State Variables

Project planning was consisted of one initial and two re-planning due to requirements’ uncertainties and feature creeps. Plans were created on the 2nd, 3rd and 5th week alike – owing to the similar durations (c.f. Table 4.8).

The following response variables were used to test C2H1:

1. Project definition effort (PDE) – managerial effort to identify main tasks, define deadlines, and select project members (in [hour])
2. Project planning effort (PPE) – managerial effort to reveal tasks from specifications and discussions, define functional dependencies, estimate duration, and identify cost-, and time-related dependencies (in [hour])
3. Feature creep ($\Delta UCP_{i+1,i}$) – counted as UCP between two planning cases (in [%])

The planning effort was collected from the project’s timesheet, and the creep was calculated from Use case specifications. The Table 4.9 shows the measured values.

Explanations of C2H2, C2H3 were produced with the utilization of the solution’s inherent properties.

<table>
<thead>
<tr>
<th>Variable</th>
<th>CVE</th>
<th>SDF</th>
<th>Pilot</th>
<th>1-Pilot absolute</th>
<th>1-Pilot absolute</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE1</td>
<td>1.5</td>
<td>1</td>
<td>1</td>
<td>0.33</td>
<td>0</td>
</tr>
<tr>
<td>PPE1</td>
<td>4.5</td>
<td>3</td>
<td>1</td>
<td>0.78</td>
<td>0.67</td>
</tr>
<tr>
<td>PPE2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>PPE3</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>$\Sigma PE$</td>
<td>9</td>
<td>7</td>
<td>3.5</td>
<td>0.61</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>CVE</th>
<th>SDF</th>
<th>Pilot</th>
<th>$\Delta UCP_{i+1,i}$</th>
<th>$\Delta UCP_{i+1,i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCP1</td>
<td>56</td>
<td>49</td>
<td>55</td>
<td>0.02</td>
<td>-0.12</td>
</tr>
<tr>
<td>$\Delta UCP_{2,1}$</td>
<td>17.6</td>
<td>9.3</td>
<td>16.7</td>
<td>0.05</td>
<td>-0.8</td>
</tr>
<tr>
<td>UCP2</td>
<td>68</td>
<td>54</td>
<td>66</td>
<td>0.03</td>
<td>-0.22</td>
</tr>
<tr>
<td>$\Delta UCP_{3,2}$</td>
<td>6.8</td>
<td>6.9</td>
<td>7</td>
<td>-0.03</td>
<td>-0.01</td>
</tr>
<tr>
<td>UCP3</td>
<td>73</td>
<td>58</td>
<td>71</td>
<td>0.03</td>
<td>-0.22</td>
</tr>
</tbody>
</table>

Table 4.9: Comparison of Projects’ Response Variables

4.3.3.4 Case Study Analysis

Table 4.9 demonstrates my expectations that PDEs were similar due to the same development coordinator activities (it is carried out at the first iterations only). Contrary, pilot’s PPEs were considerably smaller (planning acceleration is expressed with 1-Pilot absolute) than the other two projects with average 57%. This finding can be explained by both the integration of Use cases and release planning data, and the
4.4 Goal-oriented Extension of Iteration Scheduling

Plan generation of the scheduling algorithm. The integration exempts the manager from 1) collecting the main development task (use cases, actors), 2) identifying functional dependency (Include, Extend relations), 3) estimating effort separately (UCPM), and finally 4) schedule generation relieves from the significant manual work related to Gantt chart production. Therefore, this significant issues leads us to accept hypothesis C2H1: this method enables more than 50% growth in planning productivity.

In earlier projects only one release plan was constructed due to the complexity of manual planning. Contrary, the RCPSP formulation provides opportunities to carry out what-if analyzes with easy modification of planning problem parameters respect to increase/decrease number of developers, scope (use cases, actors), and temporal constraints. In pilot project, the development coordinator produced and analyzed 2-3 alternatives in each iteration before the decision was made. More decision alternatives provided more informed and established decisions (in contrast to manual planning), so it leads us to accept hypothesis C2H2.

As previously mentioned, Use cases determine client-valued functions. These functions are gained from requirements specifications to provide elements of release plans so as to provide traceability of the requirements’ realizations. Although re-planning can be accomplished in traditional planning, in control projects (CVE and SDF) the release plans were not completely re-planned (e.g. durations were not updated as the development progressed) due to the difficulty of manual planning. This omission yielded more imprecise dates compared to the pilot, where replanning was carried out many times in consequence of the tool support. For this reason both planning element derivation from specification, and more precise dates in release plans lead us to accept hypothesis C2H3: this method provide precise Use case level requirement’s tracing.

4.4 Goal-oriented Extension of Iteration Scheduling

In this section I introduce extension elements to the previously introduced Use case-based iteration scheduling (Sec. 4.3) to integrate goals and scenarios modeling techniques into iteration scheduling (see Sec. 4.2.3) (C2.5). Finally, the analysis of my elaborated solution is presented.

4.4.1 Mapping GRL to Use Case-based Specification

In software engineering, client-valued functions are typically expressed with scenarios. Scenarios help in communicating with the customer and improve mutual understanding of the system to be developed [78, 71]. There are many ways of expressing scenarios, such as Activity-, Sequence Diagram in UML [79], or Use Cases Maps [81, 82]. They express fairly detailed design (how aspect). The reason why I have selected Use case is to complement GRL, since project estimation and planning must be carried out at the early requirements’ phase.

There are two complementary elements to complete the high level Use case specification: i) textual description (e.g. Cockburn’s form [80]), and ii) UML interaction definition (e.g. Activity, Sequence diagrams) to specify the dynamic’s aspect of the system [29, 80]. These elements are not relevant according to my high level modeling aspect.

In the goal-scenario approach GRL goals are operationalized into GRL tasks and these are elaborated in (mapped to) use case scenarios i.e. UML use cases [81, 82].
4.4. Goal-oriented Extension of Iteration Scheduling

### Table 4.10: Mapping GRL to Use Case-based Specification

<table>
<thead>
<tr>
<th>GRL Object</th>
<th>Use Case Object</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Actor</td>
<td>CreditController ↦→ CreditController</td>
</tr>
<tr>
<td>Task</td>
<td>Use Case</td>
<td>CheckDisbursementDoc ↦→ CheckDisbursementDoc</td>
</tr>
</tbody>
</table>

The Figure 4.6 (right) illustrates some of the Use case concepts with the formerly shown CCM example, and two from GRL task to use case mapping (dotted lines). The figure (left) illustrates some of these concepts with a partial example extracted from the EHKR application [165]. It continues the previous example of Section 4.3.1.3.

#### 4.4.2 Tool support

The previously presented theoretical foundation is implemented in the previous PYTHIA PROJECT PLANNER™ [186] tool. This tool was used to support the case study. Whereas my intend was not to create a custom-designed metamodel, I have chosen the popular Profile mechanism of UML2. The above mentioned Figure 4.6 (right), Figure 4.4 and XML input were produced by this tool [74].

#### 4.4.3 A Case Study

A longitudinal experiment (case study) was carried out to evaluate the C2.5 part of my contribution [187]. This pilot project was the same as in the Section 4.3.3, the Credit controller module of the EHKR.

#### 4.4.3.1 Setting the hypotheses

My initial intend (c.f. Sec. 4.1.2) was to mitigate the following problems: 7) *longer iteration time periods* (because of implementation rework) (C2P7), 8) *more costly realization* of the required solution (C2P8),
and 9) poor planning support (realizing not the mostly demanded requirements) (C2P9). To validate my elaborated method I stated the following hypotheses:

C2H4: Fusioning of requirements metrics and project planning constraints and objectives leads to improved efficiency in project execution in addition to less synchronization and documentation overhead. (see above 7,8)

C2H5: Derivation of AISP from software requirement specification enables improved production in project planning contrary to the application of the traditional, mainly manual, project planning method. (see above 9)

4.4.3.2 Context and Methodology

The context of the study can be found in Sec. 4.3.3.2, but the object of the investigation is extended with two additional past projects: namely Basel2Rating /B2R/, and CreditRating/CrR/.

4.4.3.3 Data collection and Results

Commonly, project effort predictions are usually based on the size of the demanded system and form the equation \( PM = A \cdot (\text{Size})^B \cdot \prod EM_i \), where \( PM \) expresses effort in person month, Size is in SLOC, \( EM_i \) integrates 17 cost factors (e.g. technical, environmental and complexity), \( B \) is a scaling exponent (e.g. process maturity) and \( A \) denotes productivity [100]. As set forth, effort – related to UCP – can be expressed by \( TPH = UCP \cdot MR \) (Eq. 1.7) also.

The two equations point out that i) UCP and SLOC equations express complexity by \( AW_i, UW_i \) and \( EM_i \) (where \( \leq i \leq n \)) as linear factors respectively (c.f. Eq. 1.5, [100]); additionally 2) both equation emphasize linear influence of productivity to effort by factor \( A \) and \( MR \) respectively. As a consequence, the next relation (Rel. 4.5) can be stated.

\[
\text{Effort} \propto \text{Productivity} \cdot (\text{Size}) \cdot (\text{Complexity}) \quad (4.5)
\]

The following variables are used to capture facts that are likely affect the findings of project execution’s acceleration (for C2H4) (all other environmental and technical factors were leaved out on account of their substantial similarity):

- Project manager experience in year [100]
- Team experience (domain and cohesion) in year [100]
- Number of developers
- Product size in SLOC [188] and UCP [102]
- Cyclomatic complexity (CC) of source code [188]
- Project duration in month [100]
- Development effort in total working hour (TPH) [102]
All metrics above were calculated by post-mortal analysis of projects. UCPs were calculated by PYTHIA PROJECT PLANNER™ (past projects were imported in XML); SLOC and complexity were calculated by VB Law Workbench [189]. The duration and effort were measured based on project’s timesheet records. The collected values are presented in Table 4.11.

**TABLE 4.11: Comparison of Project Measures - I.**

<table>
<thead>
<tr>
<th>Variable</th>
<th>CVE</th>
<th>B2R</th>
<th>CrR</th>
<th>SDF</th>
<th>Pilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager experience [Year]</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Team experience [Year]</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Number of developers</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Size [KSLOC]</td>
<td>30.5</td>
<td>36.3</td>
<td>15.6</td>
<td>23.2</td>
<td>31</td>
</tr>
<tr>
<td>Size [UCP]</td>
<td>73</td>
<td>51</td>
<td>25</td>
<td>58</td>
<td>71</td>
</tr>
<tr>
<td>Complexity (CC)</td>
<td>3.61</td>
<td>2.26</td>
<td>2.72</td>
<td>3.66</td>
<td>3.49</td>
</tr>
<tr>
<td>Duration [Month]</td>
<td>1.5</td>
<td>1.6</td>
<td>1</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>Effort [TPH]</td>
<td>1767</td>
<td>1077</td>
<td>517</td>
<td>1378</td>
<td>1400</td>
</tr>
</tbody>
</table>

Project planning was consisted of one initial planning and two re-planning accomplished by the development coordinator due to requirements’ uncertainty and feature creep. These projects plans were created on the 2nd, 3rd and 5th week alike – owing to the similar durations (c.f. Table 4.11). The following variables were used to measure the planning effort and requirements’ variability (for C2H5):

- Planning effort (PE) – manual effort to produce Gantt chart by the manager – in hour
- Feature creep (ΔUCPi+1,i) – counted as UCP between two planning cases – in percentage

The planning effort was collected from project’s timesheet, and the creep was calculated from Use case specifications. The Table 4.12 shows the measured values.

**TABLE 4.12: Comparison of Project Measures - II.**

<table>
<thead>
<tr>
<th>Variable</th>
<th>CVE</th>
<th>B2R</th>
<th>CrR</th>
<th>SDF</th>
<th>Pilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>PE2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>PE3</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Σ PE</td>
<td>9</td>
<td>8</td>
<td>5.5</td>
<td>7</td>
<td>3.5</td>
</tr>
<tr>
<td>UCP1</td>
<td>56</td>
<td>41</td>
<td>20</td>
<td>49</td>
<td>55</td>
</tr>
<tr>
<td>ΔUCP1,1 [%]</td>
<td>17.6</td>
<td>12.8</td>
<td>31</td>
<td>9.3</td>
<td>16.7</td>
</tr>
<tr>
<td>UCP2</td>
<td>68</td>
<td>47</td>
<td>29</td>
<td>54</td>
<td>66</td>
</tr>
<tr>
<td>ΔUCP2,2 [%]</td>
<td>6.8</td>
<td>7.8</td>
<td>-16</td>
<td>6.9</td>
<td>7</td>
</tr>
<tr>
<td>UCP3</td>
<td>73</td>
<td>51</td>
<td>25</td>
<td>58</td>
<td>71</td>
</tr>
</tbody>
</table>

4.4.3.4 Case Study Analysis

As Table 4.11 and 4.12 illustrated the company baseline projects were very similar to the pilot project in respect to experience, duration and feature creep. Although Table 4.11 draw attention to the fact that there is ΔMR = 1 – (MRpilot/MRbaseline) * 100% = 12% significant improvement in efficiency expressed in MR (c.f. Eq. 1.7) in the pilot project, where MRpilot = TPH/UCP = 1400/71 = 19.72 and MRbaseline = (24.2, 21.12, 20.68, 23.76)/4 = 22.44 – details in Figure 4.7.

Additionally, to provide validity I constructed a simple metric SC = TPH/(KSLOC * CC) to investigate productivity considering SLOC and complexity as the main driving factor of effort (c.f. Rel. 4.5). This metric also indicates considerably speed up in the pilot project (c.f. Sec. 4.2.6.3): ΔA = 1 – (Apilot/Abaseline) * 100% = 1 – (SCpilot/SCbaseline) * 100% = 10.1%, since Apilot ∝ SCpilot =
1400/(30.952 * 3.49) = 12.96 and \( A_{\text{baseline}} \propto SC_{\text{baseline}} = (16.07, 13.13, 12.19, 16.25)/4 = 14.41 \) due to Rel. 4.5 – statistical details depicted in Figure 4.7.

This data triangulation (multiple measures based on different sources) reinforces my findings. These results can be interpreted as RCPSP-based solution is more refined against manual planning. Its advantages include 1) shorter iteration time period and less costly implementation because of less iteration rework; 2) efficient handling of complex dependencies between requirements’ realizations, which constitutes less waiting time between activities (waiting to each other); 2) better utilization of resources with the extensive exploration of the problem space by scheduling algorithm (finding better solution); 3) faster re-planning, which leads to less waiting time also. Theoretically both 2) and 3) can be dealt with manual method, they require exhausting efforts from the manager, so they are most times omitted. As a consequence, these results lead us to accept hypothesis C2H4: this method enables more than 10% increase in project execution’s efficiency.

Table 4.12 demonstrates that similar feature changes (\( \Delta UCP_{i+1,i} \)) – except project CrR – positively correlate with planning efforts. Despite project CrR had twice as many changes and less than half as large as the pilot, pilot’s planning effort was significantly smaller. Comparing planning efforts, pilot’s PE (\( PE_{\text{pilot}} = 3.5 \)) was considerably smaller than the company baseline’s (\( PE_{\text{baseline}} = (9; 8; 5.5; 7)/4 = 7.37 \)) with \( \Delta PE = 1 - \frac{PE_{\text{pilot}}}{PE_{\text{pilot}}} = 52\% \). This finding can be explained by both 1) integration specification with business objectives to form a common, consistent repository for both development and management; and 2) automatic schedule generation by automatic schedule generation applying the previous results (C2.2-4). The integration exempts the manager from collecting the main development tasks, moreover the schedule generation relieves development coordinator of his significant manual work related to estimation and Gantt chart production. Therefore, this significant issue leads us to accept hypothesis C2H5: this method enables more than 50% growth in planning efficiency in addition to less synchronization and documentation overhead.

The \( TPH = \{1767, 1077, 517, 1378, 1400\} \) wh (working hours) was between \{504(= 3 * 168), 5040(= 30 * 168)\}, therefore the calculation of UCMP was also valid. These facts lead us to generalize my finding beyond the presented case study to the goal-scenario approach using the presented technology elements.

### 4.5 Discussion

My elaborated method intends to provide a sound decision support to the typical constraints and objectives of agile iteration scheduling. These problems are C2P1) precedences (to express temporal precedences between realizations), C2P2) balancing resource workloads (to avoid resources overloading), and
4.5. Discussion

C2P3) *optimality* (to choose the best one from different plans).

First I constructed a general agile iteration planning information model (AISM) (C2.1) that helped us to identify the objects and the subject of my elaborated optimization model. Its precise relationships can also be used as database schema definition for an agile planning and scheduling application such as our SharePoint-based prototypic tool for collaborative data collection.

Then I formulated iteration scheduling model (AISP) (C2.2) as a special case of RCPSP problem to provide decision support in feature implementation sequencing. The formulated model considers temporal constraints (Sec. 4.2.1, 4.2.3.1), team’s resources, and defines makespan minimization scheduling objective. As a matter of fact many different objectives are possible – depending on the goals of the decision makers – but in my scheduling case (‘maximize users’ and/or customers’ satisfaction in the least time possible’) the makespan minimization is the most adequate. This interpretation of iteration schedule makes it possible to adapt extremely successful heuristic algorithms applied for solving RCPSP. To provide suitable scheduling method for wide-ranging iteration scheduling situations I extended the ordinary RCPSP problem with i) pre-assignments (i.e. assigning certain tasks to resources before scheduling) and ii) timeboxed iteration duration control.

Generally, RCPSP problems (see Sec. 2.5.3.4) are combinatorial \( \mathcal{NP} \)-hard problems (see Appendix F.2.1) and a variety approximation algorithms are elaborated. The most popular heuristics in approximation algorithms are SPT or LTP (Shortest/Longest Processing Time first) [146]. However, I constructed and applied my *assigned task first (AF)* scheduling rule demanded by pre-assignments (defect corrections and onward development of a formerly delivered functionalities). My elaborated combinatorial algorithm (C2.3) is capable to provide acceptable results with good time complexity (\( O(n + m) \) – see Appendix F) for practical applications.

Despite planning and development should be based on a common collection of agreed requirements, they usually separated into two repositories, which is the root of many problems, such as 4) *difficult iteration planning* (collecting planning data and producing Gantt charts etc.) (C2P4), 5) *poor decision support* (weak tool support etc.) (C2P5), and 6) *difficult tracing of requirements realization* (due to requirements changes etc.) (C2P6). These problems lead to imprecise planning and improper management of requirements, and all of them result in lower quality, increased efforts and additional project risks.

To diminish the negative effects of the C2P4-P6 I elaborated 4) a generic extension of UML-based specification with planning objectives by constructing a domain-specific metamodel (C2.4). This extension can be used as a consistent iteration scheduling repository that enables schedule generation applying the previous results (C2.2-3).

Requirements from various users and/or customers are often expressed as objectives or goals. Directly modeling goals, rather than entities, allow searching for alternatives before deciding a particular solution. Moreover, goal representation enables validating requirements’ completeness with tasks and scenarios. Although planning and development should be based on common collection of requirements that aligned with the business goals, they are separated in different repositories: goal definitions in business specifications and requirements in the system requirements specifications. This separation may lead to solutions that are not aligned with the business goals and it results in 7) *longer iteration time periods* (because of implementation rework) (C2P7), 8) *more costly realization* of the required solution (C2P8), and 9) *poor planning support* (realizing not the mostly demanded requirements) (C2P9). These problems lead to improper satisfaction of business needs and results in increased efforts and additional project risks.
4.6. Conclusions

I also elaborated 6) a generic extension of UML-based specification with business objectives to eliminate the previous problems C2P7-P9. It forms a common, consistent repository for both development and management (C2.5). Additionally, this solution also enables automatic schedule generation applying the previous results (C2.2-4).

This approach gives the business increased visibility, and it can also provide constantly up-to-date schedule decision support considering changes necessitated by shifting business priorities. Moreover, the decision maker can accommodate quick what-if scenarios and replanning on-the-fly. However, as my simulation carried out post mortem analysis, examination of the method is recommended in real development cases in order to investigate it in dynamical situations.

4.6 Conclusions

In this chapter, I presented my novel concepts, models and algorithms for agile iteration planning. The elaborated method significantly improves load balancing of resources ($\approx 4 - 5\times$), significantly accelerates iteration scheduling production (> 50%), enables more than 10% increase in project execution’s efficiency, and more than 50% growth in planning efficiency comparing to the traditional approaches.

I carried out a post mortem analysis on four real-life representative data sets extracted from a software development company [165] to validate the C2.1, C2.2 and C2.3 parts of my contribution. Additionally, a case study was carried out to evaluate my elaborated method (c.f. C2.4, C2.5) on a real-life software planning situation [187]. The pilot project was the whole module of a real-life application that is developed by [165].

Additionally, I also carried out simulations on 480 generated representative data sets – by varying parameters of the iteration problem (see Sec. 1.4.3: effort for developing technical tasks ($w$), resource capacities ($R$), iteration capacity ($c$), iteration length ($l^I_k$) and precedences ($P$)) – to get an insight into the performance and quality of the presented approach and to filter out the statistical staggering of different agile iteration planning problems.

The results of experiments on representative real-life data sets, representative generated data sets and representative case studies indicate that my approach can provide practical value as a decision support method for agile iteration planning.

The summary of the Contribution 2 can be found in the following box:

<table>
<thead>
<tr>
<th>C2: I have elaborated a novel method (namely Agile Iteration Scheduling (AIS)) for iteration planning to provide improved efficiency and effectiveness in iteration-centered decisions.</th>
<th>[C5, I26, D7, E9, E14, E11, E8, I25, G18, G16]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2.1 I have constructed a consistent ontology-styled information model of agile iteration scheduling (namely Agile Iteration Scheduling Model (AISM)) to specify the components of the agile iteration scheduling decision space. This model involves the main concepts and their relations, and can be applied as a conceptual model of agile iteration planning tools.</td>
<td>[C6, C5]</td>
</tr>
<tr>
<td>C2.2 I have formulated agile iteration scheduling as an optimization problem (namely Agile Iteration Planning (AIP))</td>
<td></td>
</tr>
</tbody>
</table>
4.6. Conclusions

The Iteration Scheduling Problem (AISP) to provide governance support in feature implementation sequencing. The main novelty of my approach lies in the mathematical precise formulation of the problem. The elaborated problem formulation is solved with an extension of the RESOURCE-CONSTRAINED PROJECT SCHEDULING (RCPS) model that considers temporal constraints, team’s resources, and defines makespan minimization scheduling objective. Moreover, it is an extension of the RCPSP to cover the iteration scheduling situations with the expression of 1) pre-assignments (i.e. assigning certain tasks to resources before scheduling) and 2) timeboxed iteration duration control. [C5]

**C2.3** I have developed a heuristic scheduling algorithm for the RCPSP problem. Its greedy strategy misses the global optimal solution, but produces quick (time complexity is clearly $O(n + m)$) and usually sufficient results for practical applications. [C5]

**C2.4** I have constructed an information model to extend UML diagrams with planning objectives and constraints. The extension is constructed with the application of standard UML Profile extension mechanism [79]. I have formulated a mathematical formulae to determine the required effort for implementing a Use case model element based on the popular Use-case point estimation method (UCPM) [102]. With this information fusion and the formulated equation semi-automatic model-driven planning can be exploited from UML artefacts applying the previous results (C2.2-3). Additionally, I developed a graph transformation to create an AoN graph (Activity-on-Node) from the schedule in order to make schedule visualization possible in popular project planning tools such as [74], [E9, E14, E11, E8]

**C2.5** I have constructed a generic extension of UML-based specification with business objectives, which are expressed with the Goal-oriented Requirements Language to form a common, consistent repository for both development and management. This information fusion enables easy alignment of system requirements with business needs. Moreover, it also enables automatic schedule generation applying the previous results (C2.2-4). [E9, E14, E11, E8]
Chapter 5

Distributed extension of Agile Release Planning

Distributed software development have been becoming a common business reality. Software development organizations are striving to blend agile development methods and distributed development to reap the benefits of both. However, agile and distributed development approaches differ significantly in their key tenets. While agile methods mainly rely on informal processes to facilitate coordination, distributed development typically relies on formal mechanisms. In this chapter I introduce my third contribution (C3) that is a novel method (namely Feature Partitioning Method) for distributed agile release planning to provide improved efficiency and effectiveness in release-centered decisions in distributed agile environments.

5.1 Synopsis

The seven State of Agile Development survey was conducted in 2012 [42]. The collected data collected worldwide from 4,048 participants – ranging from project managers, development managers, developers and senior managers. The survey revealed that the 35% of the respondents worked in distributed agile environments – where the members of the teams are physically dispersed. Actually, the dispersion of team members ranges from being over adjacent buildings to being over different continents. The key advantages that Distributed Software Development (DSD) aspires to achieve are 1) lowering cost of labor (cost reduction), 2) increasing or decreasing work forces without employing or laying-off (workforce scaling), and 3) obtaining locally not available expertise (talent application) [56]. The special case of DSD is the Global Software Development (GSD) in which the team distribution extends national boundaries [43]. The GSD allows organizations to overcome geographical distances, to benefit from accessing a larger resource pool and to reduce development costs [55].

The following chapter present my novel concepts, models and algorithms for agile distributed release planning. In the chapter, I present an distributed agile release planning approach (C3.1) to identify feature chunks that can be implemented co-located to minimize the communication needs between dispersed
teams. The presented method demonstrates how this approach 1) necessitates less intensive communication and coordination, 2) can provide better utilization of resources, and 3) can produce higher quality feature distribution plans. Finally, the chapter analyzes benefits and issues from the use of this approach.

The presented solutions extends the agile release scheduling method that was presented in Chapter 3.

5.1.1 Challenges in Distributed Software Development

Several additional challenges may be observed in DSD comparing to the co-located situations [43, 44, 45]. Agile development usually relies on frequent informal interactions. However in DSD, the teams cannot see or speak in person which leads to communication deficiency (Ch1) due to the geographical separation [45]. Communication impedance also raises from time zone differences that also hinders the teams’ communication. As a solution, DSD mandates that the development relies on formal documentation (such as specifications, designs) to mitigate impediments of communications between the teams.

Agile development is usually based on shared view of goals that are difficult to observe in separated locations – and it often induces lack of trust (Ch2) among dispersed teams [45]. To improve team cohesion in distributed software development, frequent personal communication is required.

Agile development usually mandates using people-oriented control, which is based on informal commitments. In these situations, the development is based on ongoing negotiation on the requirements between the developers and the customers. However, in distributed environments, it often leads to the lack of control (Ch3) [45]. Due to the lack of communication, the general DSD often relies on process-oriented development and upfront commitments to meet the customer expectations on every development location.

Addition to the previous challenges Ch1-Ch3, in [45] it was demonstrated that projects in distributed environments take about two and one-half times longer to complete – comparing to similar projects where the project team is co-located. The significant difference was explained by the communication and coordination issues rather than the size or complexity of the cross-site development [57]. As a consequence, distributed software development requires considerable effort from the team in order to be truly successful [72].

5.1.2 Related Work

Considering non-agile DSD, in [123] a method is offered to calculate the degree of relatedness of the work items at different sites using code change history. The calculated relatedness is used to distribute work in a way that minimizes the need for coordination across sites. In [124], experiences of a rapid production process are described using software components suited for distributed development in a large, geographically distributed situation. In this approach, each component can be owned by a particular site to promote independent work and to minimize coordination and communication needs.

Software outsourcing is an increasingly attractive business model for many large organizations. In [55] three outsourcing strategies are presented to maximize business value. In [121], good practices are presented that were observed in a very large (5,000 engineers) globally distributed development situation at Alcatel.
5.1. Synopsis

In comparison to the extensive research on agile software development in general, only few research dealt with DSD in the agile environment in specifically [44]. Experiences and practices of the adoption of Scrum [32], an agile management practice, by large companies such as Yahoo! or Microsoft is presented in [125] and in [126] respectively. In [72] experiences and proven practices to address challenges faced by geographically distributed agile teams are presented by the Microsoft’s Patterns & Practices group. It pointed out that the decision makers must understand risk/reward tradeoff needs before deciding to distribute software development, because it decreases the project’s likelihood of success, increases the delivery time and quality, and reduces the team’s performance. Besides cross-locations, differences in culture and language also results in low progress in globally distributed environments. To cope with these issues, in the literature, some strategies are elaborated including the use of straddlers (technical or managerial liaisons) [55], bridgehead teams [122], or rotation of management [121].

5.1.3 Problem Statement and Analysis

As it was shown, DSD has its own unique set of challenges (Ch1-Ch3) additional to the agile software development in itself. These challenges are emanated from inhibited communication because the teams are geographically separated from each other [45, 57]. These obstacles of communication – especially informal communication – plays a critical role in the success of a distributed agile team, since they heavily rely on personal physical interactions [123].

The obstacles of informal communication seem a contradiction to the general ideas of agile methods [35], and they seem to preclude the use of agile methodologies [37, 34, 57, 45]. Communication and coordination problems result in reduced team productivity (C3P1), increased production interval (C3P2), increased communication cost (C3P3), and difficult process control across distributed teams (C3P4).

The desired reduction of informal communication deficiency effects can be accomplished by two general ways with minimizing communication among the dispersed teams. One of them is increasing the formality of the interactions with detailed documentation (i.e. specifications, designs) and conventions (i.e. coding standards, templates). In this way, theoretically, the lack of communication can be diminished. Although, this method contradicts to the ideas of agility – however, it is a well-tried approach to aim at geographical distances. The other method can be realized by decreasing the need of interactions between the teams. With this approach, the communication and synchronization needs are minimized, which is also a reasonable solution to the communication problem.

5.1.4 Objectives

My contribution is inspired by David Parnas’ and Melvin Conway’s work. David Parnas recommends division of labor along with software modularity, and he defined software module as ‘a responsibility assignment rather than a subprogram’. He emphasizes the idea of modular design that enables independent decisions about the internals of each module [190]. Additionally to Parnas, Melvin Conway recognized – which is known as Conway’s Law – that the structure of software reflects the structure of the organization which designed it [191]. Conway explained this relation with that it is the necessary consequence of the communication needs between people as they are doing their work. Since these proposals appeared, researchers and practitioners have been arguing for that the system architecture plays a significant role in the coordination of development work.
My elaborated method intends to minimize communication and synchronization needs (Ch1-Ch3) to minimize their negative effects (C3P1-C3P4). As a part of my solution – named Feature Partitioning Method (FPM) – I defined: 1) Feature Architectural Similarity Analysis (FASA) and 2) Feature Chunk Construction (FCC) steps. FASA is an analytic step to determine architectural similarities between features (deliverable functional and non-functional requirements). Structural similarities can be exploited to identify features that are implemented in the similar sets of software modules (C3.1). FCC is a feature partitioning step by formulating feature partitioning as a MINIMUM K-CUT optimization problem (2.6.4.2). These feature partitions rule the distribution of development work across sites considering minimization of communication and coordination needs among the dispersed teams (C3.2).

In other words, my aim is to decrease the need of interactions between the distributed teams with identifying so-called feature chunks to reduce the communication and coordination issues. These feature chunks can be treated logically roughly independently. My elaborated Feature Partitioning Method (FPM) intends to complement the release planning step in distributed agile environments by determining different sets of features that can be implemented roughly independently. Therefore, they can be realized with minimized communication needs among the sites.

5.1.4.1 Structure of the Chapter

The rest of the chapter is arranged as follows: Sec. 5.3 outlines the elaborated method; Sec. 5.4 shows experiments; Sec. 5.5 discusses my solution and findings; and finally Sec. 5.6 concludes the chapter.

5.2 Distributed Extension of Agile Release Planning

In practice, generally two environmental variables determine the strategy of agile software development [32, 56, 44]. One of them is the team cross-functionality, which determines whether the agile team is acting as a group of people working toward the common goal on every site (i.e. cross functional), or the team is working toward a sub-goal on each site (i.e non-cross functional). The other variable is the team distribution which defines whether the team is located on the same, or on geographically different sites.

From these parameters, the following agile software development approaches are observed in practice 1) Co-located: the single agile team is co-located and cross-functional; 2) Isolated: agile teams are isolated across sites and not cross-functional; and 3) Distributed: agile teams are isolated across sites and cross-functional. Obviously, Co-located approach is out of my scope. Considering both Isolated and Distributed approaches, basically they require communication – especially informal, face-to-face communication – across sites to remove dependencies between work units. However, the communication intensities of these approaches are very different. In Isolated case, each team removes most dependencies locally (within the given site), while in Distributed case, the dependencies must be resolved across sites. Therefore, Distributed case is more difficult to implement due to delays in parties. Consequently, the former model is more often observed in practice [44, 56] and suggested by the Scrum Alliance [192]. My elaborated Feature Partitioning Method also follows the Isolated strategy since this approach aligns with my interaction minimization objective.
Formally, distributed extension of agile release planning (\(\mathcal{A}_F\)) can be defined as a decision making process, where the goal is to determine a feasible feature chunk assignment to each distributed team before the local agile release planning is carried out. Cohesive feature chunks are usually constructed by considering the module relatedness, the development competence, etc. of features. In this chapter, the elaborated feature chunk construction considers the communication and coordination needs (i.e., how many person hours are spent on communication and coordination during a given time period) between the dispersed teams during implementation.

The optimized version of the distributed extension of agile release planning problem can be derived by selecting the extreme-valued plan from the potentially feasible alternatives. This problem can be considered as a representative of MINIMUM CUT graph partitioning problem that requires finding a set of edges (communication paths) between vertices (teams) whose removal would partition the graph into connected components (feature partitions) [77]. The minimization objective reflects the aim of minimizing the intensity of communication and synchronization needs (see Figure 1.4) in order to minimize their negative effects—such as reduced team productivity, increased production interval, increased communication cost and difficult process control across distributed teams [37, 34, 57, 45].

Feature chunks are usually identified according to their cohesiveness from the standpoint of architectural impact. The higher the cohesiveness between features, the stronger the need to group those features together. To identify cohesiveness between features, I introduce a binary relation between features (\(W\)) and software modules (\(\mathcal{M}\)), called ImplementedIn (\(\otimes\)), to express the fact that a given feature is implemented in a given software module. My goal is to group those features which are to be implemented in the similar set of software modules, i.e., they require similar architectural impact. With this approach, arranging development work (features) according to the identified feature chunks, it can significantly decrease the communication needs and coordination complexity of the distributed team (\(T\)). Traditionally, the feature chunk composition is manually accomplished that requires relatively long time (several hours) and the optimality objective (partitioning the features into \(k\) cohesive chunks) hardly can be realized. More formally, we can say that the design space of feature distribution is made up of the previous factors (\(W, \mathcal{M}, T, \otimes, W^*\)), and its objective is to find an optimal \(k\)-partitioning of

\[
\mathcal{A}_F : (W, \mathcal{M}, T, \otimes) \rightarrow W^* \quad (5.1)
\]

This chapter proposes methodical support to this mapping.

Figure 5.1: Distributed Agile Release Planning.
5.3 Feature Partitioning Method

In this section, I detail my previously outlined Feature Partitioning Method. First, I overview the method in the context, then I present its two parts: Feature Architectural Similarity Analysis (C3.1) (Sec. 5.3.1) and Feature Chunk Construction (C3.2) (Sec. 5.3.2).

5.3.1 Feature Architectural Similarity Analysis

The concept of architecture, design pattern and modularity are central in product development. Recent efforts strive for using design patterns and architectural styles, frequently in an informal way, to guarantee the quality of a design [193, 194]. The two important characteristics of modularity are the cohesiveness of modules and the coupling between modules, which describe the interaction intensity between functional elements. Modules are identified in such a way that inter-module (between modules) interactions are relatively minimal (i.e. they are loosely coupled) while intra-module (within each module) interactions relatively high (i.e. they jointly serve a functionality, so they are cohesive) [195].

This underlines the importance of decomposition in distributed software development. Modularity enables development of different functional element groups (modules) in different sites independently, and integration ensures that the whole functionality can be delivered to the customer’s site. As a consequence, objects of feature implementations (i.e. software modules) should be identified during release planning by the selected experienced members and users and/or customers of the distributed team in order to help partitioning features into feature chunks.

5.3.1.1 Module Relatedness of Features.

In order to group features, I introduce the notion of cohesiveness between features from the standpoint of architectural impact. The higher the cohesiveness between features, the stronger the need to group those features together. The elements of application architecture can be defined as separated software modules (technical units) that often include user interface, application logic and data storage parts. To identify cohesiveness between features, I introduce a binary relation between features (FRs) and software modules (SMs), called ImplementedIn, to express the fact that a given FR is implemented in a given SM (i.e. directed, one-to-many relation). Please note, implementing cross-cutting features, e.g. performance or security, require the team to break down them into more specific ones such a way that each specific one can be related to a particular subset of modules. At this point, I have to stress the fact that my elaborated FPM method is based upon two underlying assumptions: the architecture of the system (i.e. set of modules) is defined and the implementation of the identified features can be appointed to a subset of software modules.

Let define module relatedness of FRs as follows. Since every feature must be implemented in the software modules, every feature can be characterized by its ImplementedIn binary relation (More formally, I can say that the design space of feature distribution is made up of the previous factors (⊗) to these modules. As a consequence, characteristic variables of features are binary variables which express the fact that a given FR is implemented in a given SM or not. So it recommends using q number of binary variables (bits) on the ordinal scale expressing whether a given feature relates to the SMh (1 ≤ h ≤ q). From now on, I call this variable as Module Relatedness Variable (MRV), where the variable has q dimensions.
5.3. Feature Partitioning Method

For example, the Figure 5.2 shows some values of the previously defined MRV (in the cells ‘×’ and ‘○’ notations denote presence and absence of ImplementedIn relation). It can be read across an element’s row to see its targeted module.

<table>
<thead>
<tr>
<th></th>
<th>SM1</th>
<th>SM2</th>
<th>SM3</th>
<th>SM..</th>
<th>SMq</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>×</td>
<td>×</td>
<td>○</td>
<td>...</td>
<td>○</td>
</tr>
<tr>
<td>FR2</td>
<td>×</td>
<td>×</td>
<td>○</td>
<td>...</td>
<td>×</td>
</tr>
</tbody>
</table>

**Figure 5.2:** Example Values on Module Relatedness Variable (MRV): each column denotes one dimension and every row shows an instance of the MRV.

5.3.1.2 Feature Architectural Similarity Measure.

Once the relations between features and software modules are determined and the features are characterized with module relatedness variables (MRVs), the next step is to define feature architectural similarity in terms of presence or absence of relations to SMs in each MRV. Let the following variables, which describes the congruence of characteristic variables of two MRVs, are defined:

\[
\begin{align*}
    p &= \text{presence for both objects} \\
    q &= \text{presence for the } i\text{th, absence for the } j\text{th object} \\
    r &= \text{absence for the } i\text{th, presence for the } j\text{th object} \\
    s &= \text{absence for both objects}
\end{align*}
\]

For example, let \( MRV_j = [\times, \times, \times, \bigcirc] \) and \( MRV_{j'} = [\bigcirc, \times, \times, \bigcirc] \) are defined on \( FR_j \) and \( FR_{j'} \) respectively. In this case, \( p = 2, q = 1, r = 0, \) and \( s = 1 \).

Generally, similarity (from now on denoted by \( sim_{j,j'} \)) is a quantity that reflects the strength of relationship between two objects (\( j \) and \( j' \)). It usually has a range of \([-1, 1]\) or normalized into \([0, 1]\). Contrary, a distance measures dissimilarity of two objects is denoted by \( dist_{j,j'} \). The relationship between distance and similarity is given by \( sim_{j,j'} = 1 - dist_{j,j'} \) for similarities that are bounded by \( 0 \) and \( 1 \). When similarity is \( 1 \) (i.e. exactly the same), the distance is \( 0 \) and when the similarity is \( 0 \) (i.e. totally different), the distance is \( 1 \). Often measuring distance is easier than measuring similarity [152]. However, once we can measure the distance, we can convert it to its appropriate similarity measure. Henceforth, I only focus on similarities for simplicity reasons.

Defining or selecting an appropriate similarity measure using the previous variables depends on the characteristics of the problem. Number of various measures defined in the literature in the last decades [152, 154]. For example, Simple matching, i.e. \( (p + s)/(p + q + r + s) \), is useful when both positive and negative values carry equal information (i.e. symmetry). Continuing the previous example, in Simple matching case, the similarity of \( FR_j \) and \( FR_{j'} \) is equal to \( sim = 3/4 \).
5.3. Feature Partitioning Method

My goal is to group those features which are to be implemented in the similar set of software modules, i.e. they require similar architectural impact. Counting the non-existence relations in both objects has no meaningful contribution to the architectural similarity of features, so variable s should be left out from both the denominator and the numerator. As a consequence, to express the architectural similarity between FRs, we need to count the relations (p) to the same software modules and the number of relations between an FR and a SM (i.e. p + q + r), so I formulated the following measure:

\[ \sim_{j,j'} = \frac{p}{p + q + r} \]  

(5.3)

Thus, continuing the previous example, \( \sim_{j,j'} = \frac{p}{p + q + r} = \frac{2}{3} \). It is clear that, the range of both \( \sim_{j,j'} \) and \((1 - \sim_{j,j'})\) (distance) are in \([0, 1]\). Thus, if we can measure architectural similarity/distance of features, then we can use this information to group them into feature chunks according to the values of the measurements.

5.3.2 Feature Chunk Construction

To support the parallel development, I elaborate a feature chunk construction step by formulating feature partitioning as a MINIMUM K-CUT optimization problem (2.6.4.2). The aim of the formulation is to break down the whole development work into cohesive feature chunks. In this case, connected blocks of features are partitioned into smaller blocks in such a way that the decomposition is carried out along the smallest similarities. This decomposition ensures that the communication demand and coordination complexity are reduced since the elements are grouped such a way that communications predominately occur within grouped FRs rather than between grouped FRs.

5.3.2.1 Mapping to Minimum k-cut.

The optimized version of the distributed extension of agile release planning problem can be derived by selecting the extreme-valued plan from the potentially feasible alternatives. In the following analogy between feature partitioning and MINIMUM K-CUT optimization problem (see Sec. 2.6.4.2) is presented. Generally, in graph theory, a cut is a partition of the vertices of a graph. The minimum k-cut, is a combinatorial optimization problem that requires finding a set of edges whose removal would partition the graph to \( k \) connected components. These edges are referred to as k-cut.

The previously introduced similarity (5.3) can be used to compose a non-directed, edge-weighted graph that I call as Feature Architectural Similarity Graph (FASG). It can be constructed by representing features as vertices and relationships between features as edges where the edges are weighted with the strength of the architectural similarity ranging from \( (0, 1] \) (cf. Sec. 5.3.1.2). FASGs can be interpreted as features with their expected communication intensities (weighed edges) between teams because they are implemented in the similar set of modules. At this point, I should lay emphasis on the difference between the notions of the feature architectural similarity (i.e. architecture impact) and the communication intensity between developers. Conway’s law only states that there is a strong correlation between these two notions [191], but this correlation does not imply causation (i.e. cum hoc ergo propter hoc logical fallacy). In other words, there may be some feature implementations that do not necessitate communication between teams. Therefore, the presented FPM method considers a stricter approach: implementation of
5.3. Feature Partitioning Method

Some features do not necessitate implementing them on the same site. However, during release planning it cannot be predicted without systematic and in-depth analysis of the implementable features, software modules and their relations and this detailed up front analysis would contradict the ideas of agility.

Continuing the previous informal definition, let define the Feature Architectural Similarity Graph (FASG) more precisely as the following: The $G_{FS}$ is a non-directed, edge weighted graph where

1. $W$: denote the set of deliverable features, where $j \in W$.
2. $\sim$: express the architectural similarity (see 5.3) between features ($\sim_{j,j'}: j,j' \in W$):

$$\sim_{j,j'} = \begin{cases} 1 & \text{if } \sim_{j,j'} > 0 \\ 0 & \text{otherwise (missing edge)} \end{cases} \quad (5.4a)$$

Therefore, if the $\sim_{j,j'} > 0$, then there is an edge between feature $j$ and $j'$ and the weight of edge is equal to the strength of $\sim_{j,j'}$ (cf. 5.3), otherwise the edge is missing.

To determine the proper feature decomposition, I apply a graph partitioning method on the FASG, which is a common technique in areas such as routing, and VLSI placement [159]. Partitioning $G_{FS}$ into $k$ partitions means that the goal is to find subsets $W_1, W_2, ..., W_k$ of features $j \in W$ such as that

1. $\bigcup_{i=1}^{k} W_i = W$ and $W_i \cap W'_i = \emptyset$ for $i \neq i'$ (subsets are disjoint set of features),
2. the sum of edge weights (similarities) that are crossing subsets is minimized.

5.3.3 Formulating FCCP Optimization Model

Let a $G_{FS} \triangleq (W; \sim)$ is a non-directed, edge weighted FASG and a $k$ integer ($k \in 2, ..., |W|$). The task is to partition the $G_{FS}$ into $k$ disjoint sets ($W_i : i = 1, ..., k$).

Minimize

$$\sum_{i=1}^{k-1} \sum_{i'=i+1}^{k} \sum_{j \in W_i} \sum_{j' \in W'_i} \sim_{j,j'}\quad (5.5a)$$

subject to $\bigcup_{i=1}^{k} W_i = W$ and $W_i \cap W'_i = \emptyset$ for $i \neq i'$ \quad (5.5b)

where Eq. 5.5b is the disjoint constraints of the partitioning and Eq. 5.5a is the $k$-cut minimization objective. More formally, we can say that the design space of feature distribution is made up of the previous factors $(W, M, T, \otimes, W^*)$, and its objective is to find an optimal $k$-partitioning of $\mathcal{A}_F : (W, M, T, \otimes) \rightarrow W^*$.

A partial example of partitioning can be seen in Figure 5.3. This figure shows the result of an FASG partitioning with parameter $k = 8$. In the presented FASG, the similarities of features are shown on the edges and the cost of partitioning is equal to the sum of cut edges ($\approx 5.167$).
5.4 Experimental Results

In order to validate my contribution, first I carried out a post mortem analysis on seven real-life representative data sets extracted from a software development company [165], then I investigated my solution with the numerous representative generated data sets. The latter validation is described in Sec. 5.4.5, the former one is presented in the following.

In post mortem analysis I used the extracted historical data as an input for my elaborated method to made it possible to compare the release planning result of the method with the historical result [167]. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical in terms of organization size (18 developers), applied agile methods (Scrum-like development process) and techniques (XP development practices). At this organization, the release scheduling process is made up of the typical agile planning steps (see Sec.1.4.2).

5.4.1 Research Questions

My initial intends (C3P1-C3P3) was to minimize the effects of informal communication deficiency in distributed agile environments. To validate my method the next questions were addressed: How FPM can be compared with the intuitive one in terms of C3Q1 communication and coordination needs, C3Q2 resource workload, and C3Q3 feasibility?
5.4. Experimental Results

5.4.2 Context and Methodology

IRIS is an integrated client rating and client risk management system for credit institutions for analyzing the non-payment risk of clients and indicating potential clients so as to realize their future liabilities on the basis of their solvency and attitude. The system is written in VB and C# (approx. 2 million SLOC, 41 modules), the applied methodology is a custom agile process.

The release planning process (Sec. 5.2) consisted of the following steps. First, the features were selected (expressed in User stories [67]) from the backlog – considering users’ and/or customers’ demands. Then every User story was estimated by two teams and assigned to these teams intuitively. The fix-membered teams worked in different locations, so they could not see or speak often in person that resulted from geographical separation. Communication was mostly based on video conferences, phone calls and emails; since all developers were Hungarians, there was no language, cultural or time zone barriers.

5.4.3 Data Collection

Seven data sets (seven releases: \(R_1 - R_7\)) were selected to make a comparison between the algorithmic and the intuitive method. These data sets were results of typical agile processes and as a consequence they were appropriate for analysis. All releases had the same iteration length (80 working hours, i.e. 2 weeks), domain, customer, and development methodology, but they were characterized by different number of User stories \((US - \text{deliverable features})\), team capacities \((TC - \text{the amount of deliverable Story points [67] by the team in the releases})\), effective developer workforce \((ED - \text{available developers})\), and delivered User stories at the end of the releases \((RUS - \text{in Story point})\). (Please note, the value of a Story point correlates with the required implementation effort of a User story.) The cardinality interval of module relatedness was \([1, 3]\).

Table 5.1 summarizes the state variables that were used to capture facts that were likely affecting the findings, where values between round brackets pointing out how these variables were divided between the sites of team 1 and 2 – respectively. These variables were collected from the IRIS’s backlog.

<table>
<thead>
<tr>
<th></th>
<th>(US)</th>
<th>(TC = (TC_{T1},TC_{T2}))</th>
<th>(ED = (ED_{T1},ED_{T2}))</th>
<th>(RUS_p = (RUS_{T1},RUS_{T2}))</th>
<th>(\text{CCI}^{\text{hist}})</th>
<th>(\text{CCI}^{\text{alg}})</th>
<th>Efficiency ((\text{CCI}^{\text{hist}}/\text{CCI}^{\text{alg}}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_1)</td>
<td>88</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>331.0 (175.5,155.5)</td>
<td>256.25</td>
<td>12.67</td>
<td>20.23</td>
</tr>
<tr>
<td>(R_2)</td>
<td>119</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>379.0 (210,0169.0)</td>
<td>203.70</td>
<td>14.42</td>
<td>14.13</td>
</tr>
<tr>
<td>(R_3)</td>
<td>131</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>342.0 (197.0,145.0)</td>
<td>310.46</td>
<td>15.83</td>
<td>19.61</td>
</tr>
<tr>
<td>(R_4)</td>
<td>95</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>402.5 (227.5,175.0)</td>
<td>228.33</td>
<td>11.85</td>
<td>19.27</td>
</tr>
<tr>
<td>(R_5)</td>
<td>110</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>324.5 (180.5,144.0)</td>
<td>275.53</td>
<td>14.40</td>
<td>19.13</td>
</tr>
<tr>
<td>(R_6)</td>
<td>101</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>320.0 (176.0,144.0)</td>
<td>255.53</td>
<td>13.40</td>
<td>19.07</td>
</tr>
<tr>
<td>(R_7)</td>
<td>91</td>
<td>360 (200,160)</td>
<td>18 (10,8)</td>
<td>386.0 (208.0,178.0)</td>
<td>240.00</td>
<td>12.49</td>
<td>19.22</td>
</tr>
<tr>
<td>(\sum)</td>
<td>735</td>
<td>2520.00</td>
<td>126</td>
<td>2484</td>
<td>(\sum) (\text{CCI})</td>
<td>1769.82</td>
<td>95.06</td>
</tr>
</tbody>
</table>

5.4.4 Results and Analysis

To answer the questions of C3Q1-C3Q3, simulations were performed on the input data (summarized in Table 5.1) to compare the characteristics of the two approaches.
5.4. Experimental Results

I constructed a response variable to test C3Q1 (Sec. 5.4.1) – namely the Communication and Coordination Intensity between teams (CCI) in terms of the sum of cut edges in FASG – in order to set the historical (CCI\textsubscript{hist}) cases against the algorithmic (CCI\textsubscript{alg}) cases. As my aim is to promote independent work and to minimize the coordination and communication needs between teams, the less CCI value is treated better.

In Table 5.2, rows from R\textsubscript{1} to R\textsubscript{7} show that the historical cases had far greater CCI value than the algorithmic ones. This points out the fact that, due to the intuitive feature distribution, feature chunks were less efficiently separated (CCI\textsubscript{hist}/CCI\textsubscript{alg}) in the historical cases comparing to the algorithmic ones. The $\sum$ CCI summarizes the differences between the historical and the algorithmic cases, and points out a remarkable difference of average communication and coordination intensities – it resulted in $\nabla_{CCI} \triangleq CCI_{hist}/CCI_{alg} = 1769.82/95.06 \approx 18.62 \times$ less in the algorithmic cases.

Considering the historical data (Table 5.1), the planned average velocity of User story development per person is $20$, and the planned team capacities are $T_{C_{T1}} = 20 \times 10 = 200$ and $T_{C_{T2}} = 20 \times 8 = 160$. To express the deviation of the historical planned team capacities ($T_{C_{T1,T2}}$) from the historical actual team capacities ($RU\textsubscript{S}_{T1,T2}$) at team 1 and team 2 levels, I also constructed two additional response variables to test C3Q2 – namely $\Delta_{T1} \triangleq |T_{C_{T1}} - RU\textsubscript{S}_{T1}|$ and $\Delta_{T2} \triangleq |T_{C_{T2}} - RU\textsubscript{S}_{T2}|$.

The historical $\Delta_{T1}^{hist}$ and $\Delta_{T2}^{hist}$ values are presented in Table 5.3. The columns $RU\textsubscript{S}_{T1}^{alg}$ and $RU\textsubscript{S}_{T2}^{alg}$ show the deliverable User stories at the end of the releases that were elaborated by the partitioning algorithm (partition numbers were $k = 2$). These values are significantly different from the historical values due to the optimal partitioning of the FRs (cp. Table 5.1). Based on the values of $RU\textsubscript{S}_{T1}^{alg}$ and $RU\textsubscript{S}_{T2}^{alg}$, I also constructed two additional response variables to test C3Q2 – namely $\Delta_{T1}^{alg} \triangleq |T_{C_{T1}} - RU\textsubscript{S}_{T1}^{alg}|$ and $\Delta_{T2}^{alg} \triangleq |T_{C_{T2}} - RU\textsubscript{S}_{T2}^{alg}|$.

<table>
<thead>
<tr>
<th>$\Delta_{T1}^{hist}$</th>
<th>$\Delta_{T2}^{hist}$</th>
<th>$RU\textsubscript{S}_{T1}^{alg}$</th>
<th>$RU\textsubscript{S}_{T2}^{alg}$</th>
<th>$ED_{T1}^{alg}$</th>
<th>$ED_{T2}^{alg}$</th>
<th>$T_{C_{T1}}^{alg}$</th>
<th>$T_{C_{T2}}^{alg}$</th>
<th>$\Delta_{T1}^{alg}$</th>
<th>$\Delta_{T2}^{alg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.5</td>
<td>4.5</td>
<td>273</td>
<td>14</td>
<td>280</td>
<td>7</td>
<td>58</td>
<td>3</td>
<td>60</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>326</td>
<td>16</td>
<td>320</td>
<td>6</td>
<td>53</td>
<td>3</td>
<td>60</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>307</td>
<td>15</td>
<td>300</td>
<td>7</td>
<td>35</td>
<td>2</td>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>27.5</td>
<td>15</td>
<td>290.5</td>
<td>15</td>
<td>300</td>
<td>9.5</td>
<td>112</td>
<td>6</td>
<td>120</td>
<td>8</td>
</tr>
<tr>
<td>19.5</td>
<td>16</td>
<td>272.5</td>
<td>14</td>
<td>280</td>
<td>7.5</td>
<td>52</td>
<td>3</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>24</td>
<td>16</td>
<td>282</td>
<td>14</td>
<td>280</td>
<td>2</td>
<td>38</td>
<td>2</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td>224</td>
<td>11</td>
<td>220</td>
<td>4</td>
<td>162</td>
<td>8</td>
<td>160</td>
<td>2</td>
</tr>
<tr>
<td>$\sum$</td>
<td></td>
<td>116.50</td>
<td>93.50</td>
<td>99.0</td>
<td>43.0</td>
<td>27.0</td>
<td>34.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using the same average velocity of User story development per person as in the historical cases (i.e $20$) we could calculate the planned team capacities ($T_{C_{T1,T2}}^{alg}$) and its deviation from the elaborated capacities ($\Delta_{T1}^{alg} \triangleq |T_{C_{T1}}^{alg} - RU\textsubscript{S}_{T1}^{alg}|$ and $\Delta_{T2}^{alg} \triangleq |T_{C_{T2}}^{alg} - RU\textsubscript{S}_{T2}^{alg}|$). To set $\Delta_{T1,T2}^{alg}$ values against the $\Delta_{T1,T2}^{hist}$ values, I can conclude that the resource workload utilization can be considerably better ($\nabla_{Res} \triangleq (|\Delta_{hist}^{T1} + \Delta_{hist}^{T2}| - (|\Delta_{alg}^{T1} + \Delta_{alg}^{T2}|))/(|\Delta_{hist}^{T1} + \Delta_{hist}^{T2}|) = ((116.5 + 93.5) - (99.0 + 27.0))/(116.5 + 93.5) \approx 40\%$) since the algorithm helped to allocate resources to teams according to their needs instead of allocating them uniformly. As a consequence, the elaborated method could provide lower implementation risk and more economical resource exploitation by decreasing resource over- and underload respectively.
Statistical analysis was also performed on the response variables. Besides the previously detailed average differences ($\nabla CCI$ and $\nabla Res$), the results of the analysis (Table 5.4) also underline the usefulness of the elaborated method. In the worst case, that is defined as $\nabla_{CCI} = \text{Min}(CCI_{hist})/\text{Max}(CCI_{alg}) \approx 12.87$, points out that the communication and coordination can be significantly reduced, and in the best case there is $\nabla_{bc} \triangleq \text{Max}(CCI_{hist})/\text{Min}(CCI_{alg}) = 26.20$ communication and coordination needs between the teams during development. Considering the Min and Max statistics of the $\Delta_{hist,T1,T2}$ and the $\Delta_{alg,T1,T2}$ variables, similar differences can be seen ($\nabla_{Res} \approx 0\%$ and $\nabla_{max,Res} \approx 60\%$ respectively) as in the average (c.f. $\nabla_{Res}$).

### Table 5.4: Comparison of Statistics.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>Std.dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CCI_{hist}$</td>
<td>255.68</td>
<td>256.25</td>
<td>203.70</td>
<td>310.46</td>
<td>35.32</td>
</tr>
<tr>
<td>$CCI_{alg}$</td>
<td>13.58</td>
<td>13.40</td>
<td>11.85</td>
<td>15.83</td>
<td>1.3834</td>
</tr>
<tr>
<td>$\Delta_{hist,T1,T2}$</td>
<td>16.64</td>
<td>19.5</td>
<td>3.0</td>
<td>27.5</td>
<td>9.55</td>
</tr>
<tr>
<td>$\Delta_{alg,T1,T2}$</td>
<td>6.14</td>
<td>7.0</td>
<td>2.0</td>
<td>9.5</td>
<td>2.46</td>
</tr>
</tbody>
</table>

From these, I can conclude that the elaborated method 1) can break down the development work into cohesive feature chunks such a way that the dispersed teams are roughly working on the same set of modules, therefore communications predominately occur within grouped FRs rather than between grouped FRs; 2) necessitates less intensive communication and coordination needs (i.e. $\nabla_{bc}$, $\nabla_{wc}$ and $\nabla_{CCI}$) than the intuitive ones (c.f. C3Q1); can provide better utilization of resources according to their needs instead of allocating them uniformly that can provide lower implementation risk and more economical resource exploitation (i.e. $\nabla_{Res}$, $\nabla_{max,Res}$ and $\nabla_{Res}$) by decreasing resource over- and underload respectively (c.f. C3Q2); 3) can provide higher quality feature distribution plans with the utilization of semi-automatic production of feature chunks, which makes it possible to re-partition the features any time within seconds in order to support what-if analysis or to adapt the plan to the continuously changing situations of agile environments (c.f. C3Q3).

### 5.4.5 Computational Benchmarking

Problem solving time of the historical data set, which can be considered as medium-sized problem, took less than one second with my prototypic tool. Therefore, to give some orientation about the performance and the quality of the generated numerous plans (360 different cases) of my approach on larger problems, I carried out simulations with the guidance of [139].

To get a more nuanced picture of the algorithm I considered several groups of randomly generated instances of release scheduling to reflect special properties that influence the solution process. In all instances the resource demands ($w_j$) were uniformly distributed in the data set of $\{0.5, 1, 2, 3, 5, 8\}$ (reflected the typical resource demands in Story points [67]). To enlarge the problem size, I generated different feature counts $n = \{50, 100, 200, 500\}$ and I chose three different distributed team settings: i) two teams of 20 people (12 and 8 people), ii) three teams of 30 people (12, 10, and 8 people), and iii) four teams of 40 people (12, 10, 10 and 8 people) – all teams’ size were considered typical [67]. I also generated Module Relatedness Variable (MRV) between the different number of features and the 41 system modules – where the relation counts were selected from the interval $[1, 3]$ – analogously to the historical case (c.f. Sec. 5.4.3).
5.4. Experimental Results

5.4.5.1 Solving Time

The algorithms were run 30 times to calculate the mean solving time in the three distributed team settings (2 – 4 teams) and feature counts (i.e. \( n = \{50, 100, 200, 500\} \)). All tests were run on an Intel Pentium 4, 2.2 GHz, 4GB memory, MS Windows 7. The results are presented in Table 5.5, where \( ST \) denotes the algorithm’s solving time (in seconds).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( ST^{(2\text{teams})} )</th>
<th>( ST^{(3\text{teams})} )</th>
<th>( ST^{(4\text{teams})} )</th>
<th>( \nabla_1^{ST} )</th>
<th>( \nabla_2^{ST} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.20</td>
<td>0.25</td>
<td>0.27</td>
<td>0.25</td>
<td>0.08</td>
</tr>
<tr>
<td>100</td>
<td>0.43</td>
<td>0.53</td>
<td>0.57</td>
<td>0.23</td>
<td>0.08</td>
</tr>
<tr>
<td>200</td>
<td>1.24</td>
<td>1.50</td>
<td>1.54</td>
<td>0.21</td>
<td>0.03</td>
</tr>
<tr>
<td>500</td>
<td>8.03</td>
<td>8.18</td>
<td>8.30</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Considering the data within a row in Table 5.5, we can conclude that the team size affects on the solving time slightly: one more team results in between \( \min(\nabla^{ST}) = 1\% \) and \( \max(\nabla^{ST}) = 25\% \) increase, where \( \nabla_i^{ST} \triangleq (ST^{(i+1\text{teams})} - ST^{(i\text{teams})})/ST^{(i\text{teams})} \) and \( i \in 2, 3 \). This result can be explained by the fact that the applied Kernighan-Lin algorithm constraints its running time with an iteration number (as it provides approximate solution – see Section 2.6.4), therefore, the more data requires a slightly more administration time from the algorithm. As a consequence we can observe linear relationship between the team size and the solving time.

Considering the data within a column in Table 5.5, we can observe that the problem size affect on the solving time more substantially as the results show non-linear relationship. This result can be explained by the feature architectural similarity measure calculation (see Section 5.3.1.2): the measure calculates the similarities between features by pair-wise comparison of the adequate Module Relatedness of Variable (MRV) vectors. The calculation requires \( n*(n-1)/2 \) comparisons (\( n \) denotes the number of the features), therefore, we can realize quadratic relationship between the feature size and the solving time, which results \( O(n^2) \) time complexity property of the algorithm.

Although there is quadratic relationship between the feature size and the solving time, delivering 500 features in a release – which practically never occurs – requires cca. 8 seconds that is an acceptable response time. Additionally I have to underline the fact that I implemented my method on the Matlab platform, which executes the code in an interpreted way, therefore, compiled versions (e.g. implemented in C) are expected to be faster with one order of magnitude at least.

5.4.5.2 Quality of Plans

The previous randomly generated instances of release data were analyzed to measure the quality of the release plans. The algorithms were run 30 times to calculate the average. I investigated two cases: i) in the \textit{woa} case – i.e. \textit{without algorithm} in the index – the features were distributed on the teams uniformly using the team capacities but not considering the module relatedness of features; ii) in the \textit{alg} case – i.e. \textit{algorithm} in the index – the features were distributed using the before mentioned feature chunk construction algorithm. These two settings made it possible to measure the quality of the plans.

The result of the simulation is presented in Table 5.6. The first column denotes the different feature counts and the second column shows the different team settings. The definitions of the formulas in other columns are the same as previously – the only difference is that they show averages of the runs (30 runs).
5.4. Experimental Results

<table>
<thead>
<tr>
<th>n</th>
<th>No. teams</th>
<th>CCI (_{\text{woa}})</th>
<th>CCI (_{\text{alg}})</th>
<th>(\nabla_{\text{CCI}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>2</td>
<td>165.23</td>
<td>12.87</td>
<td>12.84</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>186.24</td>
<td>15.09</td>
<td>12.34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>192.16</td>
<td>17.32</td>
<td>11.09</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>355.56</td>
<td>25.24</td>
<td>14.09</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>397.28</td>
<td>29.22</td>
<td>13.60</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>465.50</td>
<td>35.50</td>
<td>13.11</td>
</tr>
<tr>
<td>200</td>
<td>2</td>
<td>1083.10</td>
<td>58.28</td>
<td>18.58</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1257.39</td>
<td>68.56</td>
<td>18.34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1423.23</td>
<td>78.20</td>
<td>18.20</td>
</tr>
<tr>
<td>500</td>
<td>2</td>
<td>2411.12</td>
<td>91.12</td>
<td>26.46</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2753.12</td>
<td>108.43</td>
<td>25.39</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3055.10</td>
<td>121.17</td>
<td>25.21</td>
</tr>
<tr>
<td>(\sum)</td>
<td></td>
<td>13745.03</td>
<td>661.00</td>
<td></td>
</tr>
</tbody>
</table>

Considering the \(\nabla_{\text{CCI}}\) column, one can realize that the greater the feature counts better results of the algorithm (i.e. increasing quotient). This phenomenon can be explained by the fact that greater number of features makes it possible to find better partitioning from the greater number of combinations of configurations. However, if we look at the different team sizes within feature counts, one can easily realize that they show decreasing values. This is a consequence of the increasing difficulty of partitioning: the more partitions the worse partitioning costs due to the inherent property of partitioning.

In the problem of 2 teams and 100 features the \(\text{CCI}_{\text{woa}} = 355.56\). The similar historical case (see Table 5.4) is \(\text{CCI}_{\text{hist}} = 252.83\). This difference can be interpreted with the fact that in the historical cases the features were distributed more rationally \((252.83 < 355.56)\) than the applied uniform distribution in the woa cases. This can be interpreted so as the development coordinator probably considered implicitly the cohesiveness of deliverable features during manual distribution.

If we compare the \(\nabla_{\text{CCI}}\) values of these two cases, we can realise that the historical value was 18.62 and the simulation had a similar value of 14.09. Additionally, the Min and Max statistics of the simulation were \(\nabla_{\text{CCI}} \approx 1.36\%\) and \(\nabla_{\text{CCI}} \approx 237.38\%\). These results point out that the greater the problem the better the CCI value. Therefore, the algorithmic partitioning necessitates less intensive communication and coordination needs (i.e. \(\nabla_{\text{CCI}}\)) than the intuitive ones (c.f. C3Q1) – this fact is congruent with and reinforce the findings of the post mortem validation.

All in all, I can state that the algorithmic approach could outperform the manual approach in terms of release plan quality. Moreover, the goodness of the algorithmic planning is affected by two factors: i) the more feature count the better \(\nabla_{\text{CCI}}\) value (i.e. positively correlates) – in other words it provides even better plan quality; ii) the more teams the less good \(\nabla_{\text{CCI}}\) value (i.e. negatively correlates) – it provides less good plan quality. Although, considering the the feature count factor and the team factor the former has much stronger effect on the \(\nabla_{\text{CCI}}\) value. This observation is particularly important when a developer coordinator configure the release and the team settings.
5.5 Discussion

The significant difference between co-located and distributed approaches is explained by the communication and coordination issues rather than the size or complexity of the cross-site development [57]. The communication and coordination problems result in C3P1-C3P4 (Sec. 5.1.3). In plan-driven situations, these communication and coordination issues (Ch1-Ch3) are diminished with increasing the formality of the interactions using detailed documentations (i.e. specifications, design plans, project plans) and conventions (i.e. coding standards, templates). Although, this approach contradicts to the ideas of agility, and it seems to preclude the use of them [37, 34, 57, 45]. To reduce these issues in agile environments, my approach is to decrease the need of interactions between the teams with identifying so-called feature chunks that can be treated roughly independently.

As a solution, my elaborated Feature Partitioning Method (FPM) (Sec. 5.3) was presented which extends agile release planning to distributed environments. The FPM consists of two successive steps namely Feature Architectural Similarity Analysis (FASA) (C3.1; Sec. 5.3.1) and Feature Chunk Construction (FCC) (C3.2; Sec. 5.3.2). First, in C3.1, I introduced a ImplementedIn relation between requirements (FRs) and software modules (SMs), and constructed a so-called Module Relatedness Variable (MRV) to indicate the fact that FRs are implemented in different set of SMs (Sec. 5.3.1.1). Then, I elaborated an architectural similarity measure (∼j,j′; see 5.3) to characterize architectural similarity of features in terms of occurrence of relations between FRs and SMs. This measure is used to form feature chunks according to the values of the measurements.

Next, in C3.2, I constructed a so-called Feature Architectural Similarity Graph (FASG; Sec. 5.3.2.1) from the features and their architectural similarity relations. To support the parallel development, I introduced the feature chunk construction step which aim was to break down the whole development work into cohesive parts. To determine the proper decomposition of blocks I applied a graph partitioning method on the defined FASG. Actually, finding the optimal solution for the edge cutting minimization problem of graph partitioning (MINIMUM K-CUT problem) for non-trivial cases is known to be NP-complete [158]. A variety of algorithms were developed to solve this graph partitioning problem class [159]. The implemented algorithm is based on the Kernighan-Lin (KL) method [196, 159] – an \( O(n^2 \log n) \) algorithm – which is one of the earliest methods for graph partitioning, and more recent heuristics methods are often its variations. Although, there are more efficient global optimization algorithms to this problem class, I implemented KL method because of its relatively easy implementation and its acceptable results for practical applications. However, evaluation of other algorithms is recommended to provide a sophisticated insight (e.g. in terms of efficiency and quality of solutions) into the consequences of the usage of different algorithms.

An important characteristic of elaborated method is that it is based upon two underlying assumptions: the architecture of the system (i.e. set of modules) is defined and the implementation of the identified features can be appointed to a subset of software modules. In agile approaches, the architecture often evolves organically therefore the cohesion of modules and coupling between modules may decrease and increase respectively [30]. Therefore, the utility of the technique may be impaired over time, unless refactoring occurs. After refactoring, the ImplementedIn relations should also be maintained by the the selected members of the distributed team (typically managers, customer representatives and lead developers) during release planning.
The presented method also assumes flexible team sizes that are derived from the partition sizes. This method gives an opportunity to adjust resource usages to the needs while the communication intensities are minimized. If the team sizes are fixed, then a complementary approach is needed. For example, the number of partitions should be increased and properly selected subsets of them should be allocated to each team. This extension can handle the additional constraint of the latter case. However, this case obviously worsens the effectiveness of the presented method considering the coordination and communication needs.

Considering the data sets, the evaluation showed that arranging development work (FRs) according to the identified feature chunks may significantly decrease the communication needs and coordination complexity of the distributed team. However, as my simulation carried out post mortem analysis, in-depth investigation of the method is recommended in different development situations (e.g. more than two teams, larger time sizes). Complementary agile release and iteration planning methods can be found in [B2] and in [C5] respectively. Those methods fit well to the presented method.

5.6 Conclusions

In this chapter, I presented my novel concepts, models and algorithms for agile iteration planning. The elaborated method necessitates \(14 - 18\times\) less intensive communication and coordination needs and can provide \(40\%\) better utilization of resources comparing to the traditional distributed agile release planning approaches.

I carried out a post mortem analysis on seven real-life representative data sets extracted from a software development company [165] to validate my contribution. I also proved that this construction can express wide range of distributed agile release planning situations, and it provides an optimal composition of feature packages comparing to the present methods.

Additionally, I also carried out simulations on 360 generated representative data sets – by varying parameters of the iteration problem (see Sec. 1.4.4: feature counts (W), team settings (T) and module relatedness (\(\otimes\))) – to get an insight into the performance and quality of the presented approach and to filter out the statistical staggering of different agile iteration planning problems.

The results of experiments on representative real-life data sets and representative generated data sets indicate that my approach can provide practical value as a decision support method for agile distributed release planning.

The summary of the Contribution 3 can be found in the following box:

**C3**: I have elaborated a novel method (namely Feature Partitioning Method) for distributed agile release planning to improve efficiency and effectiveness in release-centered decisions in distributed agile environments. [C4, A1, E13, C3]

**C3.1** I have defined Feature Architectural Similarity Analysis (FASA) analytic method to determine architectural similarities between features (deliverable functional and non-functional requirements) that can be exploited to identify features that are implemented in the similar sets of software modules. [C4, A1, C3]
5.6. Conclusions

C3.2 I have defined a Feature Chunk Construction method – by formulating a Feature Chunk Construction optimization problem (FCCP) – that rules the distribution of development work across sites considering minimization of communication and coordination needs among the dispersed teams. The main novelty of my approach lies in the mathematical precise formulation of the problem. The elaborated problem formulation is solved with edge cutting minimization optimization model (MINIMUM K-CUT) of graph partitioning. With this approach, by arranging development work according to the identified partitions, the communication needs and coordination complexity of the distributed team can be significantly decreased. [C4]
Chapter 6

Postlude: Conclusions

Industrial software development is a highly complex and dynamic process. The success of software organizations depends on the effectiveness and efficiency of development activities where development coordination takes a key role. The coordination usually faces a decision problem which aim is to determine which features should be delivered in the next sequence of releases. This chapter summarizes the initial research objectives, my contributions, validations and draws up additional open questions.

A major problem faced by coordination is determining which features should be in the next version of software. This decision making is manifested by development plans. These give the description about which features to implement in which deliveries of software systems to provide maximal business value.

Agile software development phrase is used to label many recently emerged software development methods. These methods are suggested by experienced practitioners and have had a huge impact on how software is developed worldwide nowadays. Their primary aims are to deliver faster, better, and cheaper solutions both for the developers’ and the customers’ side.

Agile processes offer numerous benefits to organizations, including quicker return on investment, higher product quality, and better customer satisfaction (Sec. 1.1.3). However, they lack a sound methodological support of agile release and iteration planning – contrary to the traditional, plan-based approaches. The previously cited survey [42] points out that the second and the fifth principal factors from the identified 26 ones are iteration and release planning respectively. These critics underline the importance of providing a more established method for agile planning, that lacks of solid theoretical basis currently due to its novelty. Therefore, agile release and iteration planning are important and up-to-date research areas. The aim of this research is to elaborate methodical support for agile release and iteration planning in both co-located and distributed environments.

In this chapter, as a final conclusion, I compare the results presented in the research with the main research objectives (Sec. 1.6.2). Additionally, I summarize the practical applications of the results and outline some future research directions.
6.1 Fulfillment of Research Objectives

The research objectives and the related contributions are presented in the next:

**RO1** *Improve the productivity of agile software development planning* by introducing interactive, semi-automated methods in different phases of the development process.

I have elaborated the Agile Iteration Scheduling (AIS) method for agile iteration planning in C2. The elaborated method significantly improves load balancing of resources ($\approx 4 - 5 \times$), significantly accelerates iteration scheduling production ($> 50\%$), enables more than $10\%$ increase in project execution’s efficiency, and more than $50\%$ growth in planning efficiency comparing to the traditional approaches.

**RO2** *Reduce cognitive complexity of agile software development planning* to resolve complex decision situations easily by formulating mathematical models.

I have constructed the Agile Release Scheduling (ARS), Agile Iteration Scheduling (AIS) and the Feature Chunk Construction (FCC) optimization models and related algorithms for co-located and distributed agile release and iteration planning to accomplish these goals in C1.2, C1.3, C2.2, C2.3, and C3.2. The optimization models formulated as special cases of BINARY MULTIPLE KNAPSACK, RESOURCE-CONSTRAINED PROJECT SCHEDULING and MINIMUM K-CUT problems. For the former two models I have developed the mksched and the lscap algorithms.

**RO3** *Improve the quality of agile software development planning* to provide lower level risks by considering all major planning factors (e.g. dependencies, capacities) in mathematical optimization models.

I have formulated the Agile Release Scheduling (ARS) optimization model for agile release planning in C1.2 and C1.3. This approach shows more smooth and fully padded iterations ($> 80\%$ of the total iterations are optimal), and prevents resource overload comparing to the traditional approaches. Additionally, I have elaborated the Agile Iteration Scheduling (AIS) optimization model for agile iteration planning in C2.2, C2.3 also. This method significantly improves load balancing of resources ($\approx 4 - 5 \times$), and enables more than $10\%$ increase in project execution’s efficiency.

**RO4** *Support decisions in agile software development planning* to tailor the best plan for the specific project context and users’ and/or customers’ feedbacks by altering constraints, capacities and priorities.

I have constructed the the Agile Release Scheduling (ARS), Agile Iteration Scheduling (AIS) and the Feature Chunk Construction (FCC) methods in C1, C2 and C3 to provide optimal plans, semi-automatic plan generation and to support what-if analysis. Therefore, they lead to more informed and more established decisions.

**RO5** *Improve communication and coordination efficiency of distributed agile software development teams* by introducing semi-automated methods.

I have elaborated the Feature Partitioning Method to achieve these objectives in C3. The elaborated method necessitates $\approx 14 - 18 \times$ less intensive communication and coordination needs and can provide $40\%$ better utilization of resources comparing to the traditional distributed agile release planning approaches.

The main contribution of the research proves the design, application and validation of an Integrated Agile Planning Approach (IAPA). The elaborated IAPA supports development coordinators to cope with the
6.1. Fulfillment of Research Objectives

complexity and the dynamics of software development by providing conceptual models, optimization models and algorithms for agile release and iteration planning.

The presented contribution is fundamentally different from the existing – mainly intuitive – methods in its information fusion and its mathematical optimization approach. Integration of managerial and software engineering information, and the provided algorithmic optimization method easily resolves complex decision situations. It also gives the business increased visibility, and it can also provide constantly up-to-date decision supports considering changes necessitated by shifting business priorities.

The contributions have additional important characteristics: the presented optimization models represent more general problem classes of planning and scheduling. Since these models only define the structure of the problems, the scheduleable elements (in these cases: features, tasks) may be completely different entities (e.g. packages, CPU operations) in other contexts. Moreover, the presented problem formulations can be easily refined by constraint relaxation (e.g. removing temporal constraints) or by constraint completion (e.g. adding resource intensity constraints) to provide solution to other similar planning and scheduling problems.

I have used two validation types to check, establish and reinforce the findings of the contributions by analyzing them from multiple perspectives – this approach is known as triangulation. One of the validation types was a real life experiment (small number of real contexts) and the complementary one was a laboratory experiment (large number of artificial contexts) [129].

The real life experiment helped to validate the contributions at a software developer company in real agile software development contexts. Although every agile development process implementation is different, the applied software process at the selected company can be regarded as typical.

On the other hand, the laboratory experiment helped to generalize the findings of the former experiments by investigating representative large number population of generated problems (120, 360 and 480 different cases). In order to model the typical agile development situations I have carefully generated the data sets in terms of problem complexity factors (such as features, team sizes, temporal dependencies) and environmental factors (such as team distribution) that constitute the parameters of the agile planning problems. The values of these factors were used as inputs during the laboratory experiment (simulation).

The Table 6.1, Table 6.2 and Table 6.3 summarizes the identified problems, my elaborated solutions, its validations, results, and the fulfillment of the objectives.
6.1. Fulfillment of Research Objectives

<table>
<thead>
<tr>
<th>Agile release planning</th>
<th>Description</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problems:</strong></td>
<td>The following planning factors are considered informally: <strong>C1P1</strong> resource capacities (resource demands during iterations), <strong>C1P2</strong> priorities (importance of each requirement delivery), <strong>C1P3</strong> dependencies (relations between requirements), <strong>C1P4</strong> staged-delivery (delivery time of iteration timeboxes), and <strong>C1P5</strong> maximal value (to choose the high priority one from different plans)</td>
<td>Ch. 3.1.2</td>
</tr>
<tr>
<td><strong>Solution:</strong></td>
<td>ARSM information model</td>
<td>Ch. 3.2.3</td>
</tr>
<tr>
<td></td>
<td>ARSP optimization model</td>
<td>Ch. 3.2.6</td>
</tr>
<tr>
<td></td>
<td>mksched and produceCCPrec algorithms</td>
<td>Ch. 3.2.2.3,3.2.7</td>
</tr>
<tr>
<td></td>
<td><strong>SERPA</strong>™ web application</td>
<td>Ch. 3.2.3</td>
</tr>
<tr>
<td></td>
<td><strong>ROPAS</strong>™ matlab toolbox</td>
<td>Ch. 3.3.4</td>
</tr>
<tr>
<td><strong>Validation 1:</strong></td>
<td>Post mortem analysis of 7 data sets that were gained from real life software development situations. The following key questions were addressed: <strong>C1Q1</strong>: What are the staffing requirements over time?; <strong>C1Q2</strong>: How many iterations do we need per release?; and <strong>C1Q3</strong>: How buffers for contingencies are allocated?</td>
<td>Ch. 3.3</td>
</tr>
<tr>
<td><strong>Validation 2:</strong></td>
<td>Simulation on numerous (120) representative data sets that were generated – considering different problem ( n = {50, 100, 200, 500} ) sizes with different settings: 1) without dependencies ( (P \cup C) ), 2) without couplings ( (C) ), and 3) with dependencies ( (P \cup C) ) in order to get insight into their effects on the solving time and the quality of the generated plans and to filter out the statistical staggering of different agile release planning problems.</td>
<td>Ch. 3.3.4</td>
</tr>
<tr>
<td><strong>Results:</strong></td>
<td>It results in global optimal plan (&gt; 80% of the total iterations are optimal) that were generated semi-automatically. It handles dependencies, supports what-if analysis, provides more balanced resource workload, prevention of resource overload, lower level risk of delivery slippage – therefore it leads to more informed and more established decisions comparing to the manual approaches.</td>
<td>Ch. 3.4</td>
</tr>
<tr>
<td><strong>Publications:</strong></td>
<td>[E13, B2, G18]</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Summary of Contribution 1
### 6.1. Fulfillment of Research Objectives

#### Table 6.2: Summary of Contribution 2

<table>
<thead>
<tr>
<th>Description</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Agile iteration planning</strong></td>
<td></td>
</tr>
<tr>
<td>Problems</td>
<td></td>
</tr>
<tr>
<td>1: The following planning factors are considered informally: C2P1 precedences (to express temporal precedences between realizations), C2P2 balancing resource workloads (to avoid resources overloading), and C2P3 optimality (to choose the best one from different plans)</td>
<td>Ch. 4.1.2</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
</tr>
<tr>
<td>2: Separation of engineering solutions and planning data are the root of many problems: C2P4 difficult iteration planning (collecting planning data and producing Gantt charts etc.), C2P5 poor decision support (weak tool support etc.), and C2P6 difficult tracing of requirements realization (due to requirements changes etc.). These problems lead to imprecise planning and improper management of requirements, and all of them result in lower quality, increased efforts and additional project risks.</td>
<td>Ch. 4.1.2</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
</tr>
<tr>
<td>3: Separation of engineering solutions and business goals are the root of many problems: C2P7 longer iteration time periods (because of implementation rework), C2P8 more costly realization of the required solution, and C2P9 poor planning support (realizing not the mostly demanded requirements)</td>
<td>Ch. 4.1.2</td>
</tr>
<tr>
<td>Solution</td>
<td></td>
</tr>
<tr>
<td>1: AISM information model C1.1 SERPA™ web application</td>
<td>Ch. 4.2.1</td>
</tr>
<tr>
<td>- AISP optimization model C2.2</td>
<td>Ch. 4.2.3</td>
</tr>
<tr>
<td>- lcap algorithm</td>
<td>Ch. 4.2.4</td>
</tr>
<tr>
<td>- PROPS™ matlab toolbox</td>
<td></td>
</tr>
<tr>
<td>Solution 2</td>
<td></td>
</tr>
<tr>
<td>UML Model extension C2.4 PYTHIA PROJECT PLANNER™ prototype</td>
<td>Ch. 4.3</td>
</tr>
<tr>
<td>Solution 3</td>
<td></td>
</tr>
<tr>
<td>GRL Model extension C2.5 PYTHIA PROJECT PLANNER™ prototype</td>
<td>Ch. 4.4</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
</tr>
<tr>
<td>1: Post mortem analysis of 4 data sets that were gained from real life software development situations. The following key questions were addressed: How does optimization-based iteration scheduling compare with informal one in terms of C2Q1 resource workload over time, C2Q2 quality and C2Q3 feasibility of the plans.</td>
<td>Ch. 4.2.6</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
</tr>
<tr>
<td>2: Simulation on numerous (480) representative data sets that were generated – considering different problem sizes with different, with different team sizes ([5, 10, 15, 20]) and different precedence densities ([5%, 10%, 20%, 40%]) in other to get insight into their effects on the solving time and the quality of the generated plans and to filter out the statistical staggering of different agile iteration planning problems.</td>
<td>Ch. 4.2.7</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
</tr>
<tr>
<td>3: Case study was carried out to evaluate my elaborated method on a real-life software planning situation. The following hypotheses were stated: C2H1: The elaborated method and tool with the formulation of software requirements metrics, planning constraints and objectives as RCPSP enables improved production (i.e. less manual work) in project release planning contrary to the traditional and mainly manual method.; C2H2: The elaborated method supports what-if analysis by different parameterizing of the scheduling algorithm which leads to improved decision support (i.e. choose from several alternatives) in release planning decisions.; C2H3: The elaborated method with the derivation of release plans from requirements specification leads to more informed requirement’s level tracing.;</td>
<td>Ch. 4.3.3</td>
</tr>
<tr>
<td>Validation</td>
<td></td>
</tr>
<tr>
<td>4: Case study was carried out to evaluate my elaborated method on a real-life software planning situation. The following hypotheses were stated: C2H4: Fusioning of requirements metrics and project planning constraints and objectives leads to improved efficiency in project execution in addition to less synchronization and documentation overhead.; C2H5: Derivation of AISP from software requirement specification enables improved production in project planning contrary to the application of the traditional, mainly manual, project planning method.</td>
<td>Ch. 4.4.3</td>
</tr>
<tr>
<td><strong>Results:</strong></td>
<td></td>
</tr>
<tr>
<td>Solution 1 results in semi-automatic plan generation, ≈ 5× more balanced resource workload over time, optimal and lower-risk feasible plan, handles dependencies, supports what-if analysis, resolves complex decision situations easily, ensures higher quality iteration plans – therefore it leads to more informed and more established decisions comparing to the manual approaches. Solution 2 enables more than 50% growth in planning productivity while C2H1-3 hypotheses were accepted. Solution 3 enables more than 10% increase in project execution’s efficiency while C2H4-5 hypotheses were accepted.</td>
<td>Ch. 4.5</td>
</tr>
<tr>
<td><strong>Publications:</strong></td>
<td></td>
</tr>
<tr>
<td>[E14, E12, C5, G18, E11, D7, E10, I25, G16, E9, E8]</td>
<td></td>
</tr>
</tbody>
</table>
### 6.1. Fulfillment of Research Objectives

#### Table 6.3: Summary of Contribution 3

<table>
<thead>
<tr>
<th>Description</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Distributed agile release planning</strong></td>
<td></td>
</tr>
<tr>
<td>Problems: Communication and coordination problems result in <strong>C3P1</strong> reduced team productivity, <strong>C3P2</strong> increased production interval, <strong>C3P3</strong> increased communication cost, and <strong>C3P4</strong> difficult process control across distributed teams.</td>
<td></td>
</tr>
<tr>
<td>Solution: Feature Architectural Similarity Analysis (FASA) <strong>C3.1</strong></td>
<td></td>
</tr>
<tr>
<td>FCCP optimization model <strong>C3.2</strong> PROPAS™ matlab toolbox</td>
<td></td>
</tr>
<tr>
<td>Validation 1: Post mortem analysis of 7 data sets that were gained from real life software development situations. The following key questions were addressed: <strong>How FPM can be compared with the intuitive one in terms of C3Q1 communication and coordination needs, C3Q2 resource workload, and C3Q3 feasibility?</strong></td>
<td></td>
</tr>
<tr>
<td>Validation 2: Simulation on numerous (360) representative data sets that were generated – considering different problem ( n = {50, 100, 200, 500} ) sizes and number of teams (2 – 4) and module relatedness (∗) in other to get insight into their effects on the solving time and the quality of the generated plans and to filter out the statistical staggering of different agile distributed release planning problems.</td>
<td></td>
</tr>
<tr>
<td>Results 1: It 1) can necessitate ( \approx 14 – 18 ) less intensive communications and coordinations; 2) can provide 40% better utilization of resources; and 3) can provide higher quality feature distribution plans by re-partition the features any time within seconds to support what-if analysis or to adapt the plans to the continuously changing situations of agile environments.</td>
<td></td>
</tr>
<tr>
<td>Publications: [G18, C3, C4, A1]</td>
<td></td>
</tr>
</tbody>
</table>
6.2 Utilization of the New Results and Open Problems

The targeted problems were specified, and the applied methodology was worked out in the frame of the Pythia research project [201]. The Integrated Agile Planning Approach (IAPA) methodology was successfully used in several projects at a large international bank. The scope of this project also included the demonstration of the industrial applicability of the theoretical results through three prototypes.

There are several open problems regarding to the research that can serve objectives for the future:

- One of the future research direction can be to extend the elaborated algorithms capabilities to deal with uncertainties. Since effort estimation involves uncertainties it is useful to determine the certainty of the generated plans at different confidence levels. The conceptions of the extension can be found in [F15].
- Additional direction can be an extension of the elaborated algorithms to generate several plans by exploiting automatic what-if-analysis from which the manager can choose one of them that is better than the others.
- Another interesting direction can be the extension of planning process with historical results in other to get more punctual and adequate results to the given development situation. For example, performance, estimation punctuality, defects in development, function creeps can be used as historical data that is used in the effort and time estimation adjustment. The initial ideas can be found in [D7]. These ideas fit well formally into the presented contributions, but currently they are not in a mature stage.
- In this dissertation the definitions of the problems are described as attributed graphs (i.e. attributes on edges and nodes). Although these problems can also be described with description logic (e.g. Web Ontology Language) to provide consistency checking of the problem definition with reasoners.
- The results of the optimizations in my different contributions provide only one solution (if there is any) which is the best one. However, providing some additional ‘almost the best’ solutions may increase more informed decision support for the development coordinator: e.g. an ‘almost the best solution’ with less risk may be better than the best solution with higher risk.

6.3 Epilogue

The goal of the SEMAT (Software Engineering Method and Theory) initiative is to reshape Software Engineering. Initiators of SEMAT believe that current Software Engineering is seriously unsatisfactory, due to lack of solid theoretical basis and existence of a large number of competing methods, practices and fads. The goal of SEMAT is to ‘refound software engineering based on a solid theory, proven principles, and best practices’.

I believe that my contributions fit to this endeavor with providing sound theoretical and methodical support to the agile release and iteration planning. I think that my elaborated methods, models and algorithms are fluent combinations of the present theories, and methods of software engineering. Their success were demonstrated by the applied methods, presented results and the implemented prototypes.

---

1 The development is supported in part by the GVOP grant (GVOP-3.3.3-05/1.-2005-05-0046/3.0) and realized by the cooperation of Multilogic Ltd [165] and OptXware Llc. [197]. Validation of the results are performed at Multilogic Ltd [165], and the data sets were gained from the IT department of Hungarian Post [198], IT department at Hungarian Tax and Financial Control Administration (APEH) [199], Prolan Corp. [200] and from Multilogic Ltd [165].
6.4 List of Publications

Book Chapters


Articles in an Edited Book


Journal Papers Published Abroad


Journal Papers Published in Hungary

6.4. List of Publications

International Conference Papers


Journal Papers in Hungarian


Hungarian Conference Papers


6.4. List of Publications


Publications on Other Fields


Other Works


Chapter 7

References


7.1 Additional References


7.1 Additional References


7.1. Additional References


# Appendix A

## Notations

Notations of agile planning are summarized in the following:

### Miscellany

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>Empty set</td>
</tr>
<tr>
<td>$</td>
<td>X</td>
</tr>
<tr>
<td>$\mathbb{N}$</td>
<td>Set of positive integers</td>
</tr>
<tr>
<td>$\mathbb{R}$</td>
<td>Set of real numbers</td>
</tr>
<tr>
<td>$\lfloor a \rfloor$</td>
<td>Largest integer not greater than $a$</td>
</tr>
<tr>
<td>$\lceil a \rceil$</td>
<td>Smallest integer not less than $a$</td>
</tr>
<tr>
<td>$\mathcal{O}(f(n))$</td>
<td>Order of $f(n)$</td>
</tr>
</tbody>
</table>

### Planning

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>Development coordinator</td>
</tr>
<tr>
<td>$O$</td>
<td>Planning objectives</td>
</tr>
<tr>
<td>$C$</td>
<td>Planning constraints</td>
</tr>
<tr>
<td>$W$</td>
<td>Set of deliverable features</td>
</tr>
<tr>
<td>$\Delta W$</td>
<td>Changes that influence deliverable features</td>
</tr>
<tr>
<td>$W'$</td>
<td>Changed deliverable features</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>Set of selected developers</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Feedback from the customer</td>
</tr>
<tr>
<td>$\mathcal{S}$</td>
<td>Scheduling function</td>
</tr>
</tbody>
</table>

### Agile Release Planning

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W$</td>
<td>Set of deliverable features</td>
</tr>
<tr>
<td>$j$</td>
<td>Deliverable feature index ($j \in W$)</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of deliverable features ($n =</td>
</tr>
<tr>
<td>$p_j$</td>
<td>Priority of a feature ($p_j \in \mathbb{N}$)</td>
</tr>
<tr>
<td>$w_j$</td>
<td>Estimated effort for feature implementation ($w_j \in \mathbb{R}$)</td>
</tr>
<tr>
<td>$\mathcal{D}$</td>
<td>Release dependencies between features ($\mathcal{D} \triangleq P \cup C$)</td>
</tr>
<tr>
<td>$P$</td>
<td>Set of precedences between features</td>
</tr>
</tbody>
</table>
Precedences between features \( P_{j', j} \) \((P_{j', j} \in [0, 1])\)

Set of coupling between features \( C \)

Coupling between features \( C_{j', j} \) \((C_{j', j} \in [0, 1])\)

Set of developers \( \mathcal{R}^* \)

Set of selected developers \( \mathcal{R} \)

Resource index \((i \in \mathcal{R})\)

Number of developers \((m = |\mathcal{R}|)\)

Effectiveness factor of a developer \((e_i \in [0, 1])\)

Effectiveness factor of the team \((e \in \mathbb{R}^+ : e = \sum e_i)\)

Set of iterations \( I \)

Iteration index \((k \in I)\)

Number of iterations \((o = |I|)\)

Length of iteration \((l^I \in \mathbb{N})\)

Capacity of team resource in an iteration \((c_k \in \mathbb{R})\)

Used up iterations when release is scheduled \((y_k)\)

Length of release \((l^R \in \mathbb{N})\)

Velocity of selected resources \((v^R \in \mathbb{R})\)

Assignment matrix \((X_{i, j} \in [0, 1])\)

Delivered value (in a release) \((z)\)

Date-driven release planning problem \(1|\mathcal{P}, \mathcal{C}, c_k|_{\text{max}}(z)\)

Scope-driven release planning problem \(1|\mathcal{P}, \mathcal{C}|_{\text{max}}(z)\)

Release scheduling function \(\mathcal{G}_{AR}: (W, I, R, p, w, e, D, e, \mathcal{P}, \mathcal{C}, l^I, k) \rightarrow X\)

Agile Iteration Planning

Set of technical tasks \(A\)

Technical task index \((j \in A)\)

Number of technical tasks \((n = |A|)\)

Estimated effort for technical task implementation \((w_j \in \mathbb{R})\)

Resource pre-assignment of a technical task \((a_j \in \mathcal{R})\)

Set of precedences between technical tasks \(P_{j', j} \in [0, 1] \)

Set of developers \(\mathcal{R}^* \)

Set of selected developers \(\mathcal{R} \)

Resource index \((i \in \mathcal{R})\)

Number of developers \((m = |\mathcal{R}|)\)

Length of iteration \((l^I \in \mathbb{N})\)

Schedule vector (Start time of technical task implementation) \((S_j \in \mathbb{N})\)

Completion time of technical task implementation \((C_j \in \mathbb{N})\)

Delivered value (in an iteration) \((z)\)

Start time (virtual) of technical task implementation \((S_0 \in \mathbb{N})\)

End time (virtual) of technical task implementation \((C_{\text{max}} = S_{n+1} \in \mathbb{N})\)

Iteration planning problem (date-driven) \(\mathcal{G}_{AI}: (A, \mathcal{R}, a, w, P, \mathcal{I}, l^I) \rightarrow S\)

Distributed extension of Agile Release Planning
$\mathcal{W}$ Set of deliverable features

$j$ Deliverable feature index ($j \in \mathcal{W}$)

$n$ Number of deliverable features ($n = |\mathcal{W}|$)

$w_j$ Estimated effort for feature implementation ($w_j \in \mathbb{R}$)

$\mathcal{M}$ Set of modules

$l$ Module index ($l \in \mathcal{M}$)

$p$ Number of modules ($p = |\mathcal{M}|$)

$\otimes$ Implemented in relation between features and modules ($\otimes = \mathcal{W} \times \mathcal{M}$)

$\mathcal{T}$ Set of distributed teams

$k$ Team index ($k \in \mathcal{T}$)

$m$ Number of teams ($m = |\mathcal{T}|$)

$\mathfrak{A}_F$ $\mathfrak{A}_F : (\mathcal{W}, \mathcal{M}, \mathcal{T}, \otimes) \rightarrow \mathbb{W}^*$ (Feature chunks)

**Dependencies**

$G_P$ Precedence graph (DAG) with node set $\mathcal{W}$ and arc set $P$ ($G_P \triangleq (\mathcal{W}; P)$)

$G_C$ Coupling graph with node set $\mathcal{W}$ and arc set $C$ ($G_C \triangleq (\mathcal{W}; C)$)

$G_D$ Release dependency multi-graph with node set $\mathcal{W}$ and arc set $P, C$ ($G_D \triangleq (\mathcal{W}; P, C)$)

$G_R$ Release graph (DAG) with node set $\mathcal{W}$ and arc set $P'$ ($G_R \triangleq (\mathcal{W}; P')$)
# Appendix B

## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAPA</td>
<td>Integrated Agile Planning Approach</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process method</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming method</td>
</tr>
<tr>
<td>IDP</td>
<td>Iterative Development Process</td>
</tr>
<tr>
<td>FDD</td>
<td>Feature Driven Development method</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>SEMAT</td>
<td>Software Engineering Method and Theory</td>
</tr>
<tr>
<td>IP</td>
<td>INTEGER PROGRAMMING</td>
</tr>
<tr>
<td>LP</td>
<td>LINEAR PROGRAMMING</td>
</tr>
<tr>
<td>ILP</td>
<td>INTEGER LINEAR PROGRAMMING</td>
</tr>
<tr>
<td>BIP</td>
<td>BINARY INTEGER PROGRAMMING</td>
</tr>
<tr>
<td>MIP</td>
<td>Mixed INTEGER PROGRAMMING</td>
</tr>
<tr>
<td>KP</td>
<td>KNAPSACK Problem</td>
</tr>
<tr>
<td>BKP</td>
<td>BINARY KNAPSACK Problem (0-1 KNAPSACK Problem)</td>
</tr>
<tr>
<td>SSP</td>
<td>SUBSET-SUM Problem</td>
</tr>
<tr>
<td>BMKP</td>
<td>BINARY MULTIPLE KNAPSACK Problem</td>
</tr>
<tr>
<td>BPP</td>
<td>BINARY BINPACKING Problem (0-1 Binpacking Problem)</td>
</tr>
<tr>
<td>MSSP</td>
<td>MULTIPLE SUBSET-SUM Problem</td>
</tr>
<tr>
<td>SPO</td>
<td>Strict Partial Order</td>
</tr>
<tr>
<td>TCPSP</td>
<td>TIME-CONSTRAINED PROJECT SCHEDULING Problem</td>
</tr>
<tr>
<td>RCPSP</td>
<td>RESOURCE-CONSTRAINED PROJECT SCHEDULING Problem</td>
</tr>
<tr>
<td>SPT</td>
<td>Shortest Processing Time first</td>
</tr>
<tr>
<td>WSPT</td>
<td>Weighted Shortest Processing Time first</td>
</tr>
<tr>
<td>EDD</td>
<td>Earliest Due Date first</td>
</tr>
<tr>
<td>AoN</td>
<td>Activity-on-Node network</td>
</tr>
<tr>
<td>AoA</td>
<td>Activity-on-Arc network</td>
</tr>
<tr>
<td>ARIS</td>
<td>Agile Release and Iteration Scheduling</td>
</tr>
<tr>
<td>ARISM</td>
<td>Agile Release and Iteration Scheduling Model</td>
</tr>
<tr>
<td>ARS</td>
<td>Agile Release Scheduling</td>
</tr>
<tr>
<td>ARSM</td>
<td>Agile Release Scheduling Model</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>ARSP</td>
<td>Agile Release Scheduling Problem</td>
</tr>
<tr>
<td>EDA</td>
<td>Exploratory Data Analysis</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>RE</td>
<td>Requirements Engineering</td>
</tr>
<tr>
<td>PM</td>
<td>Project Management</td>
</tr>
<tr>
<td>NPV</td>
<td>Net Present Value</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>GRL</td>
<td>Goal-oriented Requirement Language</td>
</tr>
<tr>
<td>COCOMO</td>
<td>Constructive Cost Model</td>
</tr>
<tr>
<td>FPA</td>
<td>Function Point Analysis</td>
</tr>
<tr>
<td>UCP</td>
<td>Use Case Points</td>
</tr>
<tr>
<td>UCPM</td>
<td>Use Case Point Method</td>
</tr>
<tr>
<td>AIS</td>
<td>Agile Iteration Scheduling</td>
</tr>
<tr>
<td>AISM</td>
<td>Agile Iteration Scheduling Model</td>
</tr>
<tr>
<td>AISP</td>
<td>Agile Iteration Scheduling Problem</td>
</tr>
<tr>
<td>SLOC</td>
<td>Source Lines of Code</td>
</tr>
<tr>
<td>AF</td>
<td>Assigned task First</td>
</tr>
<tr>
<td>KSLOC</td>
<td>Kilo Source Lines of Code</td>
</tr>
<tr>
<td>COQUALMO</td>
<td>Constructive Quality Model</td>
</tr>
<tr>
<td>FP</td>
<td>Function Point</td>
</tr>
<tr>
<td>DEARMS</td>
<td>Distributed Extension of Agile Release Scheduling Model</td>
</tr>
<tr>
<td>DSD</td>
<td>Distributed Software Development</td>
</tr>
<tr>
<td>GSD</td>
<td>Global Software Development</td>
</tr>
<tr>
<td>SM</td>
<td>Software Module</td>
</tr>
<tr>
<td>FR</td>
<td>Feature</td>
</tr>
<tr>
<td>FR</td>
<td>Feature</td>
</tr>
<tr>
<td>FASA</td>
<td>Feature Architectural Similarity Analysis</td>
</tr>
<tr>
<td>MRV</td>
<td>Module Relatedness Variable</td>
</tr>
<tr>
<td>FASG</td>
<td>Feature Architectural Similarity Graph</td>
</tr>
<tr>
<td>FCC</td>
<td>Feature Chunk Construction</td>
</tr>
<tr>
<td>FCCP</td>
<td>Feature Chunk Construction Problem</td>
</tr>
<tr>
<td>FCCM</td>
<td>Feature Chunk Construction Model</td>
</tr>
</tbody>
</table>
Appendix C

Algorithms

This chapter lists all of the Matlab algorithms that are presented in different chapters.

C.1 Algorithm produceCCPrec

Matlab implementation of the produceCCPrec algorithm that can be found in Chapter 3.2.2.3.

```
function [sets,cnode,Cprec] = produceccprec(Copl,Prec)
% PRODUCECCPREC produce precedence matrix for the connected components (CC) of graphs on
% NON-DIRECTED SCG (based on set unioning)
%
% Syntax:
% [sets,cnode,Cprec] = produceccprec(Copl,Prec)
% Input params:
% Copl - coupling matrix between elements
% Prec - precedence matrix between elements
% Return values:
% sets - sets of component nodes
% cnode - forefather node list (root elements of the components)
% Cprec - precedence matrix between CC's elements
%
% Reference:
% -
% Remark: 1) it works on NON-DIRECTED G(V,E) single connected graph
% 2) index of 'sets' determines the index of the Cprec graph!
%
% Complexity: -
% Space : -

Comment on the operation

In graph theory, a connected component of an undirected graph is a
subgraph in which any two vertices are connected to each other by paths,
and to which no more vertices or edges can be added while preserving its
connectivity.

An alternative way to define connected components involves the
equivalence classes of an equivalence relation that is defined on the vertices
of the graph. In an undirected graph, a vertex v is reachable from a vertex u
if there is a path from u to v. In this definition, a single vertex is counted
as a path of length zero, and the same vertex may occur more than once within
```
C.1. Algorithm produceCCPrec

% a path. Reachability is an equivalence relation, since transitive, % reflexive, and symmetric.

%=======================================================================
% Author: Akos Szoke (aszoke@mit.bme.hu)
% Example: -
% See also: isDirected, isDAG, findCC
% Copyright 2006-2009
% -- input checking --
[b, str] = isValidAdj(Copl);
if ~b
    error(str);
end;
if ~isNonDirected(Copl)
    error('Wrong input parameter: Not a non-directed graph!');
end;
% precedence checking
if {size(Prec,1) ~= size(Copl,1)} || {size(Prec,2) ~= size(Copl,2)}
    error('Size of precedence matrix is not consistent with graph!');
end;
if ~isDAG(Prec)
    error('Precedence matrix is not a simple (loop free, single connected) DAG!');
end;
% -- function body --
% since 'Copl' is non-directed, the upper triangular matrix minus diagonal is enough
ndadj = triu(Copl,1); % upper triangular part of matrix /non-directed Copl/
% create singleton sets
sets = cell(1,size(ndadj,1));
for i = 1:size(ndadj,1)
    sets{i} = i;
end;
% find list of edges
[rows,cols] = find(ndadj);
% going through the edges according to the rows
for i = 1:length(rows)
    ndxA = 0; ndxB = 0;
    j = 1;
    while (j <= length(sets)) ...
        if ismember(rows(i),sets{j})
            ndxA = j;
        end;
        if ismember(cols(i),sets{j})
            ndxB = j;
        end;
        j = j+1;
    end;
    if ~ismember(sets{ndxA},sets{ndxB}) % if the two item is not in the same set
        u = union(sets{ndxA},sets{ndxB});
        sets(ndxA) = [u];
        sets(ndxB) = [];
    end;
end;
C.2 Algorithm \textit{mksched}

\begin{verbatim}
\% reveal component nodes (selected as the smallest elements in the component)
cnode = zeros(1,length(sets));
for i = 1:length(sets)
cnode(i) = min(sets{i});
end;

Cprec = zeros(length(sets),length(sets)); \% initialize component precedence matrix
\% fine list of precedence edges
[rows,cols] = find(Prec);
\% going through the edges according to the rows
for i = 1:length(rows)
  ndxA = 0; ndxB = 0;
  j = 1;
  while (j <= length(sets)) ... \% finding sets containing the actual row and column AND
    if ismember(rows(i),sets{j}) \% we not found the indexes of the items
      ndxA = j;
      \% item rows(i) /from node/ is index jth set of the sets
      end;
    if ismember(cols(i),sets{j}) \% item cols(i) /to node/ is index jth set of the sets
      ndxB = j;
    end;
  j = j+1;
end;
\% to get a valid precedence graph, it needs the following properties:
\% - single connected: merge can produce multiple connection -> so we
\% - loop free: merge can produce loops -> so we can deal with it
\% - not DAG graphs: merge cannot produce it
if (ndxA \not= ndxB) \% it is not a loop
  Cprec(ndxA,ndxB)=1;
end;
\end{verbatim}

--- EOF ---

\begin{verbatim}
\% Algorithm mksched
\function [X,z,NDX,exitflag,itc] = mksched(prf,wgt,PRC,c,opt)
\end{verbatim}

--- EOF ---

\begin{verbatim}
\% MKSCHED multi 0-1 knapsack branch and bound algorithm upgraded with
\% priority and SOFT precedence (based on Horowitz-Sahni /1974/)
\% It assumes integrality of prf and wgt!
\% Syntax:
\% [X,z,NDX,exitflag,itc] = mksched(prf,wgt,PRC,c,opt)
\% Input params:
\% prf - profit of elements (item count is N)
\% wgt - weight of elements
\% c - capacity of the knapsacks (knapsack count is M)
\% PRC - precedence matrix between elements
\% Return values:
\% X - unknown /0,1/ - assignment matrix.
\% X(i,j)= 1 if wgt(i) is assigned to knapsack(j)
\% z - objective (profit)
\% NDX - index of selected items
\% exitflag - for identifying the reason the algorithm terminated
\% 0: terminated normally
\% 1: terminated because it is reached MaxIter
\% itc - iteration count
\% Complexity: Ordo(M*2^N) /plus Ordo(N*logN) for the initial sorting/
\% Space : Ordo(N)
\% ========================================================================
\% Comment on the operation
\end{verbatim}

170
C.2. Algorithm mksched

% Assumption: items are in decreasing order according to prf(i)/wgt(i).
% In the first step, we start to fill the initial solution with 1s from
% left to right while we exceed the capacity limit. After this, we select
% that item which exceed the limit and we set it to 0. Then we continue the
% filling with 1s -- until we reach the initial best solution. (So we decide
% whether a given item is in the knapsack or not.)
% After computing the initial best solution, we set the last (from right
% the first one) 1 to 0 (we prune a branch) and we try to find a better
% solution. (In this situation we use estimators to decide if a
% branch may contain a better solution or not.) If we find a better solution
% than the initial one we store it as the actual best solution. Then we
% continue this steps.
% We find the best solution if we cannot do backtrack. So it provides
% global optimum.

% (The binary tree represents the fact that we select or not select an item
% into a solution: so the tree branch according to this decision.)
% In this algorithm the precedence constraint is SOFT: if we scheduled the
% set of rlist element, and there is a room in the given iteration (the remained
% capacity is greater than the least packable item) we don't increment the
% iteration count and we try to put the next set of rlist items to the
% actual iteration's remained capacity (i.e. remained capacity :=
% (current residual capacity) - (sum weight of scheduled items))

% Reference:
(szoke:csm10)
[Martello:knp90]
Martello and Toth - Knapsack problems: algorithms and computer implementations,
John Wiley & Sons, Inc., 1990,
Chapter 2, 0-1 Knapsack problem
Author: Akos Szoke (aszoke@mit.bme.hu)
Example: BKPB1
See also: -
Copyright 2006-2009

/* input checking */
chkschedinput(prf,wgt,PRC,c);

/* function body */
itr = 0;  % iteration counter
while(˜isequal(slist,ones(1,length(prf))) ... % there is no schedulable item
&& idx <= length(c)) % there is no more knapsack

/* ordering jobs according to prf/wgt */
[b,ix] = sort(prf./wgt,'descend');
prf = prf(ix);
wgt = wgt(ix);

/* -- initialization */
if = [];
slist = zeros(1,length(prf));  % 'slist' means 'scheduled list'
X = zeros(length(prf),length(c));  % assignment matrix: column=knapsack, row=item
idx = 1;  % counter of knapsacks
remcap = false;  % is there remained capacity?
while(˜isequal(slist,ones(1,length(prf))) ... % there is no schedulable item
&& idx <= length(c)) % there is no more knapsack
C.2. Algorithm mksched

% -- actualizing the 'ready list'
% find potentially schedulable items according to scheduled items
pot = find(slist == 0); % select potential (not scheduled) items
precrows = PRC(pot,:); % selecting potential rows of precedence matrix
sumprecrows = sum(precrows,2); % find not preceded items
rlist = pot(npi);

if isempty(rlist)
    str = strcat('Infeasible problem! There is no schedulable item!');
    error(str);
end;

prn = prf(rlist); % select not scheduled priorities
wn = wgt(rlist); % select not scheduled weights
n = length(rlist);
prn(n+1) = 0; % technical profit value
wn(n+1) = inf; % technical weight value

z = 0; % value of best solution so far
zn = 0; % current objective

% capacity determination (for SOFT precedence)
if remcap
    cn = cnrem; % ... remained capacity
else
    cn = c(idx);% ... the capacity of the knapsack
end

xn = []; % current solution
j = 1; % search index (!): from this point we are finding a better solution part
backtrack = false; % backtrack is necessary
rlistopt = false; % knapsack rlistopt is reached

while(˜rlistopt)
% it iterates from j=1 to n to calculate a solution (xn(1:n))
while (j <= n) ... % not all items are selected/unselected to xn
    % (backtrack == false) % no need to backtrack (a better solution is possible)
    % ------- Administrative block - Start
    itc = itc + 1; % increment iteration counter
    if opt.Verbose == 1 % dispay iteration info
        fprintf('%.d',itc);
    end;
    if opt.MaxIter == itc % reached the maximum iteration
        exitflag = 1;
        X = [];
        NDX = [];
        return;
    end;
    % ------- Administrative block - End

    % computing upperbound from j to n (if j<n)
    if j < n
        % ------- Administrative block - Start
        % -- compute upper bound U1 for the current solution starting from j
        % This BOUND-ing step helps to upperbound the solution which
        % can be (POSSIBLE!) reached at the given branch. If it cannot
        % provide better solution we have to backtrack (alter the branch).
        % NOTE: We can apply any other upperbound estimation method! (e.g. U1-U7)
        % ------- Administrative block - End

    end;

end;
C.2. Algorithm mksched

wagg = []; % aggregated weights
for k=2:n-j+1
    wagg(k) = wagg(k-1) + wn(j+k-1); %#ok<AGROW>
end;

% find the first critical item computed from start position j
r = min(find(wagg > cn)); %#ok<MXFND>
if isempty(r) % there is no critical item
    r = length(wagg)+1;
end;

% U1(j): upperbound of the list from j to (r-1)
u = sum(prn(j:r-1)) + floor((cn-sum(wn(j:r-1)))/wn(r));
else % upperbound cannot be computed, since it’s started from j
    u = floor(cn*prn(j)/wn(j));
end;

% checking whether (best solution) < (current solution + U1(j))
% so if the sum of zn (x[1...j-1]) + U1(j) is greater then the present best
% solution --> better solution can be achieved
if (z < (zn + u))
    % ------------------------------------------------------------------
    % -- perform a forward step: inserting the largest possible set of
    % new consecutive items into the current solution from position j
    % ------------------------------------------------------------------
    % iterates while (weight(j)) <= (current residual capacity) and
    % include items into the knapsack (x(j):=1) until they are fitted
    % (since wn(n+1)=inf --> it will stop at the end of weights/while
    % while wn(j) <= cn
    cn = cn - wn(j); % decrease the current remained capacity
    zn = zn + prn(j); % increase the current objective
    xn(j) = 1; %#ok<AGROW> % select into the knapsack
    j = j + 1; % increase the start position of the search index
end;

% exclude item j (if j<=n) from the solution (set to 0) since as
% consecutive item it exceeded the remained capacity
if j <= n
    xn(j) = 0; %#ok<AGROW> % remove the last item
    j = j + 1; % increase the start position of the search index
end;

% NOTE: in this point the solution takes the following form sol=[111...10]
% and j point out to the length(1st part of the solution)+1 position
else % best solution is better than the current possible solution in this branch
    backtrack = true;
end;

% in this point it tries (with the while cycle) to fill the remained capacity
% (cn) with values 0 or 1 while the solution is fully determined (j = n)
end; % END OF FILLING (FORWARD) WHILE

% update the best solution
% In this branch we reached a better solution than the best
% solution so far therefore let the current solution is the
% best solution. (Anyway, it occurs in case of backtrack==false case
% and in this case (j > n) is hold also)
if zn > z
    z = zn;
    x = xn;
end;
C.2. Algorithm mksched

% we have found the optimal solution to the 0-1 knapsack problem

% indices of assignments in the current solution
xidx = find(xn);

if isempty(xidx) % backtrack is not possible

    rlistopt = true;
    % we reached the rlistopt

    X(rlist(find(x)),idx) = 1; % storing the solution
    slist(rlist(find(x))) = 1; % set scheduled items

% optimum determination for SOFT precedence
% if the remain gap is less than the least ready item
if ~isempty(find(slist == 0)) %ok<EFIND>

    minnsi = min(wgt(find(slist == 0)));
    % minimal not scheduled item

    cnrem = cn - sum(wgt(rlist(find(x))));
    % remaining capacity := (current residual capacity) - (sum weight of scheduled items)

    if cnrem < minnsi %ok<FNDSB>
        idx = idx + 1; % increment index of knapsack
        remcap = false;
    else
        remcap = true;
    end
else
    idx = idx + 1; % increment index of knapsack
end

% deleting scheduled item from 'PRC', since it doesn't restrict the
% proceeded element(s) (if there is any proceeded)
PRC(:,rlist(find(x))) = 0; %ok<FNDSB>

% find the last inserted item in the current solution
i = max(xidx);
%ok<MXFND>
end;

% this step expresses BRANCH-ing facility: change an item selection (from
% inclusion to exclusion) therefore alter the branch of the tree where
% we are searching the best solution.
if (~rlistopt) && (backtrack == true) || (j > n) % best solution is better than the current

    cn = cn + wn(i); % increase residual capacity
    zn = zn - prn(i); % decrease profit
    xn(i) = 0; %ok<AGROW> remove last inserted item from
    % the current solution

    j = i + 1; % set the search index after the removed item
    backtrack = false;
end;
end; % END OF A KNAPSACK WHILE
end; % END OF KNAPSACKS WHILE

% recover the initial order
[b,ix2] = sort(ix);
X = X(ix2,:);
prf = prf(ix2);

% number of bins used
binned = size(find(sum(X)>0),2);

% determining the lengths
NDX = cell(1,2*binned);
for i=1:binned
    NDX(i) = {find(X(:,i))};
end;
Matlab implementation of the mksched algorithm that can be found in Chapter 3.2.7.2.

C.3 Algorithm lscap

Matlab implementation of the lcap algorithm that can be found in Chapter 4.2.4.

```matlab
function [sl cmax reg] = lscap(p,m,ass,prec,schedstrategy,cap)
%LSCAP list scheduling with different strategies algorithm for the P|prec|Cmax problem
%
% Idea: list scheduling (LS) algorithm is a generic greedy algorithm: whenever
% a machine becomes available, process any unprocessed job
% (In contrast to P|pmtn|Cmax, P|prec|Cmax is NP-hard)
% CMAX: is the lower bound approximation, namely:
% 1) CMAX >= sum(pj/m)
% 2) CMAX >= pj for all jobs
% (i.e. Cmax <= 2*CMAX)
% Syntax:
% [sl cmax] = lscap(p,m,prec,schedstrategy,cap)
% Input params:
% p - processing times
% m - machine number
% ass - resource assignment to task
% prec - precedence constraints (matrix)
% schedstrategy - heuristics for the schedule (since it is an approx. algorithm)
% cap - capacity (iteration velocity)
% Return values:
% sl - schedule
% cmax - Cmax
% reg - is regulated
% (if there is at least one which can not be fitted into the 'cap')
%
% Complexity:
% without any heuristic: : Ordo(n + m)
% Space :
% Reference:
% Article: David Karger et. al. Scheduling Algorithms,
% Author: Akos Szoke (aszoke@mit.bme.hu)
% Example: -
% See also: -
% Copyright 2006-2008
% -- input checking --
% precedence checking
if (size(prec,1) ~= length(p) || (size(prec,2) ~= length(p))
    error('Size of precedence matrix is not consistent with processing time vector!');
end;
end;
end
end;
end
end
% --- EOF ---
```
C.3. Algorithm lscap

if ~isDAG(prec)
    error('Precedence matrix is not a simple (loop free, single connected) DAG!');
end;

if max(ass) > m
    error('Assignment vector refers to a does not exist resource!');
end;

% -- function body --

rlist = []; % 'rlist' means 'ready list'
slist = []; % 'slist' means 'scheduled list'

% list scheduling

sl = zeros(m,length(p)); % machine x jobs
for j=1:length(p) % number of jobs

    % -- actualizing the 'ready list'
    % find potentially schedulable items according to updated 'prec'
    % sum(prec): summing columns -> where it equals to '0' it doesn't have predecssor
    pot = find(sum(prec,2) == 0);
    % updating 'rs': {potential items} - {scheduled items}
    rlist = setdiff(pot,slist);

    if isempty(rlist)
        str = strcat('Infeasible problem! There is no schedulable item at step: ',int2str(j));
        error(str);
    end;

    switch upper(schedstrategy)
        case 'NONE'
            jobndx = rlist(1); % select the first element of rlist to schedule
        case 'AF' % schedule assigned jobs first!
            [val,i] = max(ass(rlist));
            jobndx = rlist(i);
        case 'LPT'
            [val,i] = max(p(rlist));
            jobndx = rlist(i);
        case 'SPT'
            [val,i] = min(p(rlist));
            jobndx = rlist(i);
        otherwise
            error('Unknown strategy!');
    end;

    % assign job to resource according to pre-assignment
    if ass(jobndx) == 0 % there is no assignment
        % -- find the minimally loaded machine
        [minmach,machndx] = min(sum(sl,2));

        % -- applying BEST FIT strategy
        % calculate residuals
        residual = cap-(sum(sl,2))-p(jobndx);
        % returns indices that has greater or equal residual than zero
        idx0 = find(residual - p(jobndx) >= 0);
        % returns a bin to where it is best fitted and has the
        % lowest index (BEST FIT)
        [val,idx1]=min(residual(idx0));
        % return the machine index of possible bin that is best fitted
        machndx = idx0(idx1);
        mnmach = sum(sl(machndx,:));
    else
        machndx = ass(jobndx);
        mnmach = sum(sl(machndx,:));
    end;

    posndx = min(find(sl(machndx,:) == 0)); % find the min unscheduled position
if (minmach + p(jobndx)) < cap
    sl(machndx,posndx) = p(jobndx); % schedule on machine on position
    reg = true; % regulated (exceeds capacity)
    strcat('Not scheduled job: ',num2str(jobndx))
end;

slist = vertcat(slist,jobndx); % update ‘scheduled list’

% deleting scheduled item from ‘prec’, since it doesn’t restrict the
% preceded element(s) (if there is any preceded)
prec(:,jobndx) = 0;

% calculating makespan (Cmax)
cmax = max(sum(sl,2));

end
Appendix D

Models for Agile Software Development

D.1 UML Profile Pythia

```xml
<?xml version="1.0" encoding="UTF-8"?>
<eAnnotations xmi:id="_MUbLvmtfEduJuOn8sTjZ1Q" source="http://www.eclipse.org/uml2/2.0.0/UML" />
<contents xmi:type="ecore:EPackage" xmi:id="_Y-mwYH7IEdu3kKPRQdQpTg" name="Pythia_profile" nsURI="http://schemas/Pythia_profile/0" nsPrefix="Pythia_profile" />
<eClassifiers xmi:type="ecore:EClass" xmi:id="_Y-mwYn7IEdu3kKPRQdQpTg" name="PythiaActor"/>
<eStructuralFeatures xmi:type="ecore:EAttribute" xmi:id="_Y-mwZX7IEdu3kKPRQdQpTg" name="ACW" ordered="false" lowerBound="1" />
<eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwZH7IEdu3kKPRQdQpTg" name="base_Actor" ordered="false" lowerBound="1" />
<eStructuralFeatures xmi:type="ecore:EAttribute" xmi:id="_Y-mwY37IEdu3kKPRQdQpTg" name="Risk" ordered="false" lowerBound="1" />
<eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwZ37IEdu3kKPRQdQpTg" name="Resource" ordered="false" lowerBound="1" />
<eStructuralFeatures xmi:type="ecore:EClass" xmi:id="_Y-mwaH7IEdu3kKPRQdQpTg" name="PythiaUseCase"/>
<eClassifiers xmi:type="ecore:EClass" xmi:id="_Y-mwaX7IEdu3kKPRQdQpTg" name="PythiaProfile"/>
<eStructuralFeatures xmi:type="ecore:EClass" xmi:id="_Y-mwaJ7IEdu3kKPRQdQpTg" name="PythiaProfile"/>
<eStructuralFeatures xmi:type="ecore:EClass" xmi:id="_Y-mwaK7IEdu3kKPRQdQpTg" name="PythiaProfile"/>
<eStructuralFeatures xmi:type="ecore:EClass" xmi:id="_Y-mwaL7IEdu3kKPRQdQpTg" name="PythiaProfile"/>
```

178
D.1. UML Profile Pythia

```xml
<eClassifiers xmi:type="ecore:EClass" xmi:id="_Y-mwcH7IEdu3kKPRQdQpTg" name="PythiaResource">
  <eAnnotations xmi:id="_Y-mwcX7IEdu3kKPRQdQpTg" source="http://www.eclipse.org/uml2/2.0.0/UML" references="_XT6goGt2Edus0NRhhaH65A"/>
  <eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwc37IEdu3kKPRQdQpTg" name="Resource" ordered="false" lowerBound="1">
    <eType xmi:type="ecore:EClass" href="http://www.eclipse.org/uml2/2.0.0/UML#//Class"/>
  </eStructuralFeatures>
</eClassifiers>

<eClassifiers xmi:type="ecore:EClass" xmi:id="_Y-mwcX7IEdu3kKPRQdQpTg" name="ExtendedClass">
  <eAnnotations xmi:id="_Y-mwen7IEdu3kKPRQdQpTg" source="http://www.eclipse.org/uml2/2.0.0/UML" references="_3KPVsHugEdu7zbKFvh7W0g"/>
  <eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwe37IEdu3kKPRQdQpTg" name="base_Class" ordered="false" lowerBound="1">
    <eType xmi:type="ecore:EClass" href="http://www.eclipse.org/uml2/2.0.0/UML#//Class"/>
  </eStructuralFeatures>
  <eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwfH7IEdu3kKPRQdQpTg" name="Resource" ordered="false" lowerBound="1">
    <eType xmi:type="ecore:EClass" href="http://www.eclipse.org/uml2/2.0.0/UML#//Class"/>
  </eStructuralFeatures>
</eClassifiers>

<eClassifiers xmi:type="ecore:EClass" xmi:id="_Y-mweX7IEdu3kKPRQdQpTg" name="EntityClass">
  <eAnnotations xmi:id="_Y-mweH7IEdu3kKPRQdQpTg" source="http://www.eclipse.org/uml2/2.0.0/UML" references="_QOCm0XugEdu7zbKFvh7W0g"/>
  <eSuperTypes xmi:type="ecore:EClass" xmi:id="_Y-mw37IEdu3kKPRQdQpTg" name="EntityClass"/>
  <eAnnotations xmi:id="_Y-mw7Edu3kKPRQdQpTg" source="http://www.eclipse.org/uml2/2.0.0/UML" references="_2vZDmXugEdusOBKvf7W0g"/>
  <eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwe37IEdu3kKPRQdQpTg" name="base_Class" ordered="false" lowerBound="1">
    <eType xmi:type="ecore:EClass" href="http://www.eclipse.org/uml2/2.0.0/UML#//Class"/>
  </eStructuralFeatures>
  <eStructuralFeatures xmi:type="ecore:EReference" xmi:id="_Y-mwfH7IEdu3kKPRQdQpTg" name="Resource" ordered="false" lowerBound="1">
    <eType xmi:type="ecore:EClass" href="http://www.eclipse.org/uml2/2.0.0/UML#//Class"/>
  </eStructuralFeatures>
</eClassifiers>
```
D.2. Ontology of Integrated Agile Software Planning and Scheduling

Prefix: : <http://www.kese.hu/ontologies/agileswplanning.owl#>
Prefix: xsd: <http://www.w3.org/2001/XMLSchema#>
Prefix: owl: <http://www.w3.org/2002/07/owl#>
Prefix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Prefix: dc: <http://purl.org/dc/elements/1.1/>
Prefix: rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Prefix: skos: <http://www.w3.org/2004/02/skos/core#>

Ontology: <http://www.kese.hu/ontologies/agileswplanning.owl>
AnnotationProperty: dc:description
Datatype: rdf:PlainLiteral
Data type: xsd:int
Data type: xsd:float
ObjectProperty: resourceRelation
SubPropertyOf:
relationalProperty
ObjectProperty: isAssignedTo
Annotations:

dc:description "isAssignedTo signifies that a set of \texttt{FeaturePackage}s are assigned to a given \texttt{Iteration} ($\mathcal{X}$). This assignment is the result of the release scheduling ($\mathfrak{S}_{AR}$)."
D.2. Ontology of Integrated Agile Software Planning and Scheduling

SubPropertyOf: dependency
Characteristics: Asymmetric, Irreflexive
ObjectProperty: hasSubStage
Annotations: dc:description "hasSubStage denotes parent-child relationship between different kinds of \ emphasit (DeliveryStage)s."
SubPropertyOf: deliveryStageRelation
Characteristics: Asymmetric, Irreflexive, InverseFunctional
ObjectProperty: hasMember
Annotations: dc:description "hasMember denotes parent-child relationship between \ emphasit (Team) and \ emphasit (Developer)s."
SubPropertyOf: resourceRelation
Characteristics: Asymmetric, Irreflexive, InverseFunctional
ObjectProperty: hasModule
Annotations: dc:description "hasModule denotes parent-child relationship between \ emphasit (Software) and \ emphasit (SoftwareModule)s."
SubPropertyOf: workProductRelation
DisjointWith: hasPart
Characteristics: Asymmetric, Irreflexive, InverseFunctional
ObjectProperty: hasAllocationOf
Annotations: dc:description "hasAllocationOf means that the given \ emphasit (TechnicalTask) is allocated for a \ emphasit (Developer) (S\bf{f}S). This allocation is the result of the iteration scheduling (S\mathfrak{S}_{AI})$. (It is the inverse relation of the usual resource allocation to work products.)"
SubPropertyOf: dependency
Characteristics: Symmetric, Reflexive
ObjectProperty: isCoupledWith
Annotations: dc:description "isCoupledWith is a joint realization (S\_\{f,j}\_) prescription between \ emphasit (Feature)s."
SubPropertyOf: dependency
Characteristics: Symmetric, Reflexive
ObjectProperty: hasTeamPartitionTo
Annotations: dc:description "hasTeamPartitionTo means that the given set of \ emphasit (Feature)s are partitioned to the given set of distributed \ emphasit (Team). This partitioning is the result of the distributed agile release planning (S\mathfrak{A}_{F})$."
SubPropertyOf: dependency
ObjectProperty: isImplementedIn
Annotations: dc:description "isImplementedIn denotes that a given \ emphasit (Feature) in which \ emphasit (SoftwareModule) will be implemented (S\otimes)."
D.2. Ontology of Integrated Agile Software Planning and Scheduling

- SubPropertyOf: dependency
  - Characteristics: Asymmetric, Irreflexive
  - ObjectProperty: hasPreAllocationOf
    - Annotations:
      dc:description "hasPreAllocationOf means that the given \textsf{TechnicalTask} - before scheduling - maybe allocated for a \textsf{Developer} ($\textbf{a}$)."
  - SubPropertyOf: dependency
    - Characteristics: Asymmetric, Irreflexive
    - ObjectProperty: isPrecedentedBy
      - Annotations:
        dc:description "isPrecedentedBy is a realization precedences ($P_{j',j}$) between \textsf{Features} and between \textsf{TechnicalTasks}.
  - SubPropertyOf: dependency
    - Characteristics: Asymmetric, Irreflexive
    - ObjectProperty: hasPart
      - Annotations:
        dc:description "hasPart denotes parent-child relationship between \textsf{Feature} and \textsf{TechnicalTask}s.
  - SubPropertyOf: workProductRelation
    - SubPropertyOf: relationalProperty
      - DisjointWith: hasModule
        - Characteristics: Asymmetric, Irreflexive
          - InverseFunctional
  - ObjectProperty: isSelectedTo
    - Annotations:
      dc:description "isSelectedTo signifies that a set of \textsf{Feature}s are selected to a given \textsf{Release}.
  - SubPropertyOf: dependency
    - Characteristics: Functional
      - Range: FeatureValuePartition
      - ObjectProperty: deliveryStageRelation
        - SubPropertyOf: relationalProperty
          - DataProperty: hasCapacity
          - DataProperty: hasIndex_j
          - DataProperty: hasIndex_i
          - DataProperty: hasIndex_l
          - DataProperty: hasLength
          - DataProperty: hasIndex_k
          - DataProperty: hasStartDate
D.2. Ontology of Integrated Agile Software Planning and Scheduling

**DataProperty**: hasPriority

**DataProperty**: hasWeight

**DataProperty**: hasEffectiveness

**DataProperty**: hasStartDate

**DataProperty**: hasEndDate

**DataProperty**: hasDependency

**Class**: owl:Thing

**Class**: TechnicalTask

Annotations:

\[ \text{hasPriority} \]

\[ \text{hasWeight} \]

\[ \text{hasEffectiveness} \]

\[ \text{hasStartDate} \]

\[ \text{hasEndDate} \]

\[ \text{hasDependency} \]

**Annotations:**

\[ \text{dc:description} \text{"is a fundamental working unit accomplished by one developer. Proper} \]

\[ \text{coordination requires individually realizable working units thus each } \text{RequirementFeature} \]

\[ \text{and } \text{DefectRepairFeature} \text{ should be broken down into several} \]

\[ \text{technical tasks. They usually require some working hour ( } \text{Wh} \text{) manpower that is} \]

\[ \text{estimated by developers and denoted by duration } (d) \text{. Additionally, every technical} \]

\[ \text{task is characterized with its start/completion dates } (\text{S and C}) \text{ to precise} \]

\[ \text{scheduling.} \]

**SubClassOf:**

\[ \text{hasStartDate exactly 1 xsd:int,} \]

**Workproduct,**

\[ \text{hasPreAllocationOf max 1 Developer,} \]

**hasCompletionDate exactly 1 xsd:int,**

\[ \text{hasAllocationOf exactly 1 Developer,} \]

\[ \text{hasDuration exactly 1 xsd:int} \]

**Class**: Project

Annotations:

\[ \text{dc:description} \text{"Project is a planned endeavor, usually with specific } \text{Workproduct}s, \]

\[ \text{Release}s by some } \text{Resource}s.*\]

**SubClassOf:**

\[ \text{hasSubStage only Release,} \]

**DeliveryStage,**

\[ \text{hasSubStage some Release} \]

**Class**: _ValuePartition

**SubClassOf:**

\[ \text{owl:Thing} \]

**Class**: Product

Annotations:

\[ \text{dc:description} \text{"Product is an abstract concept of the product elements. A product element} \]

\[ \text{can be a } \text{Software} \text{ or } \text{SoftwareModule}.\]

**SubClassOf:**

\[ \text{owl:Thing} \]

**Class**: Developer

Annotations:

\[ \text{dc:description} \text{"Developer is the unit of human manpower. In iteration planning, developers} \]

\[ \text{(its index is } \text{hasIndex}_i: \text{ })\text{ are allocated to (inverse of } \text{hasAllocationOf}) \text{ low level workproducts i.e. } \text{TechnicalTask}s." \]

**SubClassOf:**

\[ \text{hasIndex}_i \text{ exactly 1 xsd:int,} \]

**Resource**

**DisjointWith:**

\[ \text{Team} \]

**Class**: Resource

Annotations:

\[ \text{dc:description} \text{"Resource is an abstract concept of human manpower selected for a given } \text{Project}. A resource can be a } \text{Developer} \text{ or a } \text{Team}. Each resource is} \]

\[ \text{characterized its effectiveness factor } \text{hasEffectiveness}: \text{ ) that gives how effectively can take part in the project beside its other activities (e.g. other}} \]

\[ \text{projects, support tasks). In case of developers its value is between in } [0,1] \text{,} \]

\[ \text{while in case of team the effectiveness factors of individual members are aggregated} \]

\[ \text{(i.e. } e = \sum_i e_i).\]

**SubClassOf:**

\[ \text{owl:Thing,} \]

\[ \text{hasEffectiveness exactly 1 xsd:float} \]

**Class**: RequirementFeature

Annotations:

\[ \text{dc:description} \text{"RequirementFeature is a type of } \text{Feature that describes a customers'} \text{ Requirement."} \]

**EquivalentTo:**

\[ \text{Feature} \]

\[ \text{and } \text{hasFeatureType some Requirement} \]

186
D.2. Ontology of Integrated Agile Software Planning and Scheduling

SubClassOf:
Feature

Class: Team
Annotations:
dc:description "Team is a group of developers that are selected to the realization of a \emph{Release} from available \emph{Developer}s. During release planning, high level workproducts i.e. \emph{Feature}s are realized by the \emph{Team}.*

SubClassOf:
hasMember some Developer,
hasMember only Developer,
Resource

DisjointWith:
Developer

Class: DefectRepairFeature
Annotations:
dc:description "DefectRepairFeature is a type of \emph{Feature} that fixes defects in former product variants."
EquivalentTo:
Feature
and (hasFeatureType some DefectRepair)

SubClassOf:
Feature

Class: FeatureValuePartition
Annotations:
dc:description ""
EquivalentTo:
DefectRepair
or Requirement

SubClassOf:
_ValuePartition

Class: Release
Annotations:
dc:description "Release produces (usually external) selected deliverable \emph{Feature}s for the customer by a selected \emph{Team}, and it usually contains $1$--$4$ \emph{Iteration}s. In case of date-driven planning the length (or deadline) of release is defined ($1'$--$8$) in working months/weeks/days."

SubClassOf:
hasLength exactly 1 xsd:float,
hasSubStage some Iteration,
hasSubStage only Iteration,
DeliveryStage

Class: Requirement
Annotations:
dc:description "Requirement is a deliverable that is value for the customer. A requirement can be new or changed (including functional and non-functional ones). In most cases requirements mandate several realization steps that may include cooperation of some developers."

SubClassOf:
FeatureValuePartition

DisjointWith:
DefectRepair

Class: Feature
Annotations:
dc:description "Feature is an abstract concept of deliverable. It is selected for a given \emph{Release} and they can be classified into two kinds of set of elements according to its type: \emph{RequirementFeature}, and \emph{DefectRepairFeature}. Its realization usually needs several working days (\emph{Wd}) manpower that is estimated by developers or some methods (\emph{hasWeight}: $w$ $\{w\_j\}$)."

SubClassOf:
hasWeight exactly 1 xsd:float,
Workproduct,
hasPart some TechnicalTask,
hasPart only TechnicalTask,
isSelectedTo exactly 1 Release,
isImplementedIn only SoftwareModule,
isImplementedIn some SoftwareModule

Class: Workproduct

Annotations:
D.2. Ontology of Integrated Agile Software Planning and Scheduling

```
dc:description "Workproduct is an abstract concept of deliverables. At the release planning level, deliverables are Feature\(s\), while at the iteration planning level they are TechnicalTask\(s\). Every workproduct (its index is \(\text{hasIndex}_{j}: j\)) is characterized with its value for the customer which is denoted by priority \(\text{hasPriority}_{j}: p_{j}\)."

SubClassOf:
  owl:Thing,
  hasIndex_{j} exactly 1 xsd:int,
  hasPriority exactly 1 xsd:float

Class: FeaturePackage
Annotations:
  dc:description "FeaturePackage holds together some Feature\(s\) (expressed by isCoupledWith relations) that must be delivered together to be valued for the customer. Its resource demand is aggregated value of its parts \(\sum w_{j}\)."

EquivalentTo:
  Workproduct and (hasPart some Feature)

SubClassOf:
  hasPart only Feature,
  Workproduct,
  hasPart some Feature,
  isAssignedTo exactly 1 Iteration

Class: SoftwareModule
Annotations:
  dc:description "SoftwareModule is a deliverable product module which contains the implementation of a given Feature\(\)."

SubClassOf:
  Product,
  hasIndex_{l} exactly 1 xsd:int

DisjointWith:
  Software

Class: FeatureChunk
Annotations:
  dc:description "FeatureChunk holds together some Feature\(s\) that must be delivered by one of the distributed team to provide minimal coordination and communication intensities. Its resource demand is aggregated value of its parts \(\sum w_{j}\)."

EquivalentTo:
  Workproduct and (hasPart some Feature)

SubClassOf:
  hasPart only Feature,
  hasAllocationOf exactly 1 Team,
  Workproduct,
  hasPart some Feature

Class: Software
Annotations:
  dc:description "Software is a deliverable product which is made up of some SoftwareModule\(\)."

SubClassOf:
  Product,
  hasModule only SoftwareModule,
  hasModule some SoftwareModule

DisjointWith:
  SoftwareModule

Class: DefectRepair
Annotations:
  dc:description "DefectRepair is a deliverable that fixes defects in former product variants, and in some cases it may include cooperation of some developers."

SubClassOf:
  hasAllocationOf FeatureValuePartition

DisjointWith:
  Requirement

Class: Iteration
Annotations:
```

D.2. Ontology of Integrated Agile Software Planning and Scheduling

dc:description "Iteration is a development timebox in which intermediate deliverables i.e. \emph{Feature}s are implemented. It is characterized by available developers and iteration length \( \text{hasLength}: L[k] \) -- often expressed by iteration capacity (or velocity) \( \text{hasCapacity}: c[k] \); how many features can be delivered by the \emph{Team} in a given iteration index \( \text{hasIndex}_{k}: k \) within the release."

SubClassOf:
  hasIndex_{k} exactly 1 xsd:int,
  hasCapacity exactly 1 xsd:float,
  hasLength exactly 1 xsd:float,
  DeliveryStage

Class: DeliveryStage
Annotations:
  dc:description "DeliveryStage is an abstract concept of delivery stages as in the agile environments the software is rolled out in stages (i.e. \emph{Release}s and \emph{Iteration}s)."

SubClassOf:
  owl:Thing

DisjointClasses:
  DeliveryStage, Product, Resource, Workproduct, _ValuePartition

DisjointProperties:
  deliveryStageRelation, dependency, resourceRelation, workProductRelation

DisjointProperties:
  hasAllocationOf, hasFeatureType, hasPreAllocationOf, hasTeamPartitionTo, isAssignedTo, isCoupledWith, isImplementedIn, isPrecedentedBy, isSelectedTo
Appendix E

Overview of the Prototypes

The following prototypes were implemented that supported the experimental validation of the presented research.

E.1 PROPAS™

PROPAS™ implements the presented release and iteration and scheduling algorithms in Matlab (a popular numerical computing environment and fourth-generation programming language) to support managerial decisions. This toolbox not only contains the implemented planning algorithms, but involves several graph theoretical, scheduling and visualization capabilities. An overview of the main functions can be found in the following:

---

The following prototypes were implemented in the cooperation of Multilogic Ltd [165], OptXware Llc. [197] and the Intelligent Systems Research Group of Measurements and Information System Department at Budapest University of Technology and Economics [202].
E.1. PROPAS™

### BIN PACKING ALGORITHMS

Files

- bkpbb1 - 0-1 knapsack branch and bound algorithm (based on Horowitz-Sahni /1974/)
- bkpbruteforce - 0-1 knapsack brute force algorithm
- bkpclip1 - 0-1 knapsack algorithm computing the critical item
- bkpdpI1 - BKPDP1 0-1 knapsack dynamic programming algorithm for only integer(!) weights
- bkpdpI2 - BKPDP2 0-1 knapsack dynamic programming with memoization (memory function)
- bkpgreedy - 0-1 knapsack greedy algorithm (approximate algorithm: provides local optimum)
- bkpreduct1 - 0-1 knapsack algorithm with reduction
- ckp1 - continuous knapsack algorithm (linear programming approach)
- ckp2 - continuous knapsack algorithm (linear programming relaxation approach)
- ubounds - computes several upperbounds of the packing problem

### GRAPH THEORETICAL ALGORITHMS

Files

- findCC - find the connected components (CC) of graphs on NON-DIRECTED SCG
  (based on set unioning)
- findPath - find (the shortest) path between two nodes on NON-WEIGHTED SCG
  (based on revealBFSTree)
- revealBFSTree - breadth-first search (graph traversal algorithm) on JOINT SG
- revealDFTree - depth-first search (graph traversal algorithm) on SCG
  (based on DFS) /by Moore/
- sortTopol - topological ordering of Dags (based on DFS: ordering nodes according
to their finish times in decreasing order) /by Knuth/
- graphanalysis - some important metrics calculations on SG
- findSemiCC - find the semi connected components (SemiCC) of graphs on SCG
  (based on DFS: complabels)
- findStronglyCC - find the strongly connected components (SCC) of graphs on SCG
  (based on DFS: applied DFS on G and G')
- produceStronglyCC - produce strongly connected components (SCC) graph from an SCG
  (based on findSCC)
- findMSTKruskal - find the minimum spanning tree (MST) of graphs on NON-DIRECTED SCG /by Kruskal/
- findMSTPrim - find the minimum spanning tree (MST) of graphs on NON-DIRECTED SCG /by Prim/

### GRAPH THEORETICAL UTILS

Files

- addNodeLabel - a utility function create node labels from node ids and
  labels with concatenation
- DFS - core algorithm of the depth-first search /by Moore/
- hasDCycle - determines whether a DG contains CYCLE
  (based on DFS: checking whether it contains backward edge)
- hasLoop - determines whether an SG contains loop
- isDAG - determines whether a SG contains a DAG
  (based on isDirected and hasDCircle)
- isDirected - determines whether a SG contains any NON-DIRECTED edge
- isValidAdj - validity checking of adjacency matrix which expresses SCG
- ewadj2adj - transform edge weighted adjacency matrix (SCG) to
  adjacency matrix (SCG)
- isConnected - determines whether a SCG is a connected graph (based on findCC)

### CLUSTERING ALGORITHMS

Files

- rocsym - Rank-ordered Clustering algorithm for large matrices
- MISC ALGORITHMS

Files

- distrtask - distribute tasks between teams using binary integer linear programming

191
E.2 **PYTHIA PROJECT PLANNER**

**PYTHIA PROJECT PLANNER** is a development coordination assistant system which integrates project planning and decision support capabilities into one software package. It includes full-fledged graphical editors and computational engines to create a stand-alone and configurable toolset on the open technology and platform neutral modeling Eclipse platforms.

This chapter contains some exemplary screenshots of the **PYTHIA PROJECT PLANNER** Plugin (see Sec. 4.3.2).

### E.2.1 Pythia Planner

**PYTHIA PROJECT PLANNER** is a requirement capturing, modelling, effort estimation, project planning, and documenting subsystem which based on and integrates wide-spread standard technologies.

**Effort Estimation and Planning Computations**

- Supported main computations:
  - effort computation: based on the integration of COCOMO II, CORADMO, COQUALMO and Use-case point metrics
  - RCPS scheduling: solves resource and temporal constraint project scheduling problems with several minimization optimality criteria
  - critical path computation and PERT analysis
- Application program interfaces for Java
- Importing/exporting models in UML2 standard format (from Rational and Eclipse)
- Exporting project plan in Microsoft Project 2003
- Reporting capabilities

**Graphical User Interface**

- Platform independent graphical user interface - Java based (Eclipse EMF/GEF based)
- Supports creating system requirement specification based on custom UML2 profile extension of UML Use case and robustness diagrams
- Activity-on-Node (AoN) representation of activities in the project plan

In the following, some exemplary screenshots of the **PYTHIA PROJECT PLANNER** Plugin can be found.
Figure E.1: Pythia Planner: Requirements Editor Screen.

Figure E.2: Pythia Planner: the Pre-scheduling State of the Project Network.
E.2. Pythia Project Planner™

Figure E.3: Pythia Planner: the Built-in CO*MO (COCOMOII, CORADMO, COQUALMO) Estimation Toolset.

Figure E.4: Pythia Planner: the Exported Project Plan Can Be Seen in Microsoft Project.
E.2.2 Pythia Decision

The Pythia Project Planner™ is also based on Bayesian belief networks (BBN) technology for representing and analyzing causal models involving uncertainty. It provides a set of tools for constructing probabilistic inference and decision support systems on BBNs, including, but not limited to, the following application areas:

- Medical diagnosis
- Defect prediction, fault diagnosis, and predicting reliability of critical systems
- Operational risk in financial institutions

Bayesian Network Decision Engine

- Supported main inferences and computations:
  - posterior belief: conditional probability of a variable given the evidence
  - MPE: Most Probable Explanation given the evidence
  - SE Analysis: how sensitive the conclusion to the changes of the evidence
  - Decision support: choose best action which maximizes expected utility

- Application program interfaces for C, C++, XML and Java.
- Importing Microsoft XBN standard format

Graphical User Interface

- Platform independent graphical user interface - Java based (Eclipse GMF based)
- Construction, maintenance and usage of knowledge bases using Bayesian networks and Influence diagrams technology
- Supports development of object-oriented Bayesian networks
- Automated learning of knowledge bases from databases
E.3 SERPA™

Previously presented ARISM conceptual model is realized as an MS SharePoint-based website (named SERPA™) at Multilogic [164, 165]. SharePoint is browser-based collaboration and a document-management platform, and its capability includes creating different lists (as database tables) – such as list of technical tasks and resources. The previously constructed release and iteration information models were implemented as SharePoint lists. Thus, the portal was targeted as a collaborative workspace for developers, and a tool for the development coordination to collect all planning information. With this web-based tool, developers can break-down requirements into technical tasks, indicate precedences, set effort estimation, status of tasks/defect corrections and they also can share these information to facilitate communication. This prototype is used in every project in every day basis at Multilogic Ltd.

Features
SERPA™ capabilities includes – but not limited to – the following features:

- Platform independent web user interface
- Supports entering planning data into a common database (MS SQL Server)
- Integration with MS Office Suite
- Charting capabilities (pie chart, bar chart etc.)
- Alarm functions
- Reporting capabilities (order log, backlog, buglog) for the customers
Graphical User Interface

Figure E.7 shows the visual appearance of the prototype pointing out the Requirements list with User story, Priority, Story point and Precedence properties. The start screen of SERPA™ can be seen in Fig E.6.

Figure E.6 points out several capabilities (on the left pane):

- **Backlog**: list of the deliverable technical tasks
- **Bugs**: list of the repairable defects
- **Order log**: list of the order (that contains several backlog items)
- **Wiki**: project wiki
- **Blog**: developer blog
- **Documents**: repository of MS Office project documents

The Figure E.7 points out the Requirements list with User story, Priority, Story point and Precedence properties. Thus, the portal is targeted as a collaborative workspace for developers, and a tool for the management to collect all planning information. With this web-based tool, the team can list requirements, indicate precedences, set effort estimation and priorities, and they also can share these information to facilitate communication.
Figure E.7: SERPA™: a Prototype Website for Collaborating Workspace of Agile Projects.
Appendix F

Complexity of Algorithms and Problems

This appendix presents an introduction to the theory of algorithm and problem complexity. Complexity theory provides a mathematical framework in which computational problems are studied so that they can be classified as ‘easy’ or ‘hard’ [203, 136]. The aim of this section to provide theoretical background for the algorithms that are presented in earlier chapters.

F.1 Complexity of Algorithms

The complexity of an algorithm lies in estimating its processing cost in time (time complexity) or in the required space memory (spatial complexity). In both cases it is possible to propose a theoretical complexity and practical complexity. Theoretical complexity reflects an independent estimate on the machine which processes the algorithm. It is less accurate than the practical complexity which enables us to calculate the cost of the algorithm for a given computer. For the latter case, time complexity is obtained using an estimation of the calculation time for each instruction of the program. The advantage of theoretical complexity is that it provides an estimation independent of the calculation time for the machine [203].

From now on I use the term complexity to refer to the theoretical time complexity of an algorithm. The order of growth of the running time of an algorithm gives a simple characterization of the algorithm’s efficiency and also allows us to compare the relative performance of the alternatives. When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.
F.1. Complexity of Algorithms

F.1.1 Theoretical Time Complexity

The theoretical time complexity is established by calculating the number of iterations done by the algorithm during its processing or breaking up the algorithm into sub-algorithms of known complexity. The number of iterations depends on the size of the data, noted \( \text{Length} \), and possibly the magnitude of the largest element, noted \( \text{Max} \), belonging to these data. The theoretical complexity of an algorithm is usually a function of \( \text{Max} \), of \( \text{Length} \) and of addition and multiplying constants. If the number of iterations is bounded by a polynomial function of \( \text{Length} \) then the algorithm is of polynomial complexity. If this function is limited by a polynomial of \( \text{Max} \) and \( \text{Length} \), then we say that the algorithm is of pseudo-polynomial complexity. In other cases the algorithm is said to be of exponential complexity [203, 136].

We can also distinguish minimal, average, and maximal complexities in order to translate complexity in the best case, the average case or in the worst case respectively. These latter two actually are interesting and the easiest to calculate is maximal complexity. On the other hand, average complexity requires a statistical analysis of the processing of the algorithm by function of the input data.

We usually express asymptotic upper bound with \( O \)-notation [203]:

**Definition F.1.1 (\( O \)-notation).** Let us given a function \( g(n) \) and there exist positive constant \( c \) and \( n_0 \), we denote by \( O(g(n)) \) the set of functions

\[
O(g(n)) = f(n) : 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \quad (F.1)
\]

We usually describe asymptotic lower bound with \( \Omega \)-notation [203]:

**Definition F.1.2 (\( \Omega \)-notation).** Let us given a function \( g(n) \) and there exist positive constant \( c \) and \( n_0 \), we denote by \( \Omega(g(n)) \) the set of functions

\[
O(g(n)) = f(n) : 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \quad (F.2)
\]

And we usually denote asymptotic tight bound with \( \Theta \)-notation [203]:

**Definition F.1.3 (\( \Theta \)-notation).** Let us given a function \( g(n) \) and there exist positive constant \( c_1, c_2 \) and \( n_0 \), we denote by \( \Theta(g(n)) \) the set of functions

\[
\Theta(g(n)) = f(n) : 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \quad (F.3)
\]

To indicate that a function \( f(n) \) is a member of e.g. \( O(g(n)) \), we write \( f(n) = O(g(n)) \) [203].

Actually, the \( O \), \( \Omega \), and \( \Theta \) notations means that the complexity has an upper limit, lower limit or specify the complexity. The simplification by the notation \( O \) may lead to paradoxical situations. In effect, an algorithm \( A \) in \( O(\text{Max}^2) \) may be slower than an algorithm \( B \) in \( O(2^\text{Max}) \) for certain problems – i.e. on smaller inputs.

We use \( O \)-notation to give an upper bound on a function, to within a constant factor. To indicate that a function \( f(n) \) is a member of \( O(g(n)) \), we write \( f(n) = O(g(n)) \). Using \( O \)-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm’s overall structure. Since \( O \)-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well [203, 136].
The complexity of a well written algorithm may sometimes be improved to the detriment of the spatial complexity: it is possible to reduce the computational time of an algorithm by increasing the size of the data. However, such a step often leads to adding new functions uniquely dedicated to the management of these data. (It is clear that this complexity cannot be indefinitely broken down in order to get at the end an algorithm which complexity is null.)

F.2 The complexity of problems

Complexity theory proposes a set of results and methods to evaluate the intrinsic complexity of the problems. A problem belongs to a class of complexity, which informs us of the complexity of the ‘best algorithm’ able to solve it.

Numerous complexity classes have been defined and can be separated depending on the type of problems they address to. Basically we distinguish between DECISION, SEARCH and OPTIMIZATION problems. DECISION problems aim at existence questions: Is there an item with a given property?, SEARCH problems answers the following question types: Which item has a given property?, and finally OPTIMIZATION problems respond to questions like: Which item has extreme value within the items which have a given property?. In this section we present OPTIMIZATION problems and provide its existing complexity class [141, 203, 136, 137].

F.2.1 Basic Complexity Classes

In computational complexity \( \mathcal{P} \) is one of the most fundamental complexity classes. It contains all decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time. This class of computational problems which are ‘efficiently solvable’ or ‘tractable’. Contrary, \( \mathcal{NP} \) is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine, or equivalently can be verifiable in polynomial time by a deterministic Turing machine (i.e. we know that that the answer is indeed ‘yes’). An other important class is the \( \mathcal{NP} \)-complete decision problems, which is a subset of \( \mathcal{NP} \) and might be informally described as the most difficult problems in \( \mathcal{NP} \). A decision problem \( Q \) is called \( \mathcal{NP} \)-complete if \( Q \in \mathcal{NP} \) and, for all other decision problems \( P \in \mathcal{NP} \), we have a polynomial reduction \( \alpha \) such that it transforms all inputs for \( P \) into inputs \( Q \), i.e. \( P \preceq Q \). So, if there is a polynomial-time algorithm for even one of them, then there is a polynomial-time algorithm for all the problems in \( \mathcal{NP} \). The class of \( \mathcal{NP} \)-hard denotes those problems that ‘at least as hard as the hardest problems in \( \mathcal{NP} \)’ [204]. We are dealing with scheduling and packing problems which are not decision problems, but optimization problems. An optimization problem is called \( \mathcal{NP} \)-hard if the corresponding decision problem is \( \mathcal{NP} \)-complete [136].

The Figure F.1 shows the relations between the previous complexity classes.

![Figure F.1: Basic Complexity Classes](image_url)
F.2.2 The complexity of optimization problems

The central question is whether an OPTIMIZATION problem can belong to \( \mathcal{NP} \) or not. It is shown that class of DECISION problems are in \( \mathcal{P} \). It is not difficult to associate a DECISION problem with an OPTIMIZATION problem by searching for a solution which has a better value than a given bound. So, the OPTIMIZATION problems are subproblems of SEARCH problems: not just search for an arbitrary solution but for a solution which optimizes a given objective function. In this case, the SEARCH problem turns to an OPTIMIZATION problem and the aim becomes to calculate any optimal solution. Formally, an optimization problem is defined as follows [136].

**Definition F.2.1 (OPTIMIZATION problem).** Given instances (input) data \( I \in D \), where for each instance \( I \) a set of optimal solutions \( S \) which optimize the given objective function (criterion).

An algorithm is said to solve an OPTIMIZATION problem if, given an instance \( I \in D \), it returns the answer "no" if \( S \) is empty and otherwise returns an optimal solution \( s \in S \).

As previously stated, all the concerning problems in this research, namely the BINARY KNAPSACK 2.4.2, BINARY BINPACKING 2.4.4, BINARY MULTIPLE KNAPSACK 2.4.3 and RCPSP 2.5.3.4 problems, are \( \mathcal{NP} \)-hard problems which requires prudent treatment since without it we are only able to solve small-size problems; for the bigger one we should let the computer running for hundred years! [136]

F.3 Dealing with \( \mathcal{NP} \)-hardness

Packing and scheduling problems are optimization problems. When we address a scheduling problem, we must always look for its complexity, since this determines the nature of the algorithm to implement. If the problem under consideration belongs to the class \( \mathcal{P} \), we know that an exact polynomial algorithm exists to solve it. Otherwise, if the problem is \( \mathcal{NP} \)-hard, two alternatives are possible. The first is to propose an approximated algorithm, therefore an heuristic one, which calculates in polynomial time a solution which is as close as possible to the optimal solution. One of the most successful methods of attacking hard combinatorial optimization problems is the discrete analog of 'hill climbing', known as local (or neighborhood) search, which will be discussed at the beginning of this subsection [203, 205].

The second is to propose an algorithm which calculates the optimal solution of the problem, but for which the maximal complexity is exponential. In this case, the challenge is to design an algorithm which can solve problems of the largest possible size.

F.3.1 Local Search Techniques

Local search techniques are useful tools for solving discrete optimization problems 2.1.1. All non-preemptive scheduling problems introduced thus far are discrete optimization problems. Local search is an iterative procedure which moves from one solution in \( S \) to another as long as necessary. In order to systematically search through \( S \), the possible moves from a solution \( s \) to the next solution should be restricted in some way. To describe such restrictions, we introduce
Definition F.3.1 (Neighborhood structure). A neighborhood structure $N : S \to 2^S$ on $S$. For each $s \in S$, $N(s)$ describes the subset of solutions which can be reached in one step by moving from $s$. The set $N(s)$ is called the neighborhood of $s$.

A neighborhood structure $N$ may be represented by a directed graph $G = (V, A)$ where $V = S$ and $(s,t) \in A$ iff $t \in N(s)$. $G$ is called the neighborhood graph of the neighborhood structure. Usually it is not possible to store the neighborhood graph because $S$ has an exponential size. To overcome this difficulty, a set $AM$ of allowed modifications $F : S \to S$ is introduced. For a given solution $s$, the neighborhood of $s$ can now be defined by $N(s) = \{ F(s) | F \in AM \}$ [203].

Using these definitions, a local search method may be described as follows. In each iteration we start with a solution $s \in S$ and choose a solution $s' \in N(s)$ (or a modification $F \in AM$ which provides $s' = F(s)$). Based on the values of cost functions $c(s)$ and $c(s')$, we choose a starting solution of the next iteration. According to different criteria used for the choice of the starting solution of the next iteration, we get different types of local search techniques. The simplest choice is to take the solution with the smallest value of the cost function. This choice leads to the well-known iterative improvement method which may be formulated as follows [203, 205].

**Algorithm 6 Iterative Improvement**

1: Choose an initial solution $s' \in S$
2: repeat
3: $s \triangleq s'$
4: Generate the best solution $s' \in N(S)$
5: until $c(s') \geq c(s)$

This algorithm will terminate with some solution $s^*$. In general, $s^*$ is only a local minimum with respect to the neighborhood $N$ (i.e. a solution such that no neighbor is better than this solution) and may differ considerably from the global minimum. A method which seeks to avoid being trapped in a local minimum is simulated annealing by choosing $s'$ from $N(s)$ randomly. A variant of simulated annealing is the threshold acceptance method. It differs from simulated annealing only by the acceptance rule for the randomly generated solution $s' \in N(s)$. The $s'$ is accepted if the difference $c(s') - c(s)$ is smaller than some non-negative threshold $t$. Simulated annealing and the threshold acceptance method have the advantage that they can leave a local minimum. They have the disadvantage that it is possible to get back to solutions already visited. Therefore oscillation around local minima is possible and this may lead to a situation where much computational time is spent on a small part of the solution set. A simple way to avoid such problems is to store all visited solutions in a list called tabu list $T$ and to only accept solutions which are not contained in the list. So, in this case we have the freedom to choose a method for generating a solution from $s' \in Cand(s) \triangleq N(s)$. However, a simple strategy can be much too time-consuming, since the cardinality of the set $Cand(s)$ may be very large. For these reasons we may restrict our choice to a subset $V \subset Cand(s)$ that is solved usually heuristically [203, 205]:

$$c(\bar{s}) = \min \{ c(s') | s' \in V \} \quad (F.4)$$
F.3. Dealing with \( \text{NP} \)-hardness

F.3.2 Branch-and-Bound Algorithms

Branch-and-bound is another method for solving combinatorial optimization problems. It is based on the idea of intelligently enumerating all feasible solutions. Two things are needed for a branch-and-bound algorithm [203]:

- **Branching**: \( S \) is replaced by smaller problems \( S_i(i = 1, ..., r) \) such that \( \bigcup_{i=1}^{r} S_i = S \) This process is called branching. Branching is a recursive process, i.e. each \( S_i \) is the basis of another branching. The whole branching process is represented by a branching tree. \( S \) is the root of the branching tree, \( S_i(i = 1, ..., r) \) are the children of \( S \), etc. The discrete optimization problems created by the branching process are called subproblems.

- **Lower bounding**: An algorithm is available for calculating a lower bound for the objective values of all feasible solutions of a subproblem.

- **Upper bounding**: We calculate an upper bound \( U \) of the objective value of problem \( P \). The objective value of any feasible solution will provide such an upper bound. If the lower bound of a subproblem is greater than or equal to \( U \), then this subproblem cannot yield a better solution for \( P \). Thus, we need not continue to branch from the corresponding node in the branching tree. To stop the branching process in many nodes of the branching tree, the bound \( U \) should be as small as possible. Therefore, at the beginning of the branch-and-bound algorithm we apply some heuristic to find a good feasible solution with small value \( U \). After branching many times we may reach a situation in which the subproblem has only one feasible solution. Then the lower bound \( LB \) of the subproblem is set equal to the objective value of this solution and we replace \( U \) by \( LB \) if \( LB < U \).

**Algorithm 7** Branch-and-Bound

```
1: \text{LIST} \triangleq S
2: U \triangleq \text{value of some heuristic solution}
3: \text{currentbest} \triangleq \text{heuristic solution}
4: \textbf{while} \text{LIST} \neq \emptyset \textbf{do}
5: \hspace{1em} \text{Choose a branching node } k \text{ from LIST}
6: \hspace{1em} \text{Remove } k \text{ from LIST}
7: \hspace{1em} \text{Generate children } \text{child}(i) \text{ for } i = 1, ..., n_k \text{ and calculate corresponding lower bounds } LB_i
8: \hspace{1em} \textbf{for } i \leftarrow 1, n_k \textbf{ do}
9: \hspace{2em} \textbf{if } LB_i \leq U \textbf{ then}
10: \hspace{3em} \textbf{if } \text{child}(i) \text{ consists of a single solution } \textbf{then}
11: \hspace{4em} U \triangleq LB_i
12: \hspace{4em} \text{currentbest} \triangleq \text{solution corresponding with child } (i)
13: \hspace{3em} \textbf{else}
14: \hspace{4em} \text{add child } (i) \text{ to LIST}
15: \hspace{3em} \textbf{end if}
16: \hspace{2em} \textbf{end if}
17: \hspace{1em} \textbf{end for}
18: \textbf{end while}
```

There are many alternatives to implement a branch-and-bound algorithm. There may be many possibilities to organize the branching. When calculating lower bounds, one often has a choice between bounds that are relatively tight, but require much computation time, and bounds that are not so tight but can be computed quickly. A similar trade-off may exist in choosing a dominance relation.
F.3. Dealing with \( \mathcal{NP} \)-hardness

Finally, we should mention that the branch-and-bound algorithm is often terminated before optimality is reached. In this case we have a complete solution with cost \( U \), and the lowest lower bound \( LB \) of all nodes in the list provides a lower bound on the optimal cost. Note that if \( OPT \) is the optimal solution value, then 

\[
(U - OPT) / OPT \leq (U - LB) / LB \; \text{i.e.} \; (U - LB) / LB \text{ is an upper bound for the performance ratio for the heuristic we get by terminating the branch-and-bound procedure before reaching the optimum} \; [203, 205].
\]

F.3.3 Heuristic Algorithms

In designing algorithms, there are two fundamental goals finding algorithms 1) with provably good running time and 2) with provably optimal solution. A heuristic algorithm abandons one or both of these goals. For example, it usually finds good solutions, but there is no proof the solutions could not get arbitrarily bad (i.e. not necessarily optimal); or it usually runs reasonably quickly, but there is no argument that this will always be the case (i.e. not necessarily fast). Heuristic algorithms usually only find reasonably fast and acceptable solution to a problem in many practical scenarios, but for that there is no formal proof of its correctness. It may be correct, but may not be proven to produce an optimal solution, or to use reasonable resources. Heuristics are typically used when there is no known method to find an optimal solution, under the given constraints. Heuristic algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data [203].

For instance, finding a perfect solution to bin packing is very difficult (\( \mathcal{NP} \)-hard): there is no known way to do it that is significantly faster than trying every possible way of packing them. One usually applied heuristic is the ‘best fit’: considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it would have the smallest residual capacity. This will not necessarily be perfect packing, but it will usually give a packing that is pretty good.

There is a class of general heuristic strategies called metaheuristics, which often use some kind of search. They can be applied to a wide range of problems, but good performance is never guaranteed.

A greedy algorithm is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum. The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the unique worst possible solutions [203, 205].

F.3.4 Approximation Algorithms

A heuristic algorithm abandons the optimality goals. Approximation algorithms are increasingly being used for problems where exact polynomial-time algorithms are known but are too expensive due to the input size. Unlike heuristics, which usually only find reasonably good solutions reasonably fast, it is used when one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution) [205].

\( \mathcal{NP} \)-hard problems vary greatly in their approximability; some, such as the bin packing problem, can be approximated within any factor greater than 1 (such a family of approximation algorithms is often called
a polynomial time approximation scheme or PTAS). Others are impossible to approximate within any constant, or even polynomial factor unless $P = NP$, such as the maximum clique problem. $NP$-hard problems can often be expressed as integer programs (IP) and solved exactly in exponential time. Many approximation algorithms emerge from the linear programming relaxation of the integer program [203, 205].

For example, a polynomial-time approximation scheme (abbreviated PTAS) is a type of approximation algorithm for optimization problems which takes an instance of an optimization problem and a parameter $\varepsilon > 0$ and, in polynomial time, produces a solution that is within a factor $\varepsilon$ of being optimal. The running time of a PTAS is required to be polynomial in $n$ for every fixed $\varepsilon$ but can be different for different $\varepsilon$.

Not all approximation algorithms are suitable for all practical applications. They often use IP/LP/Semidefinite solvers, complex data structures or sophisticated algorithmic techniques which lead to difficult implementation problems. Also, some approximation algorithms have impractical running times even though they are polynomial time, for example $O(n^{2000})$. Another limitation of the approach is that it applies only to optimization problems and not to ’pure’ decision problems like satisfiability.
Index

II-notation, 200
Θ-notation, 200
$NP$ complexity, 201
$NP$-complete complexity, 201
$NP$-hard complexity, 201
$O$-notation, 200
$P$ complexity, 201
$100$-test, 37, 66
3-PARTITION problem, 50

absolute priorization technique, 37, 66
activity, 52, 54
Activity-on-Arc (AoA) network, 58
Activity-on-Node (AoN) network, 59
Actor, 33
adjacent of a vertex, 61
agglomerative clustering algorithms, 62
agile iteration scheduling function, 104
Agile Iteration Scheduling Model (AISM), 101
Agile Release and Iteration Scheduling Model (ARISM), 73
agile release scheduling function, 69
Agile Release Scheduling Model (ARSM), 76
Agile Release Scheduling Problem (ARSP), 77
agile software development, 22
Agile Tools survey, 65
Analytical Hierarchy Process (AHP), 37, 66
Assigned task First (AF), 122
association, 32
asymptotic efficiency, 199
asymptotic lower bound, 200
asymptotic tight bound, 200
asymptotic upper bound, 200
behavior models, 32
Belief, 33
BINARY BINPACKING problem (BBP), 49
BINARY INTEGER PROGRAMMING (BIP), 46
BINARY KNAPSACK problem (BKP), 47
BINARY MULTIPLE KNAPSACK problem (BMKP), 48
closed neighborhood, 61
closed-loop control system, 21
CMMI, 19
combinatorial optimization, 45
completion time of activity, 54
computation time of the project, 58
component, 61
connected graph, 61
Contribution, 33
Correlation, 33
coupling relation between features, 70
critical-path method (CPM) network, 58
Crystal methodologies, 18
date-driven release planning, 28
deadline, 55
DECISION problems, 201
Decomposition, 33
defect repair, 74
degree of a vertex, 60
delivery stage, 74
Dependency, 33
developer, 74
diameter, 61
Directed Acyclic Graph (DAG), 71
discrete optimization, 44
distance, 61
distance measure, 61
Distributed Extension of Agile Release Scheduling Model (DEARMS), 76
Distributed Software Development (DSD), 18
duration, 54
Dynamic Software Development method, 18
Earliest due date first, 58
earliest schedule, 55
equivalence relation, 70

207
event, 54
event-on-node (EoN) network, 59
EVOLVE, 37, 66
explicit dependencies, 54
exponential complexity, 200
extend, 32
Extreme programming, 18
feasible schedule, 56
feasible solution, 45
feature, 74
feature chunk, 74
feature dependency, 70
feature package, 74
feature package graph, 79
feature packages, 79
Feature-driven development, 18
fictitious activity, 54
fitness measures, 61
fixed start time, 55
Function Point Analysis (FPA), 34
generalization, 32
Goal, 33
graph, 60
graph clustering, 60
graph partitioning, 60
greedy algorithm, 205
has allocation of, 75
has member, 75
has module, 75
has part, 75
has pre-allocation of, 75
has substage, 75
has team partition to, 75
IEEE 610.12 1990 standard, 98
IFM method, 37
implemented in, 75
implicit dependencies, 54
include, 32
INTEGER LINEAR PROGRAMMING, 46
is assigned to, 75
is coupled with, 75
is preceded by, 75
is selected to, 75
ISO 12207 standard, 19
ISO 9000 standard, 19
ISO 9126 standard, 19
ISO/IEC 20926 Software Engineering - Function
Point Counting Practices Manual, 34
iteration, 74
Iterative Development Processes (IDP), 14
iterative improvement method, 203
just in case, 20
just in time, 20
k-cut, 63
k-means method, 62
Karlsson’s pair-wise comparison technique, 37, 66
KNAPSCAK problem (KP), 47
KSLOC, 34
latest schedule, 55
Lean software development, 18
LINEAR PROGRAMMING, 45
local search method, 203
makespan, 55
Means-end, 33
metaheuristics, 205
meta-potential method (MPM) networks, 59
minimax criteria, 57
MINIMUM CUT, 129
MINIMUM K-CUT, 63
minimum time lag, 54
minsum criteria, 57
MIXED INTEGER PROGRAMMING, 46
MULTIPLE SUBSET-SUM problem, 49
neighbor of a vertex, 61
neighborhood graph, 203
neighborhood of a vertex, 61
neighborhood structure, 203
non-renewable resource, 56
number of edges, 60
number of vertices, 60
objective function, 45, 56
open neighborhood, 61
open-loop control systems, 21
operational-level project plan, 19
optimal schedule, 57
OPTIMIZATION problem, 202
OPTIMIZATION problems, 201

partial order relation (PO), 70
PARTITION problem, 50
path, 61
person months, 34
polynomial complexity, 200
practical complexity, 199
precedence constraint, 54
precedence relation, 54
precedence relation between features, 70
precedes relation, 113
product, 74
project, 52, 74
project network, 58
project resource, 56
pseudo-polynomial complexity, 200
pull scheduling, 20
push scheduling, 20

Quantitative WinWin, 38, 66
quarantine time, 55

real activity, 54
recognition version of the problem, 49
regular criterion, 58
relative priorization technique, 37, 66
release, 74
release date, 55
renewable resource, 56
requirement, 74
Resource, 33
resource, 74
RESORCE ALLOCATION, 52
resource allocation problem, 56
resource relaxation, 57

RESOURCE-CONSTRAINED PROJECT SCHEDULING, 56
RESOURCE-CONSTRAINED PROJECT SCHEDULING (RCPSP) problem, 57
resource-feasible schedule, 56
schedule, 55
scheduling, 51
scheduling rules, 58
scope-driven release planning, 28

Scrum, 18
SDC study, 34
SEARCH problems, 201
SEQUENCING problem, 52
Shortest processing time first, 58
similarity measure, 60
simulated annealing, 203
SoftGoal, 33
software, 74
Software Engineering Method and Theory (SEMAT), 25
software module, 74
spatial complexity, 199
staged-delivery, 67
start time of activity, 54
State of Agile Development survey, 26
static models, 32
strategic-level project plan, 19
strict partial order, 54
tabu list, 203
tactical-level project plan, 19
Task, 33
team, 74
technical task, 74
technical tasks, 29
temporal constraint, 54
temporal relaxation, 57
test-driven development, 39
theoretical complexity, 199
threshold acceptance method, 203
time complexity, 199
time lag, 54
time window, 55
time-constrained project scheduling, 53
TIME-CONSTRAINED PROJECT SCHEDULING problem (TCPSP), 56
time-feasible schedule, 55
time-optimal schedule, 56
timebox, 16

UML Activity diagram, 32
UML actor, 31
UML Sequence Diagram, 32
UML use case, 31
UML Use case diagram, 31
Use Case Point Method (UCPM), 35
Use case Points, 34
Use Cases Maps, 32

vertex similarity measures, 61

Weighted shortest processing time first, 58
Wiegers’ method, 37, 66
workproduct, 74
Optimization: Pondering Over Concavities and Convexities

/M.C. Escher: Cube with Magic Ribbons/

Document compilation time: March, 2014 (15:11)