Conceptual scheduling model and optimized release scheduling for agile environments

Ákos Szőke*

Abstract

Context: Release scheduling deals with the selection and assignment of deliverable features to a sequence of consecutive product deliveries while several constraints 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.

Objective: To propose a solution, we present, (1) a conceptual model for agile scheduling, and (2) a novel multiple knapsack-based optimization model with (3) a branch-and-bound optimization algorithm for agile release scheduling.

Method: To evaluate our model simulations were carried out seven real life and several generated data sets.

Results: The developed algorithm strives to prevent resource overload and resource underload, and mitigates risks of delivery slippage.

Conclusion: The results of the experiment suggest that this approach can provide optimized semi-automatic release schedule generations and more informed and established decisions utilizing what-if-analysis on the fly to tailor the best schedule for the specific project context.

2011 Elsevier B.V. All rights reserved.

1. Introduction

Recently, many practitioners have adopted the ideas of agile software development [2]. Aspiring to agile approaches can be explained by recent surveys showed that agile teams are often more successful than traditional ones [3,4]. Several studies pointed out 60% increase in productivity, quality and improved stakeholder satisfaction [4,5], 40% faster time-to-market, and 60% and 40% reduction in pre-, and post-release defect rates [6] comparing to the industry average. The most popular agile methods are Extreme Programming (XP) [7] (58%), Scrum [38] (23%), and Feature Driven Development (FDD) [8] (5%) [9]. Despite the variety of methods, all of them share the core principles of Agile Manifesto [2].

In 2008, an Agile Tools survey [10] showed that many developers-focused tools were come out (including JUnit testing, sub versioning, auto builds, etc.) in the last decade, but most companies (>52%) are still using old-fashioned project management tools like MS Project 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 Rally [11]) and open source (e.g. XPlanner [12]) agile release planning tools are available.

The fourth State of Agile Development survey was conducted in 2009 [1]. The collected data includes information from 88 countries, from 2,570 participants – ranging from project managers, development managers, developers and senior managers. This survey points out that the first and the fourth most important agile techniques from the identified 19 ones are iteration and release planning. Besides these facts, the survey also revealed that the 13 most commonly cited barriers of agile adoption within companies. The three out of the four most important barriers are (1) the lack of upfront planning, (2) the loss of management control, and (3) the lack of predictability. These barriers are closely connected with the present 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. In this article, our aim is to diminish these barriers. First, we provide a proposed conceptual model of agile planning that defines a common language. The aim of the model to provide a global and common view for agile planning and to avoid the different interpretations that are used in the field. This model not only helps to identify and define the main notions but with its precise relationships it can also be used in the design of agile planning applications. Presently, the agile release planning, which is different from the ordinary release planning approaches, is unsatisfactory due to lack of solid
Projects usually have more candidate requirements than 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 [13]) determines the importance of requirements without comparing to any other requirement, while the relative technique is based on the expressed relative importance among requirements. Relative techniques tend to be more accurate and informative than absolute ones [14]. One relative technique is the $100$-test presented in [15] to assign a fixed amount of units among all requirements by different stakeholders 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 [16]. 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 [14] is based on the Analytical Hierarchy Process (AHP) [17,18]. 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’s importance regarding to the total value of the requirements. In [19], Karlsson and Ryan introduced an improved technique of the Karlsson’s pair-wise comparison technique by using cost and value as high-level factors against which each requirement pair is compared. Finally, in [20], Jung assigned values to each requirement then formulated selection of requirements as binary knapsack problem.

Compared to the extensive research on requirements prioritization only few researches dealt with requirements release planning. Carlshamre formulated release planning in [21] as an Integer Linear Program (ILP) problem, and in [22] as a binary knapsack problem where requirement dependencies were treated as constraints. Ruhe extended the ILP planning approach with stakeholders’ opinions [23], and Akker also extended it with some managerial steering mechanism that enabled what-if analysis [24]. Li combined the selection and scheduling to provide requirement selection and on-time-delivery project plan simultaneously [25]. In [27] a case study showed that integration of requirements and planning significantly (>50%) can accelerate UML-based release scheduling, and in [28] a bin-packing-based suboptimal release scheduling solution is presented. In [29] 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 for the project manager in agile environments. There are also techniques aimed at release planning, in particular when several stakeholders are involved, such as EVOLVE [31] and Quantitative WinWin [32]. Fenton et al. in [33], utilized Bayes belief network to take uncertainty of effort for resource decisions into consideration. In [34] Chang et al. proposed a method to focus on schedule level minimization but did not examine different productivity level of developers. In [35] Ngo-The and Ruhe proposed a two-phased optimization approach that combines the planning and resources allocation tasks of releases. 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 (>20).
with intensive customer collaboration during the process. Therefore, agile methods typically use an adaptive planning approach that includes usually two kinds of plans: a coarse-grained long-time (release), a fine-grained short-time (iteration) plan [13]. As a consequence, during release planning, these methods collect the essential information for the coarse-grained level plan (i.e. feature allocation to stages). These release plans typically last some development months. After release planning, they only collect more detailed information on the features of the next iteration in the given release to realize the fine-grained level plan (i.e. human resource allocation to all tasks of feature realizations). These iteration plans typically last one or two development weeks. These gradually detailed planning approaches require less information collection and planning effort from the project manager during agile release planning. Therefore these plans can be easily adapted to the changes of customer needs and company goals. As a consequence, they provide a more flexible planning approach.

In this paper, we present the core concepts of agile release and iteration planning, but we only propose an agile release planning method by formulating agile release planning as a variation of the multiple knapsack-based optimization model [61]. A complementary agile iteration planning method and a distributed agile release planning extension can be found in [29] and in [30] respectively. Those methods fit well to the presented agile release planning method.

1.2. Problem statement and analysis

The lack of penetration of the modern agile planning tools (see Section 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 [11,12]: the team can distribute deliverables among releases and iterations in planning meetings – while all planning constrains are taken into account informally. Typical informally managed planning factors are: (P1) resource capacities (resource demands during iterations), (P2) priorities (importance of each requirement delivery), (P3) dependencies (relations between requirements), (P4) staged-delivery (delivery time of iteration timeboxes), and (P5) maximal value (to choose the maximal valued one from different plans). The principles of agile development (see [2]) rely on communication instead of rigorous planning, this fact can be explained by the lack of easily applicable algorithmic solutions. Informal approaches work well in smaller projects and project manager intuition is necessary, but they are not sufficient in complex decision situations. As the size and complexity increase scheduling becomes a very complex process and advocates tool support [23,26,35]. Moreover, the usual manual approach makes it difficult to construct different plans to perform what-if analysis in order to select the best plan from the potential candidates. As a consequence, 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 Section 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 it cannot be carried out without proper tool support.

1.3. Objectives

To provide a systematic approach without omitting the human expert from the process, we propose a formal agile release planning problem description and algorithmic solution which consider the previous factors (P1–5) – while the deficiencies of the informal and intuitive approach are mitigated. Therefore, the main contribution of the paper can be seen as to make release planning process of agile development more objective and more qualified in terms of the results.

Firstly, our intention is to, (1) provide a detailed conceptual model of agile planning and scheduling. Then we propose an agile release scheduling method which intends to mitigate previous problems (Section 1.2) by (2) formulating release scheduling as an multiple knapsack-based optimization model that considers all the previous factors (P1–5), and (3) providing a solution to this model by a branch & bound algorithm to easily produce optimized release scheduling.

1.4. Outline

The rest of the paper is arranged as follows: Section 2 presents elements of agile planning; Section 3 introduces the proposed (1) conceptual model, (2) optimization model and (3) algorithm for agile release scheduling; Section 4 describes experiences; Section 5 discusses our solution; and finally Section 6 concludes the paper.

2. Background

In this section, we introduce the concepts of agile planning to provide the necessary background for the proposed conceptual and optimization model.

2.1. Business value in agile

Business value is a key concept in agile software development [36]. Agile approaches can be considered as a paradigm, a ‘minimalist software development attitude’ which aim is to carry out those tasks that provide business value for the customer [37]. A business value term is being used in management and economics that includes all forms of value for the company in the long-run. It usually means anything that can be translated to money such as revenue, market share, and stock price – it is generally a multi-dimensional concept.

2.2. Agile features and effort estimation

Common to all software development processes in any project is the need to capture and share knowledge about the requirements and design of the product, the development process, the business domain, and the project status. Contrary to the traditional methods (e.g. waterfall), agile methods advocate ‘just enough’ specifications where the fundamental issue is the communication, not the documentation [38,39]. In agile methods User stories, Features, and Use cases (XP [7], FDD [8], Use cases [40]) are the most popular specification techniques.

Most software products evolve as they go live, therefore there is a need to repair defects that are discovered during operation. Both deliverables (requirements and defect repairs) are usually called features. The agreed specification of features not only drives the development, but directs planning of projects and provides the basis for effort estimations [41]. Agile teams focus on ‘good enough’ estimates to try optimizing their estimating efforts. A good estima-
tion approach takes short time, and can be used for planning and monitoring of the project [13].

Requirements effort estimation models are usually based on size metrics: Story points [13], Feature points [8], Use case points [42] and Object-Oriented Function Points [43]. These metrics are abstract units, express the whole size of development and are usually not directly convertible into person hours/days/months. Whereas, defect repairs are usually estimated with person hours/day.

2.3. Agile release planning and scheduling

Release planning and scheduling are activities concerned with the implementation of selected features in the next version of the software. While planning decides what needs to be done, scheduling defines when to do it. Agile approaches mandate incremental software development approach which aims at delivering smaller releases sequentially instead of delivering the whole system at once. This approach necessitates staged-delivery release scheduling process: selecting and assigning features to consecutive increments. The selection is based on relative priorities as a function of business values (Section 2.1) and other factors such as urgencies and risks [31] – depending on the stakeholders’ needs. Relative priority values - measured on ratio-scale, which means that priority values are ordered and the ratio between values can be determined [44] – ensure that the features are comparable to support selection. Actually, measurement on ordinal-scale (values can be ordered, values are ordered and the ratio between values can be determined) or priority values - measured on ratio-scale, which means that priority values are ordered and the ratio between values can be determined – depending on the stakeholders’ needs. Relative priority values are useful in agile release scheduling (RS) as they provide a global view of agile release and iteration scheduling.

2.4. Iteration planning and scheduling

Once the maximal customer-valued 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 [13,9]. 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 features into technical tasks and to assign them to developers [13].

2.5. 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 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 (Algorithm 1). The lines in parentheses and marked with \(\dagger\) mean that they are excluded from the scope-driven and included in the date-driven case. While mark \(\ddagger\) denotes the opposite case. In the program listing lowercase/uppercase letters with indices denote vectors/matrices (e.g. \(p, D_{ij}\)) and without indices, they mean scalars (e.g. \(p\)). While bold-faced types show concise (without indices) forms (e.g. \(D\)), and Gothic types denote sets (e.g. \(\mathcal{A}\)).

In the require section the preconditions are given. At the beginning of the process, the release constraints – such as required release duration (\(\hat{R}\) in days), available developers (\(\mathcal{A}\)) and their effectiveness factors (\(e_i\); proportion of his/her daily work on this release), and dependencies between features (\(D_{ij}\)) – 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 User story \(j\) 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>/<em>Release duration</em>/</th>
<th>/<em>Available developers and their effectiveness factors</em>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\hat{R}\in\mathbb{N})</td>
<td>/<em>Selecting developers</em>/</td>
<td>/<em>Calculation of the velocity of selected resources</em>/</td>
</tr>
<tr>
<td>(i\in\mathcal{A}, e_i\in[0,1])</td>
<td>/<em>Priorities, efforts and dependencies</em>/</td>
<td>/<em>Assessment of iteration count and length of each iteration</em>/</td>
</tr>
<tr>
<td>(j\in\mathcal{W}, p_j\in\mathcal{W}, w_j\in\mathbb{N}, D_{ij}\in{0,1})</td>
<td>/<em>Calculating iteration velocities</em>/</td>
<td>/<em>Select and assign features to iterations</em>/</td>
</tr>
<tr>
<td>(D_{ij})</td>
<td>(</td>
<td>X_0</td>
</tr>
<tr>
<td>(k, t_k)</td>
<td>(i=1) repeat</td>
<td>/<em>Selecting developers</em>/</td>
</tr>
<tr>
<td>(\nu^* = \sum_{i\in\mathcal{A}} e_i i\in\mathcal{A})</td>
<td>/<em>Calculation of the velocity of selected resources</em>/</td>
<td>/<em>Calculating iteration velocities</em>/</td>
</tr>
<tr>
<td>(k, t_k)</td>
<td>/<em>Assessment of iteration count and length of each iteration</em>/</td>
<td>/<em>Select and assign features to iterations</em>/</td>
</tr>
<tr>
<td>(X = \text{schedule}(p, w, D, c))</td>
<td>/<em>Select and assign features to iterations</em>/</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).

In this process, the settings of \(i, k\), and \(t_k\) factors remained informal, their values mainly depend on the development situation and local agile practices. Their typical values are between 4–12 people, 2–4 iteration counts, 1–4 weeks respectively [3,13,9]. It is appropriate to calculate team velocities (\(\nu^*\) 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 [13].

3. Optimized agile release scheduling

In this section, first, we introduce interdependencies between features (Section 3.1) heavily influencing the scheduling process. Secondly, we present a proposed conceptual model (Section 3.2) to provide a global view of agile release and iteration scheduling. Then we point out that agile release scheduling (RS) can be characterized as a specialized multiple knapsack problem, and
we formulate a general optimization model for this problem (Section 3.4) – or in other words, a model for the schedule function (Algorithm 1). Finally, we present a solution to this optimization model in the form of a branch & bound algorithm (Section 3.5).

### 3.1. 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 [21] and summarized in Table 1. To clarify the meaning of dependencies, the following examples are given in Table 2. (Note, \( j, j' \) denote requirements):

Theoretically, there can be \( C_n = 6(n - 1)/2 = 3n(n - 1) \) 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 [21] it was found that this number is roughly between \((n, 2n)\).

#### 3.1.1. Mapping requirement dependencies to feature dependencies

Considering release scheduling two important types of dependencies should be identified [23]: (1) coupling (C): requirements connected with \( \text{AND} \) dependency should be released jointly because they expect each other to serve a unit of functionality, (2) precedence (P): requirements connected with \( \text{REQUIRES}, \text{CVALUE}, \text{ICOST} \) or \( \text{TEMPORAL} \) dependencies should be released in given temporal order (i.e. \( j \) before \( j' \)). Dependency \( \text{XOR} \) only influences preliminary requirement selection during the requirement’s specification phase and does not influence on release scheduling. Table 3 summarizes the previous mappings.

Besides requirements’ implementation, defect repairs should also be ordered during scheduling thus we interpret these two dependencies (i.e. coupling and precedence) and call them feature dependencies (cf. Section 2.3).

#### 3.1.2. Constructing singular feature dependencies

Feature dependencies can be modeled as binary relations. Precedence dependency can be defined as partial order relation [48] (usually written as ‘\( \leq \)’) so it splits the set of features up (i.e. partition) into strict subsets which group jointly releasable items. Therefore coupling possesses the Eqs. (1a), (1b) and (1d) properties from the following:

- If \( j \in \# \), then \( j \sim j \) (reflexive)
- If \( j \sim j' \), then \( j \sim j' \) (symmetric)
- If \( j \sim j \) and \( j \neq j' \), then \( j \sim j' \) (anti-symmetric)
- If \( j \sim j' \) and \( j' \sim j' \), then \( j \sim j' \) (transitive)

Analogically, coupling can be interpreted as equivalence relation [48] (usually written as ‘\( \equiv \)’ so it splits the set of features up (i.e. partition) into strict subsets which group jointly releasable items. Therefore coupling possesses the Eqs. (1a), (1b) and (1d) properties from the Table (1).

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 ‘\( \equiv \)’.

#### 3.2. Conceptual model of agile release and iteration scheduling

In order to formulate the release scheduling optimization model, first, we have to identify the main concepts of agile scheduling. Next, we 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.

Our proposed conceptual model of agile release and iteration scheduling (Section 2) is visualized with UML notation in Fig. 1 and presented in Table 4 [13,40]. Generally, scheduling mandates defining who will realize what and when. Team, Feature and Release

### Table 1

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

### Table 2

<table>
<thead>
<tr>
<th>Dependency name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{AND} )</td>
<td>A customer report functionality needs a report visualization component ( j ) (e.g. Crystal Reports [45]) and a data query ( j' ) to function</td>
</tr>
<tr>
<td>( \text{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>( \text{XOR} )</td>
<td>A customer report visualization function may be based on Crystal Reports component ( j ) or on an in-house solution ( j' )</td>
</tr>
<tr>
<td>( \text{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>( \text{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>( \text{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>

### Table 3

<table>
<thead>
<tr>
<th>Requirement dependency group</th>
<th>Type</th>
<th>Influence on selection (see [21])</th>
<th>Influence on RS</th>
<th>Feature dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality-related</td>
<td>( \text{AND} )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( C )</td>
</tr>
<tr>
<td>Value-related</td>
<td>( \text{REQUIRES} )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( P )</td>
</tr>
<tr>
<td>Time-related</td>
<td>( \text{ICOST} )</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( P )</td>
</tr>
<tr>
<td></td>
<td>( \text{TEMPORAL} )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( P )</td>
</tr>
</tbody>
</table>
Table 4
Concepts of agile release and iteration planning.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<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>(Section 2.5)</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_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 selected and assigned to the iteration. It is characterized by available developers ( \sum e_i ) and iteration length ( L ) – often expressed by iteration capacity (or velocity) ( c ): how many features can be delivered by the team in a given iteration index ( k ) within the release</td>
<td></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 ( e ) 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></td>
</tr>
<tr>
<td>Developer</td>
<td>Is the unit of human manpower. In iteration planning, developers (its index is ( i )) are allocated to low level workproducts i.e. technical tasks</td>
<td></td>
</tr>
<tr>
<td>Team</td>
<td>Is a group of developers 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></td>
</tr>
<tr>
<td>Workproduct</td>
<td>Abstract concept of deliverables. At the release planning level, deliverables are features, while at the iteration planning level they are technical tasks. Every workproduct (its index is ( j )) is characterized with its value for the customer which is denoted by priority ( p )</td>
<td></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 three kinds of set of elements: feature packages, requirements, and defect repairs. Its realization usually needs several working days (( Wd )) manpower that is estimated by developers or some methods (( 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>Defect repair</td>
<td>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>Feature package</td>
<td>Holds together some features (expressed by Coupling relations) that must be delivered together to be valued for the customer. Its resource demand is aggregated value of its parts ( \sum w_j ) (see Section 3.5.1)</td>
<td></td>
</tr>
<tr>
<td>Technical task</td>
<td>Fundamental working unit accomplished by one developer. Proper coordination requires individually realizable working units thus each requirement and defect repair 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 ( d ). Additionally, every technical task is characterized with its start/completion dates ( S ) and ( C ) to precise scheduling</td>
<td></td>
</tr>
<tr>
<td>Work product dependency</td>
<td>Abstract concept of dependencies between workproducts. A dependency can be a precedence or a coupling</td>
<td></td>
</tr>
<tr>
<td>Precedence</td>
<td>Realization precedences between requirements, defect repairs and technical tasks</td>
<td></td>
</tr>
<tr>
<td>Coupling</td>
<td>Realization precedence between requirements and defect repairs</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Conceptual model of agile release and iteration scheduling.
concepts answer to these questions in agile release scheduling (cf. Sections 2.2, 2.3). 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 (Section 2.4). 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 (Section 2.5), temporal constraints can be asserted as attributes (e.g. deadlines – Section 2.5) or Precedence and Coupling dependencies (Section 3.1.1).

In Fig. 1, shaded objects (like Developer and TechnicalTask) pertain to iteration scheduling only. 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.

3.3. Mapping to multiple knapsack problem

In this section, we 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 0–1 knapsack problem (BKP) instance is specified by a knapsack capacity and a set of items where each item has profit and weight. 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 [46,47]. An important generalization of the 0–1 knapsack problem is the 0–1 multiple knapsack problem (BMPK) which arises when more than one knapsack is given with possible multiple knapsack problem

sack capacity \([46,47]\). An important generalization of the 0–1 knapsack problem is the 0–1 multiple knapsack problem (BMPK) 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. \(v^d\) in Section 2.5) in an iteration stands for a knapsack capacity, while a feature’s resource demand (cf. \(w_j\) in Section 2.5) represents an item weight. We can view each iteration (cf. \(k\) in Section 2.5) 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 team’s iteration velocity (i.e. knapsack capacity). We apply priority instead of profit to express values of features.

The two release scheduling question types (cf. Section 2.5) can be formed as:

- 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).
- 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 other 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.

We extend the ordinary BMK problem and interpretation with the following elements to provide further computational capabilities for wide-ranging scheduling situations (cf. Section 1.2; P1–P5): (1) flexible packing approach, (2) dependencies between features (cf. Section 3.1). From now on we call this extended problem as agile release scheduling problem (ARSP).

3.4. Formulating ARS optimization model

Let us given a set of deliverable features \(j (j \in \mathbb{N}^+: |\mathbb{N}| = n)\) with resource demands \(w_j\) and iterations \(k (k \in \mathbb{N} : |\mathbb{N}| = 0)\) with different iteration velocities \(c_k\) within a release. In case of Fixed-scope scheduling \(a \leq n\) (considering \(n\) features the iteration count cannot be more than \(n\)), and in case of Fixed-time scheduling \(a \leq FT/\sum_{j=1}^{n} c_j\), where \(FT\) is the Fixed-time duration of the release and \(c_j\) 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) \(P_{ij} \in \{0, 1\}\) (where \(P_{ij} = 1\) if \(j\) precedes \(i\), otherwise \(P_{ij} = 0\)) and coupling relations (matrix) \(C_{ij} \in \{0, 1\}\) (where \(C_{ij} = 1\) if \(j\) is coupled with \(i\), otherwise \(C_{ij} = 0\)) hold. A possible mathematical formulation can be found in (2) (see definitions of \(p_j\), \(w_j\) in Section 2.5). The equation marked with \(\dagger\) means the date-driven (some tasks may be omitted due to lack of resource) case while \(\dagger\) denotes the scope-driven (every task must be done in some iteration) case.

Maximize 
\[
\sum_{k=1}^{n} \sum_{j=1}^{n} p_j x_{kj}
\]
subject to
\[
\sum_{j=1}^{n} w_j x_{kj} \leq c_k, \quad k = 1, \ldots, m
\]
\[
K - k \geq P_{ij} : \quad x_{kj} = 1
\]
\[
\lfloor 1/(k - k + 1) \rfloor = C_{ij} : \quad x_{kj} \cdot x_{kj} = 1
\]
\[
\left( \sum_{k=1}^{m} x_{kj} \leq 1 \right) \lor \left( \sum_{k=1}^{m} x_{kj} = 1 \right) : \quad j = 1, \ldots, n,
\]
where \(y_k = 0\) or \(1\), and \(x_{kj} = 0\) or \(1\), and
\[
x_{kj} = \begin{cases} 
1 & \text{if } j \text{ is assigned to iteration } k \\
0 & \text{otherwise}
\end{cases}
\]
\[
y_k = \begin{cases} 
1 & \text{if iteration } k \text{ is used} \\
0 & \text{otherwise}
\end{cases}
\]

The equations denote: (2a) deliverable value maximization of release (i.e. summing up values of delivered features \(p_j\) in each iteration \(k\), (2b) resource constraints (i.e. resource demands of delivered features \(w_j\) in iterations cannot be greater than the total effort in different iterations \(c_k\), (2c) temporal constraints (i.e. if \(j\) precedes \(i\) \((P_{ij} = 1)\) then iteration index \(K\) must be greater than or equal to \(i\), (2d) coupling constraints (i.e. if \(j\) is coupled with \(i\) \((C_{ij} = 1)\) then iteration index \(K\) must be equal to \(k\), and finally (2e) 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. Section 2.5).

We will suppose, as is usual, that the efforts \(w_j\) are positive integers. Without loss of generality, we will also assume that
\[
c_k \text{ is a positive integer}
\]
\[
w_j \leq c_k \text{ for } \forall k, j
\]
If the assumption (4a) is violated, \( c_k \) can be replaced by \( |c_k| \). If an item violates the assumption (4b), then the instance is treated as trivially infeasible. For the sake of simplicity, we will also assume that, in any feasible solution, the lowest indexed iterations are used, i.e. \( y_k \geq y_{k+1} \) for \( k = 1, 2, \ldots, n - 1 \).

### 3.5. Solving the ARS optimization problem

For the previously formulated optimization model, we developed a multiple knapsack scheduling algorithm (Algorithm 2). It is a branch & bound algorithm, which iteratively selects and schedules an item (feature) into an iteration.

#### 3.5.1. Constructing feature packages

The structural analysis of the ARSP model leads to the decomposition of the release schedule into a set \( \mathcal{W} \) of features, a set \( \mathcal{P} \) of precedence and a set \( \mathcal{C} \) of coupling relations. These elements can be represented as a feature graph \( \mathcal{G}_F = (\mathcal{W}: \mathcal{P}: \mathcal{C}) \), where the precedence and the coupling relations are drawn as directed and non-directed edges respectively. Therefore, \( \mathcal{G}_F \) is a multigraph as the same two vertices may have two edges. One can notice that \( \mathcal{G}_F = \mathcal{G}_P \cup \mathcal{G}_C = (\mathcal{W}: \mathcal{P} \cup \mathcal{C}) \), so \( \mathcal{G}_F \) and \( \mathcal{G}_C \) are spanning subgraphs of \( \mathcal{G}_P \).

According to the coupling definition (cf. Section 3.1) coupled items may be interpreted as feature packages, where coupled features are joined. As a consequence, a graph \( \mathcal{G}_C \) has to be transformed into a feature package graph \( \mathcal{G}_{FP} \) where precedences are preserved. This \( \mathcal{G}_{FP} \) can be constructed in the following way.

For each \( j \in \mathcal{W} \) we 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 \( j \in \mathcal{W} \) belongs to some equivalence class \( [j] \in \mathcal{W} \) and the union of all equivalence classes is \( \mathcal{W} \). Then we can say that the equivalence class (coupling relation) partitions \( \mathcal{W} \). As a consequence, we can define coupled items as \( [j] \) and the set of them as \( \mathcal{W}^C = \bigcup [j] \). As for precedence relation, we have to construct a new precedence relation – namely \( \mathcal{P}^C \) – on \( \mathcal{W}^C \) considering \( \mathcal{P} \) on \( \mathcal{W} \). 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 produce self (not valid) precedences (i.e. \( P_{i,j} : j \in [j] \) in \( \mathcal{W}^C \) therefore we have to leave them out. The latter one, however, can produce – possible multiple – valid precedences (i.e. \( P_{i,j} : j \in [j'], j \in [j] \)). In case of multiple precedence we need to preserve just one of them.

Fig. 2 shows a feature package construction example using post-mortem analysis results of a real life development at Multilogic [51]. 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 and 11. In this case, after graph transformation (i.e. \( \mathcal{G}_F \rightarrow \mathcal{G}_{FP} \)), 13 dependent nodes are produced from the 19 – as the coupled nodes are merged. (Notice, the transformation reduces the indices in the resulted graph so nodes 1, 5 became 1, (2) nodes 17, 18 and 19–21 on the left have no precedence after transformation, so they are – 16 and 17 respectively – not shown on the right.)

### 3.5.2. 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 ARS problem. Up to now several algorithms have been proposed to solve the BKP, although the Horowitz–Sahni algorithm is one of the most effective [47]. We extended this algorithm with (1) multiple knapsack capabilities, (2) flexible packing approach (cf. Section 3.3), and (3) precedences between items (Section 3.1). 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.

In the program listing letters are interpreted as in Algorithm 1. The algorithm assumes that the items are ordered according to decreasing values of the priorities per resource demands, i.e. \( p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n \). The course of the proposed algorithm is outlined as follows. It solves the 0-1 multiple knapsack problem as the solution of several 0-1 knapsack subproblems (see first while in the Algorithm 2). In each subproblem solving, first the schedulable (not scheduled and not preceded) 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 if backtracking is not possible.

An important feature of the algorithm is that it deals with precedences in a soft way: if not every item is packed and room remains 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 a precedence relation between two items, both may be put to the given iteration.

In the Require section the preconditions are given. Each \( w_i \) is the resource demand for item \( j \). Precedences between items are represented by a matrix where \( P_{ij} = 1 \) means that item \( j \) precedes \( i \), otherwise \( P_{ij} = 0 \). Both conditions \( P_{ij} = 0 \) (no loop) and \( P \) is directed acyclic graph (DAG) ensures that temporal constraints are not trivially unsatisfiable. Additionally, priorities \( p_i \) and iteration velocities \( c_i \) are given. The date-driven case can be controlled with the length of \( c_k \) (cf. Section 2.5). The Ensure section prescribes the two kind of postconditions on the return value (X) depends on scheduling question (cf. Section 2.5, (2e)).

During scheduling steps, first the initial values are set (lines 1–4). The assignment matrix X contains the feature package assign-
ments to iterations – initially it is set to a zero matrix ($\text{items} \times \text{iteration count}$) (line 1). The algorithm uses a ready list ($\text{rlist}$) and a scheduled list ($\text{slist}$) to keep track of schedulable and scheduled items – these lists are set to empty sets (line 2). Finally, the iteration index ($\text{idx}$) 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 ($\text{ns}$) (line 6) then constructs the actual ready list ($\text{rlist}$) from which the algorithm can choose in the current control step without violating any precedence constraint (line 7). If the ready list does not 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 ($\text{pn}$) and resource demands ($\text{wn}$) 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 (lines 13 and 14). If all the items are not packed to the given iteration then the current residual iteration velocity ($\text{cn}$) 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).

---

**Algorithm 2. mksched algorithm (2/1)**

```plaintext
Require
$W_j \in \mathbb{R}$
$P_{ij} \in 0, 1 \land P_{ij} = 0 \land P \text{ is DAG}$
$P_{ij} \in \mathbb{N}, C_k \in \mathbb{N}$

Ensure $\forall j \in (X_{ij} \in \{0, 1\})^\dagger \text{xor} (X_{ij} = 1)^*$

1: $\text{X} \leftarrow [0, \text{length}(\text{w}), \text{length}(\text{c})]
2: \text{rlist} \leftarrow 0, \text{slist} \leftarrow 0
3: \text{idx} \leftarrow 1
4: \text{remcap} \leftarrow \text{false}
5: \text{while} (\text{findNotSchedItems (slist)} \neq 0) \land (\text{length}(\text{c}) \geq \text{idx}) \text{do}
6: \text{ns} \leftarrow \text{findNotSchedItems (slist)}
7: \text{rlist} \leftarrow \text{findNotPrecItems (ns, P)}
8: \text{if} \text{rlist} = 0 \text{then}
9: \text{print} "\text{Infeasible schedule!}, \text{return} 0"
10: \text{end if}
11: \text{pn} \leftarrow \text{p(rlist)}, \text{wn} \leftarrow \text{w(rlist)}
12: \text{n} \leftarrow \text{length (rlist)}
13: \text{x} \leftarrow [0]_n, \text{xn} \leftarrow [0]_n
14: \text{z} \leftarrow 0, \text{zn} \leftarrow 0
15: \text{if} \text{remcap} \text{then}
16: \text{cn} \leftarrow \text{crem}
17: \text{else}
18: \text{cn} \leftarrow c(\text{idx})
19: \text{end if}
20: \text{backtrack} \leftarrow \text{false, opt} \leftarrow \text{false}, j \leftarrow 0
21: \text{while} \text{not opt} \text{do}
22: \text{while} j \leq n \land \text{backtrack} \leftarrow \text{false} \text{do}
23: \text{u} \leftarrow \text{computeUBound}(j)
24: \text{if} z < (\text{zn} + u) \text{then}
25: \text{while} \text{wn}(j) \leq \text{cn} \text{do}
26: \text{cn} \leftarrow \text{cn} - \text{wn}(j), \text{zn} \leftarrow \text{zn} + \text{wn}(j), \text{xn}(j) \leftarrow 1
27: \text{j} \leftarrow j + 1
28: \text{end while}
29: \text{if} j \leq n \text{then}
30: \text{xn}(j) \leftarrow 0
31: \text{j} \leftarrow j + 1
32: \text{end if}
33: \text{else}
34: \text{backtrack} \leftarrow \text{true}
35: \text{end if}
36: \text{end while}
37: \text{if} zn > z
38: \text{x} \leftarrow \text{xn}
39: \text{end if}
40: \text{xnidx} \leftarrow \text{findAssigned(xn)}
```

/*weights of each feature*/
/*precedences*/
/*priority values and iteration velocities*/
/*assignment matrix initialization*/
/*ready list and scheduled list initialization*/
/*iteration index initialization*/
/*remained velocity is not utilized*/
/*find not scheduled items*/
/*construct ready list*/
/*current priorities and resource demands*/
/*items in the 'ready list'*/
/*best and current assignment vector*/
/*best and current values*/
/*let current residual velocity equal to remained velocity*/
/*let current residual velocity equal to iteration velocity*/
/*backtracking, optimum, search index*/
/*cycle of subproblems*/
/*cycle of forward move*/
/*computing upper bound from j*/
/*forward move may lead to better solution*/
/*insert the largest set of consecutive items*/
/*take away it from the current solution*/
/*forward move does not lead to better solution*/
/*forward move produced better solution*/
/*update the current solution as the best one*/
Algorithm 3. mksched algorithm (2/2)

41: \( k = 0 \)
42: \( \text{if isEMPTY(xnidx) then} \) /*backtracking is no more possible*/
43: \( \text{opt} \leftarrow \text{true} \)
44: \( X[rlist\text{findAssigned}(X)] \leftarrow 1 \) /*store scheduled items*/
45: \( slist[rlist\text{findAssigned}(X)] \leftarrow 1 \) /*update scheduled-list*/
46: \( \text{if \_isEMPTY(findNotSchedItems(slist)) then} \)
47: \( \text{minnsi} \leftarrow \text{minWeightOfNotSchedItems(slist)} \)
48: \( \text{cnrem} \leftarrow \text{cn} - \text{sum(findWeightOfAssigned(X))} \) /*remained vel.*/
49: \( \text{if cnrem} < \text{minnsi} \then\)
50: \( \text{remcap} \leftarrow \text{false} \) /*not scheduled items*/
51: \( \text{idx} \leftarrow \text{idx} + 1 \) /*increment iteration*/
52: \( \text{else} \)
53: \( \text{remcap} \leftarrow \text{true} \) /*remained velocity utilization*/
54: \( \text{end if} \)
55: \( \text{else} \)
56: \( \text{idx} \leftarrow \text{idx} + 1 \) /*increment iteration*/
57: \( \text{end if} \)
58: \( P_{\{1,\ldots,n\},j} = 0 \) /*delete scheduled items*/
59: \( \text{else} \)
60: \( k \leftarrow \text{max(xnidx)} \) /*find the last inserted item*/
61: \( \text{end if} \)
62: \( \text{if \_opt \wedge (backtrack \vee j > n) then} \) /*backtracking -*/
63: \( \text{cn} \leftarrow \text{cn} + \text{wn}(k), \text{zn} \leftarrow \text{zn} - \text{pn}(k), \text{xn}(k) \leftarrow 0 \)
64: \( j \leftarrow k + 1 \)
65: \( \text{backtrack} = \text{false} \)
66: \( \text{end if} \)
67: \( \text{end while} \)
68: \( \text{end while} \)
69: \( \text{return X} \)

The third while (line 22) represents forward moves and iterates while the search index is not out of scope \((j > n)\) and backtracking is not necessary. Whenever a forward move performed, an upper-bound 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 (lines 23 and 24) otherwise backtracking 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 (lines 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 (lines 29–32).

If forward move produced a better current solution than the best solution so far, we update the best solution (lines 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 (lines 47, 49) then remained velocity utilization is set (line 53) else it is annulled and the iteration count is incremented (lines 50 and 51). If the current solution is not empty, it calculates the last inserted item’s index (line 60) which is used in backtracking (lines 62–66).

After termination, \(X\) contains the item assignments to iterations, where the number of nonzero columns denotes the packed iterations (cf. (2a)):

\[
z \leftarrow \text{length}
\left(\text{nonZeros}\left(\sum_{j=1}^{n} w_{j} x_{j}\right)\right)
\] (5)

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 \(O(n \cdot n^{2})\). Since several packing combinations are not feasible, the tree has less nodes in practice so it provides sufficient results for practical applications [47]. As previously stated, it assumes that the items are sorted, if it is not the case, they can be re-indexed in \(O(n \cdot \log n)\) time through an efficient sorting algorithm (e.g. Quicksort).

We 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 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 (lines 6 and 7).

As previously stated, the presented algorithm is extended the Horowitz-Sahni algorithm [47] in many ways while the core branch & bound search idea of the original algorithm is unchanged. Namely, we extended this algorithm with (1) multiple knapsack capabilities, (2) flexible packing approach (cf. Section 3.3), and (3) precedences between items (Section 3.1). The first extension is realized by dealing with different knapsack capacities (i.e. itera-
tion velocities \( c_i \) and embedding the Horowitz–Sahni algorithm into an iterative frame (the external loop, lines 5–68) in order to support multi-iteration scheduling within a release. The second extension is simply carried out by a well-formulated conjunction in the condition of the external loop (line 5). The first part of the conjunction ensures the scope-driven, the second part ensures the date-driven scheduling. Finally, the third extension is realized by introducing the ready list (\( rlist \)) and the scheduled-list (\( slist \)) to keep track of schedulable and scheduled items considering the precedence constraints between features (\( P \)) during the course of the algorithm.

Fig. 3 illustrates the scheduling concept continuing the previous example (see Section 3.5.1). It shows the post-mortem release scheduling result based on a real life development situation using the Algorithm 2. Fig. 3a 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. Fig. 3b depicts the scheduling results produced by \( mksched \) algorithm in stacked bar chart form (where labels denote the indices of feature packages and values within brackets show the size of the packages): the previous feature packages are assigned to five iterations (\( x \)-axis) with capacities 30, 30, 30, 29.5, and 18 (\( y \)-axis). Bar colors in Fig. 3a point out how feature packages are distributed in Fig. 3b, and in each bar, the values denote indices and the working days (in brackets) of the given feature package.

4. Experimentations

To obtain a proof-of-concept, we implemented a prototype as a scheduling toolbox in Matlab [50]. Seven past release data sets – extracted from the backlog of IRIS application developed by Multilogic Ltd [51] – were compared against the results of simulations applying the same inputs [49]. Additionally, we also carried out simulations on generated hypothetical data sets to get an insight into the performance of the presented algorithm on larger problems.

4.1. 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 1990s. 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 project manager 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.

4.2. Data collection and results

Seven data sets (Collateral evaluation, Risk assumption, Ukrainian deal flow I–II, Romanian deal flow I–III – respectively \( RA \) to \( RG \)) were selected to make a comparison between the algorithmic method and the manual release scheduling carried out previously at Multilogic. The \( RC \) data set is used to present the concept in the previous example (Figs. 2 and 3a and b). 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 (\( FSi \)). Table 5 summarizes the variables \( RA \) to \( RG \) collected from the backlog.

To determine the usefulness of our proposed method, we used the historical data as input of the \( mksched \) algorithm (Algorithm 2). This method made it possible to compare characteristics of the algorithmic approach against the manual one. Computed values (\( R_A \) to \( R_G \)) are shown in Table 6 (since \( FC \) were the same as in Table 5 they are not shown).

4.3. Analysis

The analysis goal was to compare the manual and the optimized approaches using the same input variables. The following key questions were addressed: \( Q_1 \): What are the staffing requirements over time?; \( Q_2 \): How many iterations do we need per release?; and \( Q_3 \): How buffers for contingencies are allocated?
4.3.1. Qualitative analysis

The following EDA techniques (called 4P EDA) are simple, efficient, and powerful for the routine testing of underlying assumptions [50]:

1. run sequence plot (Yk versus iteration k)
2. lag plot (Yk versus Yk−1)
3. histogram (counts versus subgroups of Y)
4. normal probability plot (ordered Y versus theoretical ordered Y)

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

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 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 asymmetry (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 1st to the 3rd quartiles indicating normal distributions. In contrast, the normality assumption is not reasonable in the lower case.

Interpreting the above plots, we 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

### Table 5

Historical release schedule values (Rk to Rei).

<table>
<thead>
<tr>
<th>FC</th>
<th>IC</th>
<th>FP</th>
<th>FS1</th>
<th>FS2</th>
<th>FS3</th>
<th>FS4</th>
<th>FS5</th>
<th>SFS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</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>117.0</td>
</tr>
<tr>
<td>Rb</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>85.5</td>
</tr>
<tr>
<td>Rc</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>137.5</td>
</tr>
<tr>
<td>Rd</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>116.5</td>
</tr>
<tr>
<td>Re</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>126.5</td>
</tr>
<tr>
<td>Rf</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>120.0</td>
</tr>
<tr>
<td>Rg</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>160.0</td>
</tr>
</tbody>
</table>

### Table 6

Optimized release plan values (Rk to Rei).

<table>
<thead>
<tr>
<th>IC</th>
<th>FS1</th>
<th>FS2</th>
<th>FS3</th>
<th>FS4</th>
<th>FS5</th>
<th>SFS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>4</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>27.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Rb</td>
<td>3</td>
<td>30.0</td>
<td>29.5</td>
<td>26.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Rc</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>29.5</td>
<td>18.0</td>
</tr>
<tr>
<td>Rd</td>
<td>4</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>26.5</td>
<td>0.0</td>
</tr>
<tr>
<td>Re</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>6.5</td>
</tr>
<tr>
<td>Rf</td>
<td>5</td>
<td>30.0</td>
<td>30.0</td>
<td>29.0</td>
<td>26.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Rg</td>
<td>6</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>30.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

We constructed \( Y_k = \sum_{j=1}^{n} w_j x_{kj} \) (i.e. sum of assigned features to iteration) as a *result variable* to answer our questions (Q1–3). First, we carried out Exploratory Data Analysis (EDA) [50] to get an insight into the data sets, then we performed descriptive statistical analysis to quantitatively compare the main properties of the two approaches. The results of the analysis are presented as follows.

Fig. 4. 4P of historical (upper) and optimized (lower) plans.
underloaded (the paddings were considerably smaller or larger than the limit).

4.3.2. Quantitative analysis

The qualitative analysis pointed out an important aspect of the algorithm usage in scope-driven situations (see Section 2.5): 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 \( R^*_C \) to \( R^*_C \) to avoid resource overload (cf. Tables 6 and 5). (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 \( (\xi_k) \), 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, we also constructed an other optimized case – denoted by \( R^*_C \) – by leaving out the last iteration from the release in order to get rid of the ineffective iteration. We consider this case as a more realistic resource utilization than \( R^*_C \). Since it avoids resource underload. As scope reduction does not affect answering the \( Q1 \) and \( Q3 \), we used this case in the comparisons also.

The previous data sets were analyzed with descriptive statistics (D1–3) (Table 7) 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:

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 \( R^*_C \) to \( R^*_C \) in other to avoid resource overload, explains the 5 value in the Min column of the \( R^*_C \) case. Contrary, in the \( R^*_C \) 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 \( R^*_C \) case (histogram in Fig. 4) can be interpreted by the capacity constraints (see (2b)) and the objective (see (2a)). Additionally, comparing the historical case with the \( R^*_C \) 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.

### Table 7
Comparison with descriptive statistics.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R^*_C )</td>
<td>18.0</td>
<td>36.0</td>
<td>-0.61</td>
<td>2.79</td>
</tr>
<tr>
<td>( R^*_C )</td>
<td>5.0</td>
<td>30.0</td>
<td>-2.40</td>
<td>7.28</td>
</tr>
<tr>
<td>( R^*_C )</td>
<td>26.0</td>
<td>30.0</td>
<td>-3.64</td>
<td>15.24</td>
</tr>
</tbody>
</table>

D3: relatively large kurtosis (measure of how outlier-prone a distribution) of the \( R^*_C \) case (histogram in Fig. 4) can also be explained by the capacity constraints (see (2b)) and the objective (see (2a)) of the optimization. Considering the \( R^*_C \) case, the higher kurtosis means less frequent extreme deviations which is economically better also.

Referring back to the Table 5, there were 4 iterations in the historical case where the resources were underloaded more than or equal to 20% \( (F_{51} \leq 24) \), there were 7 iterations where the underloads were more than or equal to 10% but less than 20% \( (24 < F_{51} \leq 27) \), and there were 2 iterations where the underloads were less than 10% \( (F_{51} < 20) \). Considering overload, there was 1 iteration in the historical case where the resources were overloaded more than or equal to 20% \( (F_{51} \geq 36) \), there were 10 iterations where the overloads were more than or equal to 10% but less than 20% \( (36 < F_{51} \leq 33) \), and there were 3 iterations where the overloads were less than 10% but the paddings were not optimal \( (27 < F_{51} < 30) \). Considering overload, there was 1 iteration in the historical case where the resources were overloaded more than or equal to 20% \( (F_{51} \geq 36) \), there were 10 iterations where the overloads were more than or equal to 10% but less than 20% \( (36 < F_{51} \leq 33) \), and there were 3 iterations where the overloads were less than 10% but the paddings were not optimal \( (27 < F_{51} < 30) \). Contrary, in the \( R^*_C \) 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 \( (\xi_k) \) of the optimization model. The Table 8 summarizes the distances \( (\Delta) \) from the maximal value \( (30) \) expressed in percentages (how many iterations from the total iterations of the given case fell in the defined intervals).

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 maximal. In contrast, in the \( R^*_C \) 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 maximal. 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 \( R^*_C \) 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 (Section 4.3.1).

As a consequence, the staffing requirements (cf. \( Q1 \)) showed more smooth and fully padded iterations (Fig. 4) in the algorithmic cases (both \( R^*_C \) and \( R^*_C \)). It means that the algorithm strives to (1) prevent resource overload – which often yields increasing delivery risks, and (2) underload – which captures 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. \( Q2 \)) 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. \( R^*_C \).

Finally, if we consider the buffer per releases (\( B_{PR} \)) 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 5. Their allocations in time were also occasional – see Run sequence plot in Fig. 4. The $R_{k+c}$ case points out the buffer allocation property of the algorithm: time buffers (cf. Q3) 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 [52].

4.4. Computational experiments

Problem solving time of the historical data sets, which can be considered as small-medium-sized problems, took less than one second with our prototypic tool. Therefore, to give some orientation about the performance of the mksched algorithm on larger problems, we carried out simulations with the guidance of [61].

To get a more nuanced picture of the algorithm we 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 ($w_i$) were uniformly distributed in the data set of {$0.5, 1, 2, 3, 8$} (reflected the typical resource demands in Story points [13]). The values ($p_i$) were expressed as a function of the resource demands – yielding typical properties of each group such a way that $p_i$ was chosen randomly from [$w_i - w_j/10, w_i + w_j/10$] 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) [60]. To enlarge the problem size, we chose total effort in each iteration ($c_i$) equal to 50 which can be considered as a typical team velocity of a large agile team (cca. 10-people) [13]. We 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 (Section 4.2).

### 4.4.1. Solving time

The behavior of the algorithm was considered for different problem sizes $n \in \{50, 100, 200, 500\}$, and it was run 30 times to calculate the mean solving time. The size 50 is similar to the historical data sets (Section 4.2), but 500 practically never occurs since it would require to plan 10 iterations ahead for a large agile team (the typical value is 4 [13]). Additionally, the algorithm were run in three cases: (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 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 9, 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 9, we can conclude that: (1) the more realistic case ($ST_{P-C}$) was acceptable since the computation time was less than 4 s even in case of large instances (500 features); (2) the $\langle P \cup C \rangle$ and $\langle C \rangle$ cases were exponential to the item count (roughly $ST \propto n^{1.1-1.5}$) as it was expected (see $O(o \times n^2)$ asymptotic upper bound in Section 3.5.2); (3) the $\langle C \rangle$ case was harder than the $\langle P \cup C \rangle$ 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 computational 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 (4b)). This extreme situations may be handled by partitioning the packages into smaller ones by diminishing the count of coupling relations. As a consequence, we can state that, our proposed 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 ($\langle P \cup C \rangle$), which does not contain our additional constraints (see Section 3.5.2), our 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 [61] (Chapter 5). Specifically, the proposed 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, our proposed algorithm requires additional iterations and administration to deal with the introduced extensions – particularly the multiple knapsack capabilities and the precedence handling (see Section 3.5.2). The additional iterations, that are emananated from the multiple knapsack capabilities, are manifested in the external loop of the algorithm (see lines 5–68 in Algorithm 2). Actually it is also expressed in the $O$ parameter of the algorithm time complexity (Section 3.5.2). Please note, the time complexity of the Horowitz–Sahni algorithm and the proposed algorithm is $O(n^2)$ [47] and $O(o \times n^2)$ (Section 3.5.2) respectively.) On the other hand, the additional administration can be explained by the precedence handling, namely maintaining ready list and scheduled-list (see lines 6 and 7), of the proposed algorithm.

Revert to the comparison of our algorithm and the state-of-the-art public knapsack algorithms, we have to underline the fact that we implemented our 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 our algorithm would not be accelerated by the compilation in that degree, our algorithm is able to handle the introduced constraints of agile release scheduling while it computes the results within an acceptable response time.

### 4.4.2. Quality of plans

The objective function (see (2a)) 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.

#### Table 9

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T_{P-C}$ (ms)</th>
<th>$ST_{P-C}$ (ms)</th>
<th>$I^-$</th>
<th>$ST^-$ (ms)</th>
<th>$T_{P-C}$ (ms)</th>
<th>$ST_{P-C}$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>234</td>
<td>90</td>
<td>16,545</td>
<td>1742</td>
<td>6443</td>
<td>651</td>
</tr>
<tr>
<td>100</td>
<td>1600</td>
<td>208</td>
<td>53,020</td>
<td>5565</td>
<td>6883</td>
<td>711</td>
</tr>
<tr>
<td>200</td>
<td>2058</td>
<td>684</td>
<td>72,880</td>
<td>7653</td>
<td>17,090</td>
<td>1812</td>
</tr>
<tr>
<td>500</td>
<td>33,164</td>
<td>4413</td>
<td>273,987</td>
<td>29,392</td>
<td>34,232</td>
<td>4116</td>
</tr>
</tbody>
</table>
Let us given a set of deliverable features $j \in \mathcal{W} : |\mathcal{W}| = n$ 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 Section 3.4). 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 4.3.2 (see Table 8) 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 ($D$) 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 ($D$) 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 Section 4.3.2).

The previous randomly generated instances of release data were analyzed to measure the quality of the release plans. We 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 ($\setminus(P \cup C)$), (2) without couplings ($\setminus C$), 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 10 summarizes the distances ($D$) in percentages. Please note, to save space in the Table, the previously defined distance intervals (see Table 8) are abbreviated as $d_1 \equiv (A > 20\%); d_2 \equiv (20\% > A > 10\%); d_3 \equiv (10\% > A > 0\%)$ and $d_4 \equiv (A = 0\%)$. Additionally, the average distance in an interval is denoted with $\overline{A}$. Considering these intervals along with the investigated parameters ($n$, $c_k$, $C$ and $P$), we can observe that:

**PQ1:** looking at the $\overline{A}$ values in the table, one can realize that $\setminus(P \cup C)$ case produced the highest value in the $d_4$ column and the lowest values in the $d_1 \_\_\_$ ones, therefore, it provided the best resource utilization. Comparing the remained cases, the $(P \cup C)$ case is better than the $\setminus C$ case. This order can be explained by the complexity of precedence handling and the different feature sizes. The $\setminus(P \cup C)$ case did not deal with any dependencies, so it enabled considerably larger search space for the algorithm. (Generally, the larger search space leads to greater variations of packing and often may lead to a better solution.) The $(P \cup C)$ case bothered with precedences and couplings where the latter one constituted fewer precedences and larger item sizes (see Section 4.4.1). 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 $\setminus(P \cup C)$, $C$ and $(P \cup C)$ cases), the value of the interval $d_4$ was between 82% and 100%. While in $n = 500$ instances, the value of the interval $d_4$ 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 ($R_{\mathcal{A}}$ case – see Table 8). We think of this type of problem as a small-medium-sized agile release scheduling problem. Actually, we 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 $c_b = 30$ and therefore, in this case, we 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, we 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. Section 4.3.2).

5. Discussion and future work

Release planning is an activity concerned with the implementa-
tion of the selected features (what aspect) in the next version of the software. Complexity of planning arises from the interaction be-
tween features by implicit and explicit constraints. While the previ-
ous is given by the scarcity of resources (available developers), the latter one is emerged from different dependencies (cf. Section 3.1).

Without dependencies, release planning can be considered as an 0-1 knapsack problem, which is NP-hard [47]. With dependencies planning becomes a more complex problem. Though, there are some proposed techniques to release planning, they do not con-
sider the staged-delivery (when) characteristics of agile environ-
ments – 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 we called it – agile release scheduling.

In agile environments, all scheduling factors ($P_1–5$ in Section 1.2) are managed informally in planning meetings. Informal ap-
proaches work well in smaller projects, however as the size and complexity increase planning becomes a very complex, time con-
suming process and advocates tool support [23,26]. To address this sit-
uation, we proposed a combinatorial optimization model and al-
gorithm as a solution.

First, we identified two release scheduling dependencies (i.e. coupling and precedence) between features (cf. Section 3.1) that af-
fect implementation sequences. We concluded that five dependen-
cies can be interpreted as release scheduling dependencies from the six types of dependencies identified in [21]. Whereas XOR is heav-
yly used in practice during requirement selection, we ignore it from our solution since release planning and scheduling practi-
cally 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 [53–55]. They can help to identify variability at the early require-
ment’s phase by capturing alternatives by which stakeholders can achieve their goals.

Secondly, to formulate the agile scheduling model, we identified the main concepts of agile scheduling (cf. Section 3.2). Although, there can be found partial conceptual models in [28,29], but a full-scaled model could not be found. So, we 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 iter-
ation scheduling applications.

We formulated agile release scheduling as an extension of the binary multiple knapsack optimization problem [47] – considering the previous factors ($P_1–5$). Our proposed solution covers wide-ranging release scheduling situations with the expression of: (1) date-driven/scope-driven scheduling (cf. Section 2.5), (2) dependencies between features (cf. Section 3.1). We called this as agile release scheduling problem (ARS). This interpretation made it possible to adapt efficient global optimization algorithms – we utilized the search concept of Horowitz–Sahni’s algorithm [47] – to solve ARS problems. The developed algorithm (mksched), in terms of (1) staffing requirements (cf. $Q_1$): 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 re-
leases (cf. $Q_2$): it pointed out that to avoid resource underload it should be used in an iterative manner by the project manager; fi-
nally, (3) release buffers (cf. $Q_3$): it supports lower level risk of delivery slippage [52]. Moreover, addressing the date-driven/ scope-driven scheduling questions (cf. Section 2.5) – contrary to the common date-driven approach (e.g. in [35]) – and the easy and fast computation – even on large problems (Section 4.4) – al-
lows (1) generating alternative schedules by utilizing what-if anal-
ysis to tailor the best schedule for the specific project context and (2) considering the stakeholders’ feedbacks by altering constraints and priorities. Both lead to more easily adaptable schedules and more informed and established decisions by selecting the best schedule from the alternatives. The developed algorithm can be downloaded from the author’s website.

However, several optimization algorithms are publicly available (see in [47,61]) 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 [47]) due to the followings:

1. Scheduling capability: Common knapsack models can calculate the next releasable items in contrast with our proposed solu-
tion, which can handle many (theoretically any) iterations immedi-
ately – due to the demand of staged-delivery approach of agile environments. As a consequence, our (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, we named this model as agile release scheduling.

2. Soft precedence: Multi-knapsack algorithms generally do not handle any precedence, so we had to complement it. This exten-
sion 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 [35] and in [28]) which ones deal with 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 imple-
mented by several optimization tools (such as CPLEX [56], AIMMS [57]), and they may provide better solving times (but not better quality since our 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 program-
ning language, and the lack of this knowledge is often a barrier to put the proposed models into practice; (2) integrating the callall 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 expen-
sive (typically $n \times $1000).
We can look at the proposed solution as an extension to the readily available agile planning tools (such as Rally [11] or XPlanner [12]) which helps collecting the planning data (features, required effort, team velocity, etc.), but they support only manual release planning. (At Multilogic the planning data were collected through an MS Sharepoint-based web site [62].) Therefore, with this extension, we believe that one can produce agile release schedule easily based on the collected data. We also expect that our proposed 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 Section 4.1). But actually, the major difference is not the required effort but the higher quality of the schedule: the optimized approach provides a higher quality schedule (i.e. avoiding under- and overload), produces better resource utilization (see Section 4.4.2), 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.

More comprehensive empirical data is necessary to evaluate the quality of our proposed agile release scheduling method, but 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. Section 3.2) points out that this approach can be complemented with an iteration-level scheduling i.e. allocating technical tasks to Developers. Additionally, since the presented solution pertains to single team release scheduling only, it can also be extended with multi-team scheduling capabilities to support distributed development situations (such as Isolated Scrum or Fully Distributed Scrum [58,59]). A complementary iteration planning method and distributed agile release planning extension can be found in [29] and in [30], and they fit well to the presented agile release scheduling approach. An integrated solution may provide not just one agile team's but many teams' scheduling, and not just at the release-level but at iteration-level also.

6. Conclusions

The three out of the four most important barriers of agile adoptions are the lack of upfront planning, the loss of management control, and the lack of predictability [1]. These barriers are closely connected with the present practice of agile planning. These critics initiated our endeavor to diminish these barriers. First, we provided a proposed conceptual model of agile planning that defines a common language to avoid the different interpretations that are used in the field. This model not only helps to identify and define the main notions but with its precise relationships it can also be used in the design of agile planning applications.

The present practice of agile release planning is unsatisfactory due to lack of solid theoretical basis and existence of a number of competing practices and fads. Therefore, as a second aim, we proposed a theoretically sound method for agile release planning. In agile environments, scheduling features into the upcoming delivery stages is a complex, informal and mainly manual process. Up to our best knowledge, the proposed staged-delivery global optimized model formulation is novel in the field. Although, there are some tenets to manual scheduling [3,13] algorithmic solutions could not be found. The proposal gives the main parameters of the typical agile planning space (such as objectives, constraints) and presents an optimization model that can be realized by optimization tools or by implementing our suggested custom-made algorithm. To evaluate our model simulations were carried out seven real life and several generated data sets that demonstrated that the method could easily cope with the previously manually managed scheduling factors (P1–5) besides providing optimized schedules. Additionally, this approach provides more informed and established decisions with application of what-if analysis (rescheduling the release by altering its parameters), and mitigates risks by providing more smooth feature allocation and moving buffers to the end of releases. We believe the results are even more impressive in more complex (more constraints, features, etc.) situations.

With the results of experiments on real life and generated data sets indicate that our approach can provide practical value as a decision support tool for product and development project managers. We believe these results may help to diminish the previously mentioned barriers of agile adoption.

Acknowledgements

The author would like to thank the associate editor and the anonymous referees for their valuable comments and suggestions. Discussion with colleagues at Multilogic also helped to provide a more readable content.

References
