An Integrated Approach for Requirement Selection and Scheduling in Software Release Planning.

Article *in* Requirements Engineering · November 2010

DOI: 10.1007/s00766-010-0104-x · Source: OAI

Requirements Eng (2010) 15:375–396
DOI 10.1007/s00766-010-0104-x

ORIGINAL RESEARCH

An integrated approach for requirement selection and scheduling in software release planning

Chen Li · Marjan van den Akker · Sjaak Brinkkemper · Guido Diepen

Received: 3 April 2009 / Accepted: 6 April 2010 / Published online: 5 May 2010

Abstract It is essential for product software companies to decide which requirements should be included in the next release and to make an appropriate time plan of the development project. Compared to the extensive research done on requirement selection, very little research has been performed on time scheduling. In this paper, we introduce two integer linear programming models that integrate time scheduling into software release planning. Given the resource and precedence constraints, our first model provides a schedule for developing the requirements such that the project duration is minimized. Our second model combines requirement selection and scheduling, so that it not only maximizes revenues but also simultaneously calculates an on-time-delivery project schedule. Since requirement dependencies are essential for scheduling the development process, we present a more detailed analysis of these dependencies. Furthermore, we present two mechanisms that facilitate dynamic adaptation for over-estimation or under-estimation of revenues or processing time, one of which includes the Scrum methodology. Finally, several simulations based on real-life data are performed. The results of these simulations indicate that requirement dependency can significantly influence the requirement selection and the corresponding project plan. Moreover, the model for combined requirement selection and scheduling outperforms the sequential selection and scheduling approach in terms of efficiency and on-time delivery.

Keywords Requirement selection · Requirement scheduling · Release planning · Requirement dependency · Integer linear programming (ILP) · Simulation · Scrum

C. Li (✉)
Information Systems Group, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: lic@cs.utwente.nl

M. van den Akker · S. Brinkkemper
Department of Information and Computing Sciences,
Utrecht University, Utrecht, The Netherlands
e-mail: j.m.vandenakker@cs.uu.nl

S. Brinkkemper
e-mail: s.brinkkemper@cs.uu.nl

G. Diepen
Paragon Decision Technology, Haarlem, The Netherlands
e-mail: Guido.Diepen@aimms.com

1 Introduction

Deciding on the requirements for an upcoming software release is a complex process [1]. With the evident pressures on time-to-market [2, 3] and limited availability of resources, often there are more requirements than can actually be implemented. The market-driven requirement engineering processes [4] have a strong focus on requirement prioritization [5]. The requirement list needs to fulfill the interests of the various stakeholders and takes many variables into consideration. Several scholars have presented lists of such variables, including: importance or business value, stakeholder preference, cost of development, requirement quality, development risk and requirement dependencies [6, 7, 8, 9, 3].

In order to deal with this multi-aspect optimization problem, several techniques have been applied. The analytical hierarchy process (AHP) [5, 2] assesses requirements by examining all possible requirement pairs and matrix calculations to determine a weighted list. Jung [10] extended the AHP approach [5] by using integer linear programming (ILP) to reduce the complexity of AHP and made it applicable to large amounts of requirements. The

release planning problem was modeled as a multi-dimensional knapsack problem, so that the total revenue is maximized against the available resources. Carlshamre [6] also used ILP, based on which a release planning tool was built and added requirement dependencies as an important aspect in release planning. Ruhe and Saliu [11] describe a method based on ILP which embraces stakeholder's opinions for release planning. Van den Akker et al. [12, 13] further extended the ILP technique by including different management steering mechanisms such as deadline extensions, team transfers or hiring external personnel and by including requirements dependencies into the model and ran simulations to test the influence of each mechanism. The complexity of ILP is $\mathcal{NP} - hard$. It is believed that any algorithm that solves the problem to full optimality has a running time exponential to the problem size. We refer to Wolsey [14] for a thorough presentation on ILP.

Besides ILP techniques, the cumulative voting method [15] allows different stakeholders to assign a fixed number of points among all requirements, and an average weighted requirement list is constructed. Ruhe and Saliu [11] provide a method called EVOLVE to allocate requirements to incremental releases. This approach is further extended with Genetic Algorithm to handle dynamic resource allocations in a software release [16]. Berander and Andrews [17] provide an extensive list of requirement prioritization techniques.

Scheduling of the requirements development, i.e., determining a time schedule for the work in the development teams, is also identified as an important aspect in this field [7]. Unfortunately, few prioritization methods have taken this into account. In many cases, scheduling of requirements development is considered to be the next step after requirement selection [6], and the selection and scheduling processes are often used iteratively to find a group of requirements with an on-time delivery project plan [1]. In [16], a Genetic Algorithm is used to determine a development schedule after the requirements have been fixed to a release. However, this approach handles requirement scheduling at a lower granularity level and solve resource constraint in a rather flexible way. Compared to the extensive research on requirement selection, very little research has been performed on the scheduling part. Given the fact that 80% of software projects are late and/or over budgeted [18], a precise project plan that can help to synchronize the development teams is essential. A traditional way of project planning would be to compute the critical path on the basis of the precedence relations between the development tasks, commonly depicted in Gantt charts. However, this does not guarantee that the team capacities or expertise are respected.

In addition, requirements are hardly isolated islands but interdependent with each others. Carlshamre has identified that in fact 80% of the requirements are interdependent, and there are only a few are singular ones [19]. These dependencies significantly increase the complexity of making a project plan.

Table 1 Example requirements sheets of a release planning problem

Release Definition 5.1

Nr.	Requirement	Preceded by	Revenue	Total man days	Team A	Team B	Team C
12	Authorization on order cancellation and removal	25,63	192	400	40	0	360
34	Authorization on archiving service orders		96	96	16	40	40
63	Performance improvements order processing		160	120	120	0	0
25	Inclusion graphical plan board		800	560	80	80	400
43	Link with Acrobat reader for PDF files	25	80	264	0	264	0
75	Optimizing interface with international Postal code system	25	80	120	0	0	120
35	Adaptations in rental and systems		280	320	0	160	160
66	Symbol import		40	80	80	0	0
67	Comparison of services per department		80	272	0	72	200
	Total		1808	2232	336	616	1280
	Available resources (number of developers)			24	8	8	8
	Available team capacity for release			1440	480	480	480
	Release duration				60 days		

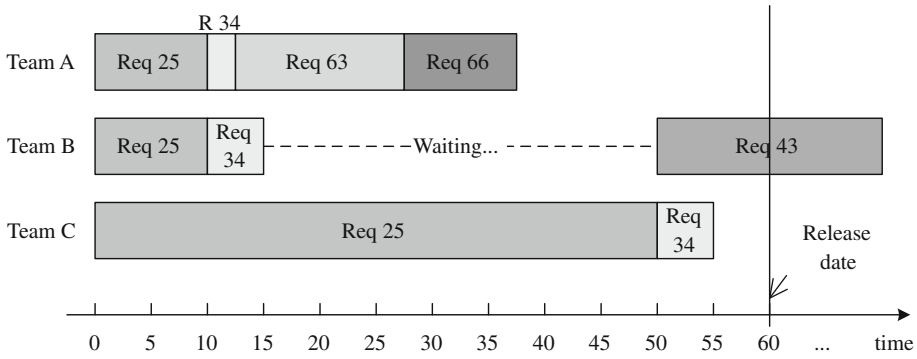

Fig. 1 A numerical example of requirement scheduling problem

1.1 Problem illustration

Table 1 depicts a simplified representation of a typical release planning problem. Nine requirements with estimated revenues (in Euros), cost (measured by man days in different teams) and dependencies are listed in the repository. In the context of this paper, we use the absolute value (expected revenue in Euros) to evaluate the importance of each requirement. It is also possible to use relative values like in AHP, where each requirement has a relative value between 0 and 100 [5] or has a weighted importance based on different stakeholders' opinions [11].

The composition of teams is defined up-front. Each team has its own special knowledge and can only contribute to specific requirements. In international organizations where this study is performed, development teams are also formed based on geographic regions.[1]

Therefore, every team is independently responsible for completing its contribution of a requirement as a whole, i.e., it is responsible for the design, realization and testing for a particular part of a requirement due to its high efficiency of knowledge exploitation [13, 20]. The development activities for a particular requirement in the teams are also decided up-front whenever a requirement is elicited. In this way, each team can work independently on a particular requirement without waiting for others. To evaluate the efforts of developing a requirement, we refer to [21, 22] for techniques based on XML use cases and feature estimations.

Given a predefined release date, the available resources in different teams are limited within the given period. In our example, teams B and C are over-loaded, while team A still have room for additional work. Based on the available resources in different teams, the revenue of each requirement and the requirement interdependencies, the best set of requirements for a next release needs to be determined. Using the existing ILP technique [5, 6, 12, 13], five requirements are selected (marked in bold) so that the total revenue is maximized against the available resources and requirement dependencies.

The next step is to schedule the selected requirements exactly in time, i.e., we need to determine the *time interval* for developing each requirement in each development team. Here, we have to deal with dependencies that result in restrictions in time. For example, requirements pertaining to foundational software components often need to be implemented before others. Similarly, certain capabilities (for example, quality issues like safety and security) need to be architected and built into the system at an early stage rather than added on later during development [7]. Therefore, an optimal implementation order of the requirements is desired.

We now formally define precedence constraints. There is a precedence constraint between R_j and R_{j*}, denoted as $R_j \prec R_{j*}$, if requirement R_{j*} can only start when requirement R_j is completely finished. Usually, precedence constraints result from dependencies. It is clear that a precedence constraint can influence the development sequence of the requirements. However, now an important question is: since we have already selected the requirements based on the available capacity, why will the precedence constraints still influence the project deadline for the release?

When there are precedence constraints and several development teams, scheduling requirements becomes complex. Figure 1 provides an example of a time schedule for the release planning problem set out in Table 1.

From Fig. 1, it is clear that although the requirement selection respects the team capacity constraints, the project will still be delayed. The reason is that there is an `implication` dependency and hence a `precedence constraint` (see Sect. 1 for a detailed analysis of requirement dependencies and precedence constraint) between requirements 25 and 43. Although team B finishes its task for R25 at day 10, it cannot start to develop R43 (which is dependent on R25's completion), because R25 is only available at day 50 when team C finishes its job. So, between day 10 and day 50, team B only needs 5 days for R34, and the remaining 35 days are wasted on waiting for team C. When R25 is finally

[1] It is possible to optimize the team composition using the concept of "team transfer", i.e., we can transfer developers to the teams which are over-loaded [13].

available at day 50, it takes team B another 33 days to develop R43, so the earliest date to finish the whole project is at day 83 instead of the expected release date day 60. Clearly, the time wasted on synchronization is undesirable. This example reveals that precedence relations may have a strong impact on the project planning.

1.2 Overview of the paper

The above example raises an important question of how to design a schedule in which teams utilize available time efficiently without waiting for others. Or if waiting time cannot be avoided, how to minimize such waiting time and also minimize the duration of the complete release project?

Another question is: if we need to spend too much time on waiting, is it possible to re-select requirements so that the release plan meets a predetermined deadline? For example, in the case described above, if we still want to keep the 60 days as the deadline, then we need to re-select the requirements so that the newly selected requirements can be implemented within the required time span. For this case, R43 would have to be dropped or replaced by others to keep the project on time.

In addition, as factors like revenue, release date and dependencies are based on estimation, they are often over-estimated or under-estimated. Obviously, pursuing preciseness of these factors is a difficult and luxurious goal. We therefore need to find out which factors have relatively high impacts on the results so that we can focus on the estimation of these factors. Or alternatively, we should allow run-time modifications if actual values turn out to deviate from estimated values.

In this paper, we focus on solving the three problems mentioned above. These can be expressed as follows. Under the circumstances that there are different development teams involved in the release planning and there are requirement dependencies between the requirements:

1. How should we schedule the requirements to minimize the project lead time, i.e. the completion time of the project?
2. How should we integrate the requirement selection and scheduling so that the revenue is maximized and the project plan remains on schedule?
3. What are the influences of different factors on the requirement selection and scheduling, and how to handle over-estimation or under-estimation which are revealed after the project has already started?

The goal of this paper is to provide mathematical models that can assist to determine the requirement selection and scheduling for the next software release. Like any planning, careful estimation of the input factors, like revenue, cost and dependencies, is key to success. We are also fully aware of the facts that in real world, many psychological, political and personality factors can influence the right choices. Decision making cannot be purely mathematical; however, mathematical models can be considered as a useful means of decision support.

The remainder of the paper is organized as follows. In Sect. 2, we first introduce precedence constraints and describe the relationship between precedence constraints and the requirement dependencies. Section 3 provides three ILP models, one for requirement selection, one for requirement scheduling and one model for combined requirement selection and scheduling. In Sect. 4, the combined selection and scheduling model is extended by requirement dependencies. Section 5 shows our prototype implementation of the models and presents two simulation experiments: the first one examines the influence of precedence constraints on requirement scheduling, and the second one compares the sequential requirement selection and scheduling approach with the combined approach. We also analyze the influences of different input parameters at the end of this section. In Sect. 6, we present two mechanisms that allow managers to adaptively handle dynamic changes. The first one provides a method to dynamically modify parameter estimations during run-time, and the second one applies Scrum concept in agile software development method [23, 24]. We conclude the paper and suggest future research directions in Sect. 7.

This paper significantly extends our work presented in [25] and provides more technical details and simulation results. For example, we provide extensions of our model to handle requirement dependencies (c.f. Sect. 4) and also introduce two mechanisms to adaptively handle dynamic changes(c.f. Sect. 6). Additional simulation results are also depicted in Sect. 5.

2 Preliminary analysis of the problem conditions

2.1 Precedence constraints and requirement dependencies

Carlshamre et al. [19] identify six types of requirement dependencies for release planning: (1) `Combination`: two requirements are to be implemented jointly; (2) `implication`: one requirement requires another one to function; (3) `Exclusion`: two requirements conflict with each other. (4) `Revenue-based` and (5) `cost-based` dependencies mean one requirement influences the revenue/cost of another; and (6) `Time-related` dependency means one requirement needs to be implemented after another.

Table 2 presents the influence of dependencies on requirement selection and scheduling. The first five dependencies have been identified as important factors for requirement selection [6, 12].

Table 2 The influences of requirement dependencies on requirement selection and scheduling

Dependency group	Dependency type	Influence requirement selection	Influence requirement scheduling
Functional dependency	Combination	✔	
	Implication	✔	✔
	Exclusion	✔	
Value-related dependency	Revenue-based	✔	
	Cost-based	✔	✔
Time-related dependency	Time-related		✔

With respect to time, some of the dependencies can not only influence the requirement selection, but will also influence the requirement scheduling. For example, if requirement R_{j*} requires R_j to function, it is normally better to start develop R_{j*} after R_j is finished; or if requirement R_j influences the implementation cost of requirement R_{j*}, it is also considered better to implement R_j first (see [6]). So, together with the explicitly mentioned `time-related` dependency, the `implication` and `cost-related` dependencies also imply precedence constraints during scheduling. Hence, when scheduling the requirements, we should take three out of six types of requirement dependencies into consideration. A more detailed discussion will be given in Sect. 4.

2.2 Two straightforward cases

Figure 1 illustrated the scheduling problem when there are precedence constraints and different development teams. However, scheduling will not pose a problem if there are no precedence constraints between requirements. Because each team works independently, and no synchronization is needed, each development team can perform its tasks in any order it prefers. In this way, scheduling becomes straightforward, and the deadline will not be exceeded.

Similarly, if there are precedence constraints but no team or task division, scheduling the activities is not a difficult issue as well. Now we can apply the traditional *critical path* approach. We first create a directed acyclic graph (DAG) G by setting the requirements R_j as vertexes and the precedence constraints $R_j \prec R_{j*}$ as a directed edges (R_j, R_{j*}). Then any topological sort of the directed acyclic graph results in a feasible schedule [26]. This sort provides a linear order of all the vertices such that if G contains an edge (R_j, R_{j*}), then R_j appears before R_{j*}. We can compute this sort in $\mathcal{O}(N+E)$ time where N equals the number of requirements, and E equals the number of dependencies. Because the development proceeds continuously without interruption, the release deadline can also be met.

3 ILP models for software requirement selection and scheduling

In this section, we present three integer linear programming models for software release planning. In Sect. 3.1, we briefly review the Knapsack model that was developed in earlier work. In Sect. 3.2, we show the scheduling model that can minimize the project time span with the precedence and resource constraints. In Sect. 3.3, we show a combined selection and scheduling model that can not only maximize the revenue based on the precedence and resource constraints, but also provides a schedule for on-time delivery of the project.

In this section, all parameters, e.g., expected revenues, requirement dependencies, resource constraints, are considered to be fixed. Handling the dynamic character of these parameters will be analyzed later in Sects. 5 and 6.

3.1 Knapsack model for requirement selection

We are given a set of n requirements $\{R_1, R_2, \ldots, R_n\}$ with expected revenue v_j of requirement R_j. Let m be the number of teams $G_i (i = 1, 2, \ldots, m)$. The development activity by team G_i for requirement R_j is considered as one individual job, i.e., each team works on one requirement independently from the other teams, and there are no predefined time restrictions between the jobs within a requirement (c.f. Sect. 1.1). Let us define a set $X = (J_1, J_2, \ldots, J_k)$ of all the jobs with non-zero development time, and there are k ($k \leq m \times n$) jobs in the set.

Because each job belongs to only one requirement, we can partition the set X into n disjoint subsets $\{X(R_1), X(R_2), \ldots, X(R_n)\}$ where $X(R_j) = \{J_k \mid \text{job } J_k \text{ is for requirement } R_j\}$, $(j = 1, 2, \ldots, n)$. Similarly, one job only belongs to one team, so we can also partition the set X into m disjoint subsets $\{X(G_1), X(G_2), \ldots, X(G_m)\}$ where $X(G_i) = \{J_k | \text{ job } J_k \text{ is in team } G_i\}$, $(i = 1, 2, \ldots, m)$.

Each job $J_k \in X(R_j) \bigcap X(G_i)$ is associated with a parameter a_{ij} defined as the amount of man days needed for Requirement R_j in team G_i. Assume the number of developers in team G_i is Q_i. Now the development time d_k for job J_k is $\frac{a_{ij}}{Q_i}$. Here, we assume that as soon as a team starts working on a job, they will continue working on it until the job is completely finished.

The planning period for the next release is T, and the total amount of working days within this planning period is denoted by $d(T)$ days.

A known ILP model for release planning is the Knapsack model [6, 10, 11, 12]. The objective of the Knapsack model to maximize the revenue subject to the constraints of limited capacity of each development team within the planning period, i.e. we want to include as many requirements in the "Knapsack" as possible to get maximal value.

To achieve this, we define a binary decision variable x_j associated with each requirement R_j, where $x_j = 1$ if requirement R_j is selected and $x_j = 0$ if it is not. The Knapsack model is formulated as follows:

$$\max \sum_{j=1}^{n} v_j x_j \tag{1}$$

Subject to:

$$\sum_{j=1}^{n} a_{ij} x_j \leq d(T) Q_i, \qquad \text{for all } i \in \{1, 2, \ldots, m\} \tag{2}$$

$$x_j \in \{0, 1\} \qquad \text{for all } j \in \{1, 2, \ldots, n\} \tag{3}$$

In this model, (1) is the objective function stating that we want to maximize revenue. Constraint (2) shows that the capacity in each group G_i is limited to the total amount of man days available within the planning period, i.e., no more than $d(T)Q_i$. If the company decides that some of the requirements have to be included in the new release in any case, we can add one more constraint stating that $x_j = 1$ if requirement R_j is fixed. A comprehensive description of Knapsack problem can be found in [27].

As we illustrated in Sect. 1, the Knapsack model cannot guarantee that the project can be finished on time, since precedence constraints between requirements can cause additional waiting time (c.f. Fig. 1). In the next section, we will show a model that can minimize the waiting time by minimizing the project makespan.

3.2 RCPSP model for requirement scheduling

To schedule the requirements exactly in time, two issues need to be considered: the limited resource availability and the existence of precedence constraints between the requirements. Within scheduling theory, the problem can be characterized as a special case of the resource constraint project scheduling problem (RCPSP) [28] (see e.g. [29] for a comprehensive description). It is a special case because the resources all have a capacity of 1 and the resources do not have to work on a requirement R_j simultaneously.

RCPS constitutes a class of $\mathcal{NP} - hard$ scheduling problems [30].

The problem complexity and its practical relevance have inspired many scholars to develop algorithms to tackle different variants of this problem. Methods like heuristic search [31], exact algorithms [32], genetic algorithms [33], evolutionary algorithms [34], column generation [35] and so on have been proposed to solve this hard problem (see [29, 36] for an overview). In this paper, we present an ILP model of the RCPSP formulation of our problem, inspired by the model in [28].

3.2.1 The precedence constraints

We define a set $A = \{(R_j, R_{j*}) | R_j \prec R_{j*}\}$ which contains all the precedence constraints. We define the set H to show the precedence relationship between jobs:

$$H = \{(J_k, J_{k*}) | J_k \in X(R_j), \qquad J_{k*} \in X(R_{j*}), (R_j, R_{j*}) \in A\}$$

In this way, we set all the jobs of requirement R_{j*} as the successors of the jobs of requirement R_j, and we make sure that any job for requirement R_{j*} can only start after all the jobs for requirement R_j are finished.

We also need to introduce two virtual jobs, the start of the project and the end of the project. The job *START* must finish before starting the jobs in X, and the job *END* can only start when all the jobs are finished. The processing time of these two virtual jobs is 0. We define a new job set $X' = X \bigcup \{START, END\}$, which includes all jobs in X and the two additional virtual jobs *START* and *END*.

If job J_k does not have any successor, then we add (J_k, END) to H. Or if job J_k does not have any predecessor, then we put $(START, J_k)$ in H. The precedence relations between jobs can be represented by a directed acyclic graph $G = (X', H)$.

3.2.2 The upper bound of the project span

Let $T_{\max}$ be an upper bound on the project span. We can compute $T_{\max}$ using $\sum_{j=1}^{n} \max(d_k | J_k \in X(R_j))$. The upper bound corresponds to developing the requirements one after another, i.e. without any time overlap between different requirements.

3.2.3 The earliest start time es_k and the latest start time ls_k of each job J_k

For each job J_k, we can compute es_k (earliest possible start) and ls_k (latest possible start) as its time window to start. To compute this, we first topologically sort the jobs, so that job J_k is before job J_{k*} in the order if $(J_k, J_{k*}) \in H$.

We can use a longest path algorithm in a Directed Acyclic Graph to compute es_k [26]. First, set $es_{START} = 0$, then compute es_k as the longest path from START to J_k. Similarly, we can compute the latest start ls_k using a longest path algorithm (backward recursion). First, set $ls_{END} = T_{\max}$ and then compute ls_k as $T_{\max}$ minus the longest path from *END* to J_k.

3.2.4 The integer linear programming model

For the integer linear programming model, we use a time-indexed formulation. This formulation has successfully been applied for machine scheduling problems and is known to have a strong LP relaxation lower bound (see e.g. [37, 38]). We discretize time, and the integer time t represents the period of $[t, t + 1)$. For each job J_k, we define a group of

variable ξ_{kt} within the time interval $[es_k, ls_k]$, where t is the possible time for J_k to start. Now ξ_{kt} is a binary variable that equals 1 if and only if J_k starts at the beginning of period t. Then, we can formulate the problem as follows:

$$\min \sum_{t=es_{END}}^{t=ls_{END}} t \cdot \xi_{ENDt} \tag{4}$$

Subject to:

$$\sum_{t=es_k}^{t=ls_k} \xi_{kt} = 1, \quad \text{for all } J_k \in X' \tag{5}$$

$$\sum_{t=es_k}^{t=ls_k} t \cdot \xi_{kt} + d_k \leq \sum_{t=es_{k*}}^{t=ls_{k*}} t \cdot \xi_{tk*}, \quad \text{for all } (J_k, J_{k*}) \in H \tag{6}$$

$$\sum_{J_k \in X(G_i)} \sum_{\tau=\sigma(t,k)}^{t} \xi_{k\tau} \leq 1, \quad \text{for all } t \in \{0,1,\ldots,T_{\max}\}, i \in \{1,\ldots,m\} \tag{7}$$

$$\xi_{kt} \in \{0,1\} \quad \text{for all } t \in [es_k, ls_k], J_k \in X' \tag{8}$$

In constraint (7), $\sigma(t, k) = \max(0, t - d_k + 1)$. Constraint (4) shows the objective to minimize the project span. Constraint (5) shows that each job is started exactly once. Constraint (6) models the precedence constraints: a requirement can only start after its predecessors are finished. Constraint (7) makes sure that a development team can only develop at most one job at one time. Please note that if we ignore constraint (7), this model turn to be another representation of the longest path algorithm [26].

The strict precedence constraints can also be generalized, so that a certain degree of overlapping between predecessors and successors becomes possible (see for example Ref 43 in Fig. 1). Instead of enforcing that a job only starts after the completion of all its preceding jobs, we can define a minimum time lag between the starting time of jobs. For example, consider the situation that job J_k precedes job J_l, and both of the two jobs take 5 days to complete. Instead of setting strict precedence relationship between them, we allow job J_l to start if the first 40% of job J_k is finished, i.e., we allow J_l to start at least 2 days after J_k starts. In this case, the value d_k in constraint (6) has to be changed to the minimum time lag (2 days in this case). In a similar way, we can define maximum time lags, so that after one job is finished the start of its succeeding jobs should not be delayed too much (e.g., if the work for both jobs contains a lot of similar issues). However, these are not so critical. We refer [35] for details.

3.3 Combining requirement selection and scheduling

Although the Knapsack model guarantees that the amount of work corresponding to the selected set of requirements fits in the teams' capacity, it is still possible that the selected set of requirements cannot be scheduled within the required lead time (c.f. Sect. 1). The RCPSP model described in Sect. 3.2 can help to minimize the project makespan, but it cannot guarantee that the computed schedule completes within the required lead time. In most of the software development process models, selection and scheduling are performed iteratively until an optimum solution is found [1]. However, doing this iteratively is not only difficult but also time-consuming, because we constantly need to repeat the following three steps:

1. Drop some requirements so that the project plan is fit.
2. Re-fill in some requirements to take up the freed capacity.
3. Re-make project plan for the new group of requirements.

Because the Knapsack and RCPSP problem are both $\mathcal{NP} - hard$, i.e., in principle the time needed to solve the problem is exponential to problem size, without a proper search algorithm, it is very difficult to find a solution that can fulfill the goals of maximizing revenue and on time delivery. Even if such a search method is found, continuously solving these two $\mathcal{NP} - hard$ problems will be very time-consuming. A better method is needed to solve this problem.

In this section, we will present a new ILP model that enables us to achieve the goals of maximizing revenue and on time delivery simultaneously. In the following, we will present a model for combined selection and scheduling of the requirements when a fixed project deadline is given. Similar to previous sections, we assume we can only select and start implementing a requirement if all its predecessors have been implemented.

3.3.1 *The precedence constraints*

We can handle the precedence constraints similarly to Sect. 3.2. We can define a set $A = \{(R_j, R_{j*}) | R_j \prec R_{j*}\}$ which contains all the precedence constraints. We define the set H to show the precedence relationship between jobs:

$$H = \{(J_k, J_{k*}) | J_k \in X(R_j), \quad J_{k*} \in X(R_{j*}), (R_j, R_{j*}) \in A\}.$$

In this way, we set all the jobs of requirement R_{j*} as the successors of the jobs of requirement R_j, and we can make sure that any job for requirement R_{j*} can only start after all the jobs for requirement R_j are finished. The precedence relationships between jobs can be represented by a directed acyclic graph $G = (X', H)$.

3.3.2 *The earliest start time es_k and the latest start time ls_k of each job J_k*

For the earliest start es_k, we can also use the longest path algorithm from Sect. 3.2. The only difference is since we

do not have the virtual job *START* any more, we need to set the earliest start $es_k = 0$ for all the jobs that do not have predecessor. We can apply this lower bound because a requirement can only be selected and developed when all its predecessors are selected and developed.

For the latest start ls_k, it equals $d(T) - d_k$. Please note that the method to compute ls_k is significantly different from the scheduling model. We cannot lower this upper bound because we do not know whether the successors of a job will be selected.

It is possible that $ls_k < es_k$ for a certain job J_k. It then means the job cannot fit in the project time span. So the requirement R_j that contains this job will also not be a candidate of the next release. Hence, we can eliminate these requirements beforehand and define a set X'' that contains only the feasible jobs.

3.3.3 The integer linear programming model

Like in [12], for each requirement R_j, we define a binary decision variable x_j, where $x_j = 1$ if and only if requirement is selected. Moreover, for each job $J_k \in X''$, we define a group of binary decision variables ξ_{kt} within its possible time interval $t \in [es_k, ls_k]$, where $\xi_{kt} = 1$ if and only if job J_k starts at time t.

We can now model the combined selection and scheduling problem as follows:

$$\max \sum_{j=1}^{n} v_j x_j \tag{9}$$

Subject to:

$$\sum_{t=es_k}^{t=ls_k} \xi_{kt} = x_j, \quad \text{for all } J_k \in X(R_j), j = 1, \ldots, n \tag{10}$$

$$x_{j*} \leq x_j, \quad \text{for all } (R_j, R_{j*}) \in A \tag{11}$$

$$\sum_{t=es_k}^{t=ls_k} t \cdot \xi_{kt} + d_k \leq \sum_{t=es_{k*}}^{t=ls_{k*}} t \cdot \xi_{tk*} + (1 - x_{j*}) \cdot d(T), \quad \text{for all } (J_k, J_{k*}) \in H, J_{k*} \in X(R_{j*}) \tag{12}$$

$$\sum_{J_k \in X(G_i)} \sum_{\tau=\sigma(t,k)}^{t} \xi_{k\tau} \leq 1, \quad \text{for all } t \in \{0, 1, \ldots, T_{\max}\}, i \in \{1, \ldots, m\} \tag{13}$$

$$\xi_{kt}, x_j \in \{0, 1\} \quad \text{for all } t \in [es_k, ls_k], J_k \in X'', j \in \{1, \ldots, n\} \tag{14}$$

where in constraint (13), $\sigma(t, k) = \max(0, t - d_k + 1)$. The objective function (9) models that we want to maximize the revenue. Constraint (10) means that a requirement is selected if and only if all its jobs are planned. Constraints (11) and (12) deal with the precedence constraints. Constraint (11) ensures that a requirement is only selected when its predecessors are selected. Constraint (12) guarantees that the jobs for the successor requirement can only start after all the jobs for its preceding requirements are finished. Please note that this constraint is different from the precedence constraint modeled in Sect. 3.2 (c.f. constraint (6)), because the successor job is not guaranteed to be selected. Constraint (13) is the resource constraint that one team is only able to develop one requirement at a time. Constraint (14) is the {0, 1} constraint for all the variables.

Note that if we ignore the precedence constraints (11) and (12), it is another way to represent the multi-dimensional Knapsack problem.

3.4 Reflections

In this section, we have presented three ILP models for requirement selection and scheduling: the Knapsack model for requirement selection, the RCPSP model for requirement scheduling, and a model for combined requirement selection and scheduling.

The three models are not independent. For example, if we ignore constraint (11) and (12), the combined model is then another way to represent the Knapsack model. If we compare the RCPSP scheduling model and the combined model, we see that they are very similar in handling the precedence constraints.

Besides the basic models, we have also built several extensions to model some management steering mechanisms. For example, in the Knapsack model, we included deadline extensions, which can extend the deadline T with certain cost. Moreover, we included hiring external resources or transferring people, which allows people to be transferred to other teams with a certain capability reduction [12]. Moreover, we have modeled requirement dependencies in the Knapsack model. For the combined model, we can also model different time availability in different groups to synchronize their works [39]. We ignore the details here due to the limited spaces. In the next section, we further extend the combined model for requirement selection and scheduling by modeling requirement dependencies and allowing combinatory use of requirement dependencies.

4 Requirement dependencies for the combined selection and scheduling model

As introduced in Sect. 2, requirements are hardly isolated islands but dependent on each others. Carlshamre has found that the majority of requirements are interdependent with each other, and only a few are singular ones [19]. Most of the requirement dependencies can already be modeled in the Knapsack model [6, 11, 12]. However this is not complete since Knapsack model cannot handle time-related dependencies or precedence constraints (c.f. Fig. 1).

The combined requirement selection and scheduling model (c.f. Sect. 3.3) provides an opportunity to include the precedence constraints. The reason is that the combined model does contain not only variables x_j that determine the selection of a requirement R_j but also variables ξ_{kt} that determine the time when a job J_k starts. The six types of requirement dependencies (c.f. Table 2), as introduced in [19], can now be modeled as follows.

4.1 Modeling requirement dependencies in the combined model

The `combination`, `implication`, `exclusion` and `revenue-based` requirement dependencies can be modeled the same way as in the Knapsack model [6, 11, 12]. Only the `cost-related` dependency needs to be modeled differently. Apparently, the `time-related` dependency can only be modeled in the combined model but not in the Knapsack model. For the sake of completeness, we will model all six types of requirement dependencies in this section.[2]

4.1.1 Combination

This dependency means requirement R_i requires requirement R_j, and R_j requires R_i as well. So, we should have either both of them or none of them. This can be modeled by adding one additional constraint:

$$x_i = x_j \tag{15}$$

4.1.2 Implication

This dependency means requirement R_i requires requirement R_j to function, but not vice-versa. So, we can only select R_i when R_j is selected. This can be done by adding one more constraint:

$$x_i \leq x_j \tag{16}$$

At this moment, we assume that the `implication` dependency only concerns the logical relationship between two requirements, but not the corresponding precedence relation in time. A detailed discussion will be given later in this section.

4.1.3 Exclusion

This dependency means we need either R_i or R_j, but it does not make sense to have both. It is also possible that we need neither of them. To model this type of dependency, we can set one additional constraint:

$$x_i + x_j \leq 1 \tag{17}$$

4.1.4 Revenue-based

This dependency means that requirement R_i affects the value of requirement R_j. In this case, if R_i is selected, the value of R_j will change, either positively or negatively. We assume that R_i increases the value of R_j by B_{ij} (B_{ij} is negative if R_i decreases the value of R_j). We need to introduce a new variable z_{ij} that measures whether both R_i and R_j are selected and add $B_{ij}z_{ij}$ to the objective function (c.f. formula (9)), i.e. the objective function should be $\sum_{j=1}^{n} v_j x_j + B_{ij}z_{ij}$. Finally, we add the following constraint:

$$z_{ij} \leq (x_i + x_j)/2 \qquad \text{if } B_{ij} > 0 \tag{18}$$

$$x_i + x_j - 1 \leq z_{ij} \qquad \text{if } B_{ij} < 0 \tag{19}$$

4.1.5 Cost-based

This type of dependency means that requirement R_i affects the development cost of requirement R_j. However, in the combined model, we assume the development time d_k for job J_k is a fixed number. The cost-based requirement dependencies will change this assumption since development time d_k for job J_k is not deterministic but influenced by other requirements. Replacing constant d_k by a variable will turn our model into a non-linear one and will enormously complicate constraint (12). To maintain the linearity, we need to model it differently.

Assume that requirement R_i influences the implementation cost of requirement R_j, the development cost of the jobs $J_k (J_k \in X(R_j))$ for requirement R_j will change from d_k to $d_{k[i]}$ man days after R_i is implemented. So we can define a virtual requirement $R_{j[i]}$ corresponding to R_j, which is obtained by changing the duration d_k of the job J_k in $X(R_j)$ to $d_{k[i]}$ for each $J_k \in X(R_j)$. In this way, this virtual requirement $R_{j[i]}$ can represent requirement R_j after taking the influence of requirement R_i into consideration. We can now analyze the relationship among R_i, R_j and the virtually created requirement $R_{j[i]}$.

1. If we want to obtain the cost benefit, i.e., to have requirement $R_{j[i]}$, we must have requirement R_i first. This means requirement $R_{j[i]}$ has `implication` dependency on requirement R_i, i.e., $x_{j[i]} \leq x_i$.
2. If we have selected requirement R_i, then we cannot select requirement R_j any more, because the requirement R_i will change the development cost of R_j and turn R_j to $R_{j[i]}$. This means Requirement R_i has `exclusion` dependency on requirement R_j, i.e., $x_i + x_j \leq 1$.

[2] In this section, we only analyze requirement dependencies between a pair of requirements. it is also possible to model requirement dependencies between two sets of requirements, i.e., between software packages. Due to space limitation, we do not discuss this issue in this paper, so refer [13, 39] for technical details.

3. It is clear that it is not possible to select both R_j and $R_{j[i]}$, because $R_{j[i]}$ is not a real requirement, but just another version of R_j which has taken the influence of the cost-related dependency between R_i into account. This means requirement R_j has `exclusion` dependency on requirement $R_{j[i]}$. i.e., $x_j + x_{j[i]} \leq 1$.

When analyzing the three constraints, the third constraint $x_j + x_{j[i]} \leq 1$ is implied by the first and second constraints. If we add the first and second constraints we have $x_{j[i]} + x_i + x_j \leq x_i + 1$. When ignoring x_i at both size, we obtain the third constraint $x_j + x_{j[i]} \leq 1$. Therefore, we can model the `cost-related` requirement dependency by creating a virtual requirement $R_{j[i]}$ and adding two new constraints:

$$x_{j[i]} \leq x_i \tag{20}$$

$$x_i + x_j \leq 1 \tag{21}$$

The artificially introduced requirement $R_{j[i]}$ leads to the following question: since we do not know whether the real requirement R_j or the artificial one $R_{j[i]}$ is actually used, which one (R_j or $R_{j[i]}$) should we use if we need to model dependencies between R_j and other requirements? We can answer this by defining a new variable x_{j*} where $x_{j*} = x_j + x_{j[i]}$. We therefore use this variable to model the dependencies between requirement R_j and other requirements. For example, if requirement R_j has `exclusion` dependency between requirement R_m, then we can set $x_{j*} + x_m \leq 1$.

4.1.6 Time-related

This type of requirement dependency means that requirement R_i needs to be implemented before requirement R_j, denoted as $R_i \prec R_j$. In fact, constraints (11) and (12) are used to model the precedence constraints. Please note that if there is no precedence constraint, i.e. no constraints (11) and (12) in the ILP model as presented in Sect. 3.3, this model turns out to be another way to represent the Knapsack model. This result also corresponds to the analysis we presented in Sect. 2: when there are no precedence constraints, the Knapsack model is sufficient and scheduling is not an issue.

It is important to mention that we consider the `time-related` dependency as *extension* of the `implication` dependency. When R_i needs to be implemented before R_j, R_i also needs to be selected if R_j is selected, i.e. R_i implies R_j. This is represented by constraint (11). The time-related dependency extends implication in the sense that it also restricts the time when a requirement can start (as shown in constraint (12)).

On the other hand, one can simply have an `implication` dependency without a time relation [19]. However, to fit the pressure on time-to-market, including the time-related dependencies can help to deal with the project plan issues already during the selection of the requirements. Here, we can consider a `time-related` dependency in fact as two dependencies, one restricting logic relationship (as what implication can do) and one restricting temporal relationship. This leads to the discussion on combined use of requirement dependencies as will be presented in the next section.

4.2 Combined use of requirement dependencies

In [19], Carlshamre assigns weights to different types of requirement dependencies, and only the highest priority one is selected as the dependency between a pair of requirements. However, Carlshamre later suggests that this mechanism for setting requirement dependencies would create some problems [6]. For example, if requirement R_1 influences the value of R_2, excluding one of them would upset customers more than excluding both of them [6]. So the user might like to add also a `combination` dependency between them. Or as discussed in former section, the user should have the flexibility to determine whether a logical `implication` dependency should or should not be associated with a precedence constraint. To handle this problem, we suggest allowing setting multiple dependencies between a pair of requirements.

Table 3 shows the possibility of combined use of requirement dependencies between a pair of requirements R_i and R_j. It applies to both the Knapsack model and the combined model. `Combination, implication` and `exclusion` cannot be set together since they are logically different. In addition, `exclusion` cannot be used in combination with others since `revenue-based, cost-based,` and `time-related` require both requirements R_i and R_j to be selected to take effect. The rest of the combinations can work well in conjunction with each other. For the example in the former paragraph, we can handle this by setting two dependencies between R_1 and R_2, i.e., `cost-related` and `combination`.

As we discussed in the previous subsection, the `time-related` dependency is the extension of `implication` dependency. Therefore, if a `time-related` dependency is used together with `combination` dependency, we can ignore the logical constraint (11). The reason is that the `combination` dependency has already implied that R_i has `implication` relationship between R_j, and vice-versa.

Although many combinations of dependencies may occur, some combinations are more natural. We already mentioned the relation between the `time-related` dependency and the `implication` dependency. In addition, if R_i reduces the development cost of R_j, it is often required to develop R_i before R_j [6]. This indicates that `cost-based` dependency that deals with the actual

Table 3 Combinatory use of requirement dependencies

Combination					
Implication					
Exclusion					
Revenue based	✓	✓			
Cost based	✓	✓		✓	
Time related	✓	✓		✓	✓
	Combination	Implication	Exclusion	Revenue based	Cost based

development process of requirements is likely to occur together with `time-related` dependencies. On the other hand, the `revenue-based` dependency deals with the functional properties of the collection of selected requirements, and is hence more related to the `combination` and `implication` dependencies.[3]

5 Prototype and simulations

5.1 Prototype

We have implemented a Java prototype for requirement selection and scheduling based on the three ILP models from Sect. 3, i.e., the Knapsack model, the scheduling model and the combined model. These prototypes run in a Linux environment and make use of the callable library of ILOG CPLEX [40] for solving the ILP problems. CPLEX is one of the best performing packages for integer linear programming [41].

Figure 2 shows a screenshot of the prototype for the combined requirement selection and scheduling model. The requirements are managed and stored in the database with estimated revenue, cost and dependencies. Given an expected release date, we can not only select the requirements for the next release, but also calculate an on-time-delivery project plan simultaneously. The selected requirements are indicated with check marks at the first column, and the dates to start are shown for each of the involved teams. Obviously, this result can easily be transformed to other types of representations, e.g., Gantt chart.

[3] By setting multiple requirement dependencies, we can also specify the internal order for developing a particular requirement. Assume requirement R_j has three jobs J_1, J_2 and J_3; and we want to perform J_1 before J_2, and J_2 before J_3. We can then consider J_1, J_2 and J_3 as three independent requirements, and set both `time-related` and `combination` dependencies between J_1 and J_2, and J_2 and J_3.

5.2 Simulation setup

In Sect. 1, we have shown that when there are different development teams and requirement dependencies in the release planning, the project plan might be delayed due to the waiting time. However, to which extend requirement dependencies can influence the project plan is still unknown. In addition, although the combined model for requirement selection and scheduling can provide an on-time-delivery schedule, additional constraints to meet the strict deadline will lead to lower revenues. The trade-off between time saving and additional revenues is also unclear. These unknown relationships lead us to investigate the following two questions.

1. What is the relationship between the number of time-related dependencies and the probability of the project running out of time?
2. What are the differences between selecting and scheduling requirements simultaneously, and selecting and scheduling requirements sequentially?

As discussed in Sect. 3, requirement select and scheduling are $\mathcal{NP}-hard$ problems with numerous factors, e.g., revenue, cost, dependencies being able to influence the final results. Such high complexity precludes us from using mathematical method (such as, algebra, logic or calculus) to obtain exact information of the models. However, we can apply *simulation* to evaluate the performance of them.

In a simulation, we numerically excise the model for the inputs and see how they affect the outputs [42]. Such technique is widely used in system design, analysis and evaluation and is one of the most widely used, if not *the* most widely used, techniques in fields like operations research and management science [42]. In the context of this paper, we use simulations to answer the two research questions mentioned above. In total, 1,600 different test cases are randomly generated in our simulations to evaluate

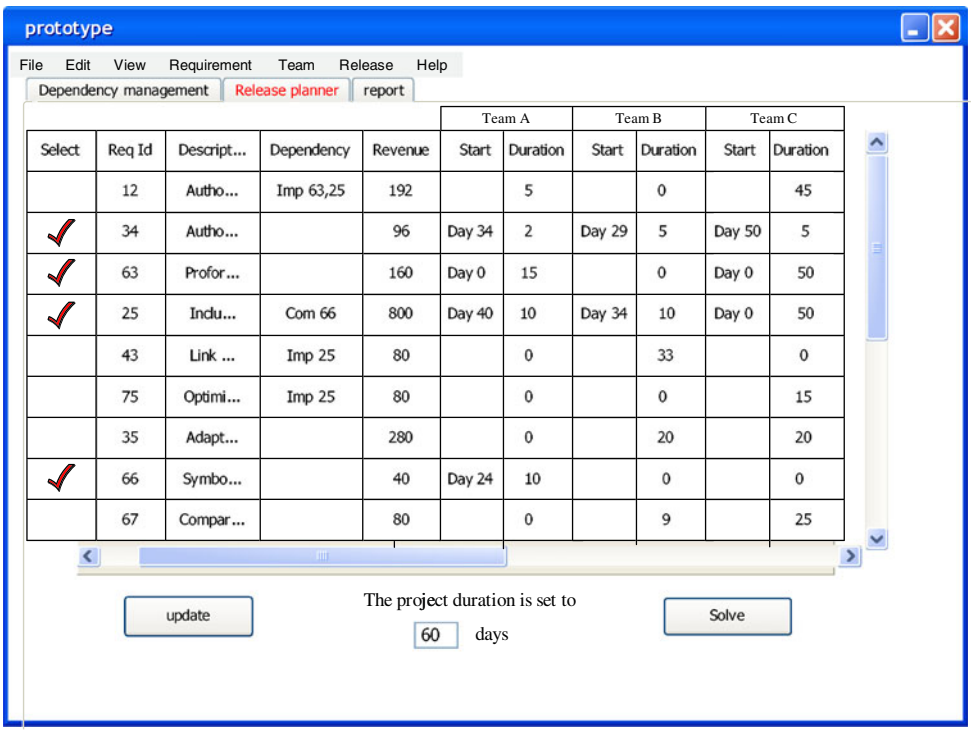

Fig. 2 Prototype of the combined selection and scheduling model

the performance of the three models and to analyze the influences of the involved factors.

To make sure that our simulations are from a *practical environment*, the following two datasets are used (available online [43] for research purposes). They are:

- *Small dataset*: 9 requirements and 3 teams, release duration 60 days.
- *Master dataset*: 99 requirements and 17 teams, release duration 30 days.

The Small dataset was the example dataset shown in Table 1. The Master dataset is collected from a large *real-life* dataset originated from a multinational ERP software vendor located in the Netherlands. All team values were kept the same, but the team capacities, and revenues were modified for confidentiality reasons.

In order to make the model not case specific, we randomly generate dependencies. In our simulation, we have only considered `implication` dependencies for the following two reasons: first, only `implication` and `cost-based` dependencies are able to influence both the requirement selection and requirement scheduling (c.f. Sect.2.1); second, a `cost-based` dependency can be mapped to an `implication` dependency and an `Exclusion` dependency (c.f. Sect. 4.1). Therefore, it is sufficient to analyze the influence of requirement dependencies on requirement selection and scheduling by only considering `implication` dependencies.

For the small dataset, we examine the situations when there are 1, 2, 3 and 4 dependencies respectively, while for the master dataset, we check the situations when there are 0.5, 1, 2 and 5% of the maximal number of possible dependencies (the maximal number of dependencies is computed by assuming every two requirements are interdependent. This equals $C_n^2 = n \cdot (n-1)/2$, where n equals the number of requirements). After determining the number of dependencies, we randomly create 100 (different) groups of dependencies and call our ILP models to determine the release for each of these 100 cases. Note that we generate these dependencies in such a way that they do not create cycles, i.e., we avoid situations like R_i depends on R_j, R_j depends on R_k and R_k depends on R_i. This is important because the requirements in the cycle would be inter-waiting each other's completion and cause a deadlock.

5.3 Results of simulation 1: the influence of dependencies on the project plan

In this simulation, we want to examine to what extend precedence constraints can influence the project span. Given the small and master dataset, we first select requirements using the Knapsack model with a given release date (60 days for the small data set and 30 for the master). In order to examine to what degree requirement dependencies (or precedence constraints) can influence the project duration, we randomly generate a certain amount of dependencies and call the scheduling model to make a project plan. We repeat such procedure for 100 times and find the maximal, minimal and average of the project makespans and count how many times the project is

Table 4 Schedule results of the first simulation

Data set	Dep ratio (%)	No of dep	The project span			Times of delay (out of 100)	Difference with lower bound		
			Max days	Min days	Average days		Max diff (%)	Min diff (%)	Average diff (%)
Small	10	1	83	55	58.80	16	0.00	0.00	0.00
(5 reqs. 60 days)	20	2	93	55	63.70	40	27.27	0.00	0.93
	30	3	103	55	70.42	62	27.27	0.00	2.64
	40	4	108	55	75.32	76	14.55	0.00	2.12
Master	0.5	14	40	30	30.93	33	30.00	0.00	2.70
(76 req 30 days)	1	29	46	30	31.38	27	8.57	0.00	0.22
	2	57	69	30	36.92	76	22.58	0.00	2.13
	5	142	84	38	56.15	100	19.23	0.00	3.47

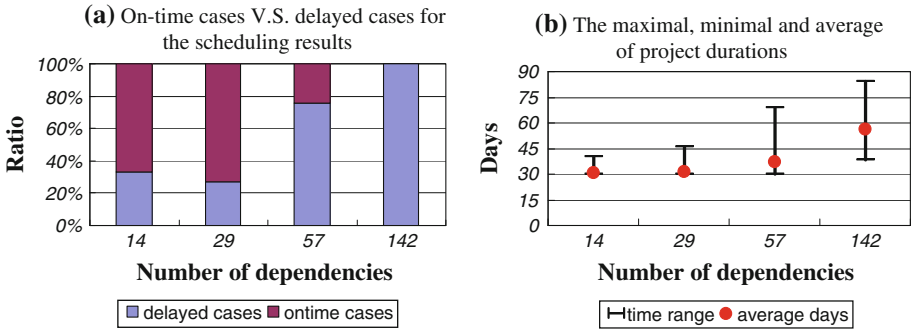

Fig. 3 Schedule results based on the master dataset

delayed, i.e., the schedule is longer than the expected release date. At last, we compare the results with the lower bound of the project makespan, which is the maximum value of the project makespan without precedence constraints and the result of longest path algorithm (c.f. Sect. 3.2). Table 4 summarizes the simulation results.

As we have 76 requirements selected for scheduling in the master dataset, it is possible to set at most 76*75/2 = 2850 dependencies. When setting the dependency ratios to 0.5, 1, 2 and 5%, we have 14, 29, 57 and 142 dependencies respectively.[4]

To visualize to what degree requirement dependencies influence the scheduling results, we plot the results of master dataset in Fig. 3. The result of small dataset follows a same trend.

Figure 3a first compare the number of cases that finish on time (on-time cases) with the number of cases which exceed the release date (delayed cases). Although in all cases the requirements selected using Knapsack model are expected to finish within 30 days, the results vary a lot. For the cases when there are 14 or 29 dependencies, the chances that the project will run out of time are not very high, i.e., at around 30%. However, the results explode when we set 57 dependencies. In this cases, in around 80% of cases the project cannot finish on time. If we further increase the number of dependencies to 142, in none of the cases the project is able to finish on time.

Figure 3b further analyzes the maximal, minimal and average of the project durations when setting different numbers of dependencies. For the cases when there are 14 or 29 dependencies, the project durations of the 100 cases range within only a few days, and their average is also close to the release date. Compared to the expected project duration of 30 days, the average project duration is 30.93 days when setting 14 dependencies, and 31.38 days when setting 29 dependencies. In these cases, it is still possible to achieve on-time delivery by means of management approaches, e.g., over-time tasks. However, the results start to explode when there are 57 dependencies between the requirements. The project durations start to vary a lot in different cases, and the average duration climbs to around 20% above the expected 30 days. If we increase the number of dependencies to 142, none of the 100 cases are able to finish on time. On average, they need 56.15 days to finish, which is almost twice as many as expected, and even in the best case it requires 38 days to complete.

[4] Here, we only count the dependencies we set but not the implied dependencies. For example, if Requirement A precedes Requirement B and Requirement B precedes Requirement C, we can also conclude from these two dependencies that Requirement A also precedes Requirement C. These implied dependencies are not counted in our simulation, since they are often not considered in practice.

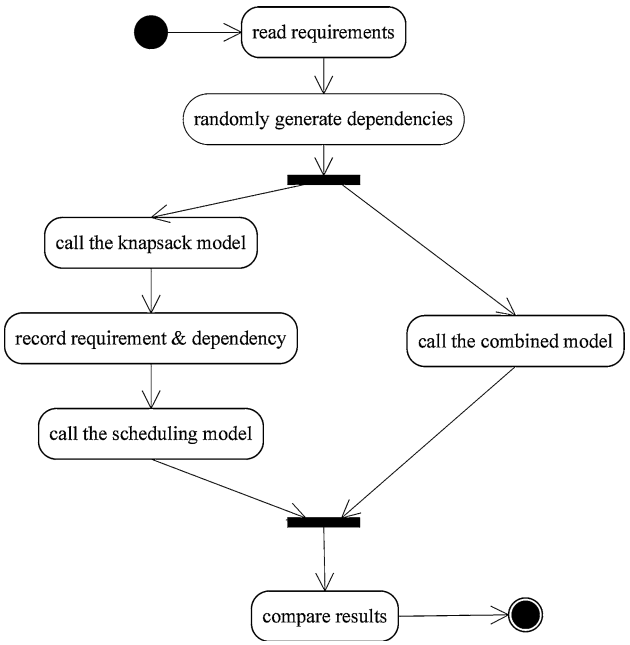

Fig. 4 Model comparison processes

It is not difficult to conclude that precedence constraints play *an important role* for release scheduling. As the number of dependencies grows, the project span grows significantly. Based on the complexity of the system, the exact number of dependencies may vary a lot, but a former survey [6] has suggested that at least 80% of requirements are interdependent, and most dependencies are `implications` and `cost-based`. This indicates that we can assume the exact number of dependencies is at least 40 for master dataset. According to our simulation, this is the number at which the simulation results start to show bad behaviors.

5.4 Results of simulation 2: model comparisons

In this simulation, we investigate the differences between two approaches. In one approach, we first apply the Knapsack model for requirement selection and then apply RCPSP model for requirement scheduling (K&S). In another approach, we call the model for combined requirement selection and scheduling to select and schedule requirements simultaneously (Comb). Figure 4 shows the procedure of this comparison.

1. Based on the small or the master datasets, we randomly generate a group of dependencies.
2. We perform two procedures in parallel. In one branch, we first use the Knapsack model to select the requirements and then document the selected requirements as well as the dependencies between them. Then, we call the scheduling model to schedule the development activities exactly in time (K&S) approach. Simultaneously in another branch, we call the combined model (Comb) approach to select and schedule the requirement simultaneously.
3. At last, we compare the results computed in the two parallel procedures in step 2. We compare the revenue difference between the Knapsack model and the combined model; the time difference between the scheduling model and release date (which is the scheduling result of the combined model) and count the number of cases the project runs out of time when following the (K&S) approach.

As we have 99 requirements in the master dataset, it is possible to set at most 99*98/2 = 4851 dependencies. When setting the dependency ratios to 0.5, 1, 2 and 5%, we obtain 24, 48, 97 and 242 dependencies respectively. For the Small dataset, we set dependency ratios to 3, 10, 15 and 20%, i.e., 1, 3, 5 and 7 dependencies respectively.[5]

Note that in case the combined model and the Knapsack model select the same collection of requirements, the RCPSP model can always find a timely schedule. Therefore, it is more interesting to only analyze the delayed cases in more details. We decided to make separate statistics only for the delayed cases. The simulation results are shown in Table 5.

5.4.1 *Influence of requirement dependencies*

When comparing the results of requirement selection (represented by revenues) and requirement scheduling (represented by the average project duration) with different number of dependencies, it becomes clear that precedence constraints play *an important role*. As the number of dependencies increase, the average revenues of the Knapsack model and the combined model keep decreasing. For example, take the master dataset. When the dependency ratio increases from 1 to 2%, the revenues of the knapsack model and the combined model both decrease by around 10%.

For the (K&S) approach, both the average project spans and the probability that the project runs out of time increase as the number of dependencies increases. The only result that remains stable is the project duration when applying the (Comb) approach, because the (Comb) approach can always find a collection of requirements that are able to finish on time. This indicates that the number of dependencies cannot influence the project duration of the combined approach. However, as the number of dependencies increases, the revenue of the combined approach decreases faster than that

[5] Note that the dependency ratios in Small dataset are different than in simulation 1. The two simulations are independent from each other, and we are also not comparing their results. Therefore, the dependency ratios do not necessarily need to be unique. The ratios are set in the way that they can better illustrate the trend in the two simulations.

Table 5 Simulation results of model comparison

Data set	Dep ratio (%)	No. of dep	Statistics for the 100 runs			Statistics **only** for the delayed cases					
			Average of revenue (comb)	Average revenue (k&s)	Average project span (k&s)	No. of delay (k&s)	Average revenue (comb)	Average revenue (k&s)	Average project span (k&s)	Average revenue diff (%)	Average time diff (%)
Small	3	1	1113.36	1130.16	56.62	9	989.36	1176	73	15.87	21.67
(9 reqs	10	3	1024.48	1060.24	58.15	17	884.24	1094.56	76	19.15	26.67
60 days)	15	5	918.48	971.6	59.25	22	794.16	1035.6	76.59	22.92	27.65
	20	7	844.72	886.96	57.72	24	832.16	1009.12	76.07	16.84	26.78
Master	0.5	24	40420.1	40429.5	30.48	17	40442.1	40493.5	32.82	0.13	9.41
(99 reqs	1	48	39275.5	39479.1	32.62	45	38965.7	39400.9	35.82	1.15	19.41
30 days)	2	97	35581.6	36103.1	36.41	68	35351.8	36118.7	39.43	2.11	31.42
	5	242	26947.7	29127.3	45.61	95	26S04.5	29098.8	46.43	7.84	54.77

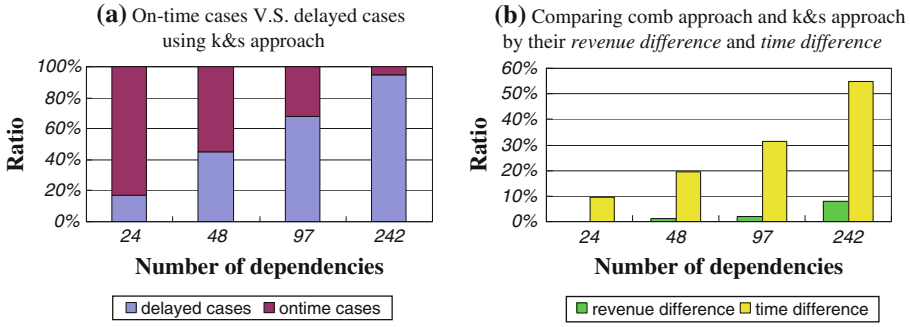

Fig. 5 Model comparison result based on master dataset

of the knapsack approach in order to make sure the selected requirements can be implemented on time. This triggers us to systematically compare the two suggested approaches.

5.4.2 *Comparing the (K&S) approach and the (Comb) approach*

To better compare the two approaches, we plot the computational results of master dataset in Fig. 5. The results based on the small dataset follow the same trends as the master dataset.

Figure 5a first compares the number of on-time cases with the number of delayed cases using (K&S) approach. Note that we only depicted the results of (K&S) approach in Fig. 5a because the (Comb) approach guarantees on-time delivery schedule, i.e., the number of delayed cases is always 0. While the (Comb) approach always provides a schedule that can finish on time, separating selection and scheduling stands a high change of being delayed, and this probability grows as the number of dependencies increases. For example, when there are 242 dependencies within the 99 requirements, the chance for missing the release date is 95%.

As discussed in Sect. 5.2, the (comb) approach can save time by providing an on-time-delivery schedule, but this additional constraint can also reduce the revenue of the selected requirements. Fig. 5b then further compares the revenue and project duration when applying the (K&S) approach and when applying the (Comb) approach. In Fig. 5b, the baselines of the comparison are the results of the (Comb) approach. We then compare the *revenue difference* between the (Comb) approach and the (K&S) approach, and the *time difference* between the (Comb) and the (K&S) approach. The differences depicted in Fig. 5b are measured by the ratios that the (Comb) approach differs from the (K&S) approach in percentage.

It becomes clear from Fig. 5b that the combined selection and scheduling approach is more efficient. For example, if we set 242 dependencies between the 99 requirements (5% of theoretical maximal number), using (K&S) would require up to 54.77% additional development time after the expected release date compared with the (Comb) approach that can always finish on time. However, it only gains about 7.84% more revenue. Such trend, i.e., the revenue gain is significantly lower than the time spent when using the (K&S) approach, can also be observed when there are different number of dependencies. Clearly, adopting the combined requirement selection and scheduling model is preferred since it is a lot more efficient with respect to revenue per unit throughput time.

Based on this simulation, we can conclude that the (Comb) approach not only can provide an on-time-delivery schedule (c.f. Fig. 5a) but is also *more efficient* with respect

to revenue per unit throughput time (c.f. Fig. 5b). In our simulation, neither the number of requirements nor the number of dependencies can influence this comparison results, i.e., such trend can be found in all scenarios.

5.5 Sensitivity of the parameters

There is no need to doubt that a good estimation of the parameters, like revenue v_j, dependencies and the definition of an appropriate release date T are the key to success in our approach. It is also apparent that a precise estimation is extremely difficult and luxury [6, 44]. Therefore, it is important to know which factors have the most impact on the solution in case of disturbances. Then, we can focus on these critical factors when given limited time or resources. Based on the simulation in this section and the work in [12, 13], we view the importance of these factors as follows:

5.5.1 Revenue v_j

The revenue v_j shows the absolute market value or relative importance of a requirement R_j. This factor might be the most difficult one to predict, since lots of external stakeholders can influence it, and it is largely based on market condition. Surprisingly, this factor has a *limited* influence on the selection result. We created 1,000 instances based on the master dataset, and for every instance, we randomly assigned ± 10, ± 20 and $\pm 30\%$ perturbation to the revenues of 10% requirements independently. When we look at the solutions found for these perturbated instances, in only five cases the set of selected requirements differ from the originally selected requirement set. When looking at the difference, this turns out to be also small: the majority of the selected requirements in the original solution are also selected in the solution for the perturbated instances [13].

5.5.2 Dependency

Requirement dependencies show the relationship between requirements and are also very difficult to estimate in practice. In this paper, we have evaluated its influences on the revenue and cost for a software release. In Sect. 5.3, we have first investigated the influence of requirement dependencies on the project plan (c.f. Fig. 3). For the master dataset, when only 5% of all pairs of requirements are interdependent, the project duration is doubled, and none of the 100 cases in the simulation lead to a project plan with on-time delivery (c.f. Table 4). In Sect. 5.4, we have examined the dependency's influence on revenues (c.f. Fig. 5). The expected revenue dropped by almost 25% if we increase the number of dependencies from 97 to 242 for the master dataset (c.f. Table 5). To sum up, requirement dependency has a *significant* influence on both the project duration and the project revenue. Special attention should be paid to the dependencies between the requirements in the critical path of the project plan. These dependencies often lead to waiting time in different teams, therefore need additional attention to judge whether these dependencies are really necessary.

5.5.3 Release date T

It is important to decide on a good value for the release date T since it directly influences the available capacity for the release. Such date is also hard to determine since it is largely determined by market conditions [1]. In [13], we examine the situation where we delay the release for a while, and the penalty costs for the deadline extension are linear in its length. The result is depicted in Fig. 6.

The curves show step characteristic due to the fact that we only obtain an additional revenue when a new collection of requirements are selected. For example, the expected revenue keeps decreasing as the number of days extended increases. The reason is that we put penalty cost for every additional day. However, if we extend the release date for 10 days, we are able to select a new collection of requirements which result in additional revenue. Therefore, we see a jump at day 10. The two curves in Fig. 6 indicate clearly that the influence of postponing the release date is largely *dependent on the cost of delaying*. When the cost is low, delaying the release for a few days can yield more profits than the cost it conducts. However, when the cost of delaying is high, it is not worth extending the release day.

6 Adaptive software release planning

Until now, our approach supports the release planning for a fixed time period based on fixed estimations of parameters. In practice, parameters like revenue, cost of requirements may evolve over time, because the release is being developed in a changing market. It may also happen that one very important customer places an order after the release is determined, and some of the new features must be added in the coming release. In this section, we focus on adaptively handling these changes of the release planning.

It may seem a straightforward solution to change the deterministic parameters into stochastic ones, i.e., change parameters from fixed values to variables following certain probability distribution in order to cover the dynamic behavior. Although some preliminary work has been conducted in stochastic scheduling, e.g., [45, 46], it is not recommended based on two reasons. First, despite the studies analyzing the type of distributions for the parameters in RCPSP models [29, 45, 46], it remains difficult to

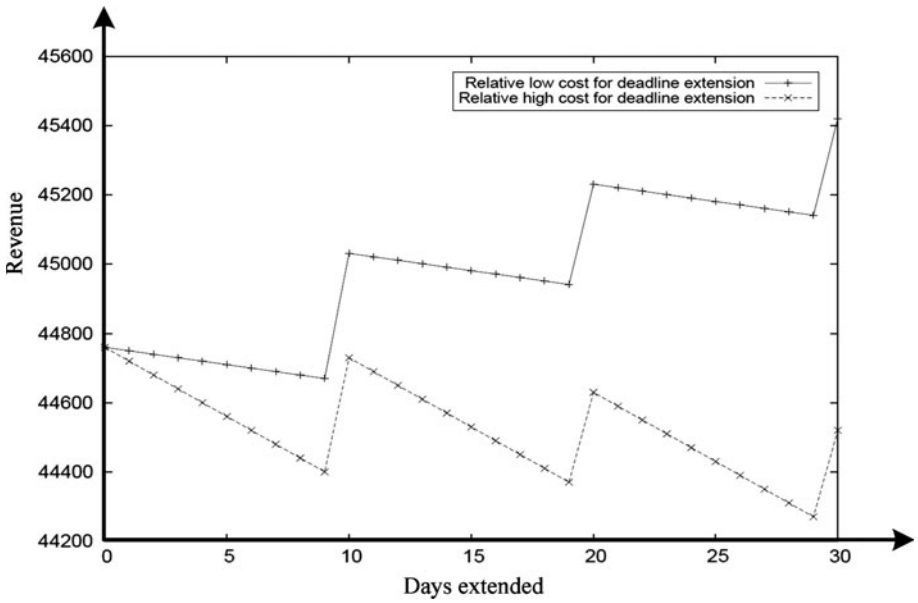

Fig. 6 Total revenues against the number of days extended

determine which distribution the stochastic data should follow in software release planning. Second, the complexity of solving stochastic RCPSP problems turns out to be extremely high, so that the performance of the suggested approach will not be very satisfactory, and it is not possible to scale up [45].

In this section, we propose to handle the changes in two ways. In Sect. 6.1, we provide a method to adaptively change the parameters after the release has started. In Sect. 6.2, we apply the concept of "scrum" in agile software development method [23, 24]. By using the "scrum" concept, we can divide the long development project into several small internal projects so that we can adjust the parameters after every scrum.

6.1 Dynamic adjustment of the parameters

Based on the requirements given in Table 1, a general example of the replanning problem is depicted in Fig. 7. The project started at time T_0 with requirements R34, R35, R63, R66 and R67 determined to be implemented in this release. Team A, Team B and Team C were assigned with a couple of jobs to perform in the release period. Assume that after the project has already started, a replanning is needed at T_1 due to over/under-estimations of parameters or an important new order. At this replanning time point T_1, we can divide the requirements into three groups.

1. The first group contains all completed requirements, for example, R34 in Fig. 7.

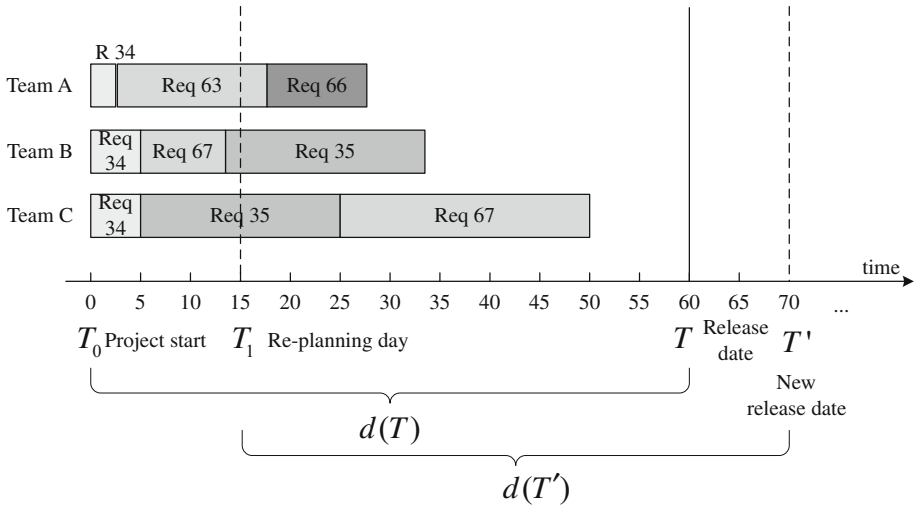

Fig. 7 An example of the release adjustment problem

2. The second group contains the requirements currently under development, for example, R63, R35 and R67 in Fig. 7. Please note that if any one of the teams has not completed its work for a certain requirement, such requirement still belongs to this group, e.g., R67.
3. The third group contains the rest of requirements that have been scheduled for development but have not yet started. For example, R67 in Fig. 7 belongs to this group. This group also contains all the requirements in the repository, i.e., the ones that have not yet been selected.

At the replanning point, the product manager needs to decide which jobs to continue and which to drop if necessary. Moreover, requirements from the repository might be selected. We can use our model to make such a decision by *re-setting the parameters* and *re-computing the new release* from T_1 until the new release date T'. In the following, we describe how to adjust the parameters to perform a replanning of the next software release.

6.1.1 Change release date

Suppose T' is the new release date. As the project has already started, we need to re-compute the number of available working days in the project. We therefore need to use $d(T')$ to measure the number of available working days between the replanning time T_1 to T'. In our case, $d(T')$ contains 55 days.

6.1.2 Change expected revenue

Suppose at time T_1, updates of the estimated revenue of some requirements become available. We therefore can change the parameters v_j to say v'_j. For the replanning, it is needed to update the revenue values of both the requirements under development and requirements scheduled to be developed later. Obviously, for planning purposes, there is no need to change revenue values of the completed requirements, since better or worse, the requirements have already been developed.

6.1.3 Change the number of man days or change requirement dependencies

If we need to change the expected number of man days for developing a requirement or want to add additional dependencies, we need to separately consider the three groups of requirements, as described in former paragraphs.

1. *First group* The requirements in this group have already been implemented. So it is not necessary to adjust the number of man days for these requirements. However, these requirements R_j might appear in existing or additional dependencies. In this case, these requirements still need to be considered in the replanning, and we need to add additional constraints $x_j = 1$ because these requirements have already been implemented. In addition, we can also ignore all the precedence constraints set between this group of requirements and other requirements, since they have already completed.
2. *Second group* For the second group of requirements, the amount of man days a_{ij} should be modified for the jobs that have not been completed at T_1. For the replanning, we consider the *remaining* days a'_{ij} to complete requirement R_j. For example, the man days needed for requirement R63 in team A should be changed to 2 days. Please note that we do not deduct the revenue v_j of requirement R_j from replanning although we have already developed something for this requirement. Reason is that we assume the expected revenue can only be obtained after the whole development is complete.

   As different teams work on a requirement independently, the development of a requirement at different group may have different situations. At the replanning time T_1, the jobs J_k for a requirement in the second group must have one of the following three conditions:

   (a) The jobs are already finished, e.g., the job for requirement 67 in team B. If some groups have already finished their jobs for a requirement, we can simply ignore these jobs in the replanning and only consider the rest of the jobs. The reason is that the completion and dependency of this requirement only depends on the remaining jobs.
   (b) The jobs are under development, e.g., the jobs for requirement R63 in team A, and requirement R35 in team B and team C. If the requirements in this group are re-selected, we would rather continue the development after this decision point. So rather than defining a group of timing variables ξ_{kt} for a job under development (c.f. Sect. 3), we only define one variable ξ_{k0}, i.e. whether this job should start at decision point. In this way, we guarantee that if the model decide to continue with this requirement, it will start immediately after this replanning point T_1.
   (c) The jobs have been scheduled but have not yet started, e.g, the job for Requirement 67 in team C. If a job is scheduled to start in future by the original project plan, we can ignore the old schedule. Since the job has not yet started, we have the flexibility to let the new model compute its starting time again based on the situation at T_1.

Therefore, we need to define ξ_{kt} in the way as described in Sect. 3.

3. *Third group* The third group contains all the requirements scheduled to be implemented according to the old plan and all the requirements in the repository, e.g., requirement R66 in Fig. 7 and all the remaining requirements which were not selected. We can change the expected number of man-days and dependencies according to the re-estimations. There is no restriction to reset them since none of these requirements have started, and none of them have been guaranteed to be re-selected again beforehand. In addition, if an important new order enforces R_j to be put into this release. We can set an additional constraint $x_j = 1$ to enforce the selection of this requirement.

6.1.4 Replanning

After modifying the parameters and re-setting the variables in the above mentioned way, we can put all the requirements in the second and third group back into the requirement repository. We also have to include, with constraint $x_j = 1$, the requirements R_j from group 1, which are involved in the dependencies with requirements from group 2 and group 3. Then, we can *run the ILP model* again. The result will show which requirements to drop or which ones to add. If we use the combined requirement selection and scheduling model (c.f. Sect. 3), a new project plan according to the new estimations will be also made simultaneously.

When we follow the suggested approach, it is possible to terminate some ongoing requirements if we are able to find better candidates. There are mainly three reasons to model it in this way:

1. From financial point of view [47], the earlier effort put on a requirement before the decision date is sunk cost. It should not influence the decision.
2. There is no guarantee that the requirements currently under development are better than the requirements in the waiting list. It is possible that after changing the estimated revenues, the requirements in the waiting list have higher revenues than the ones under development.
3. At last, the requirements under development have higher probability to be selected again, because we have already deduced a part of the development cost.

However, this preemption of ongoing development should be a management decision rather than a strict constraint. If managers decide to in any case continue with the current development for morale or other reasons, the model provides such flexibility. To model this situation, i.e., all requirements in group 2 should continue until the developments are complete finished, we need to add additional constraints $x_j = 1$ for all requirements R_j in this group and add constraints $\xi_{k0} = 1$ for all jobs J_k currently under development.

Besides how to perform replanning, another question is when to perform replanning. Two possibilities are as follows. First, we can re-plan after a fixed period of time, e.g., every month. Second, we can re-plan if the disturbances exceed a certain threshold, e.g., if 10% of the jobs are delayed for more than 20%. Computer simulation might help to find appropriate values as well [42]. However, replanning has a large impact on the human resource management of the development process, and therefore the viewpoint of the management is also essential.

6.2 Scrum development

The scrum development method divides the whole development project into several scrums. At the end of each scrum, an internal release is expected [23, 24]. After every scrum, the project manager can adjust the project plan according to the current situation. This incremental method does not only provide the manager with the flexibility to adjust the release plan but also reduces the risk of overtime, since only a limited amount of requirements are developed in each scrum. We have depicted the concept in Fig. 8. Let us revisit the example in Fig. 1, where the project (lasting 60 days) was developed using one shot development approach. Assume that the project has been divided into two scrums of 30 days (c.f. Fig. 8). At the end of each scrum, we expect to finish some requirements. Compared to the one shot development method, managers have now one additional opportunity to adjust the release plan, i.e., at the end of Scrum 1.

However, as all the requirements in a scrum are expected to finish at the end of this scrum, this strict deadline acts like "concrete walls" which will reduce the flexibility of scheduling. For example, consider the case in Table 1. If we divide the development schedule into two scrums with 30 days in each scrum, we are only able to select Requirement 34, 35, 63 and 66 in the first scrum and Requirement 67 in the second scrum. The total revenue of the selected requirements in the two scrums is 656 Euro (c.f. Fig. 8). This result is significantly lower than the result of one shot approach (1,096 Euro using the combined model), because some long running but profitable requirements, e.g., Requirement 25, cannot fit in any scrum.

We can improve the situation by allowing splitting of jobs. For example, since Requirement 25 is a long running but very profitable one, we can divide this requirement into 2 sub-requirements, Requirement 25a, and Requirement 25b. Clearly, this split can lead to additional cost, like addition setup cost or coordination cost. So we may expect that the development duration of this requirement

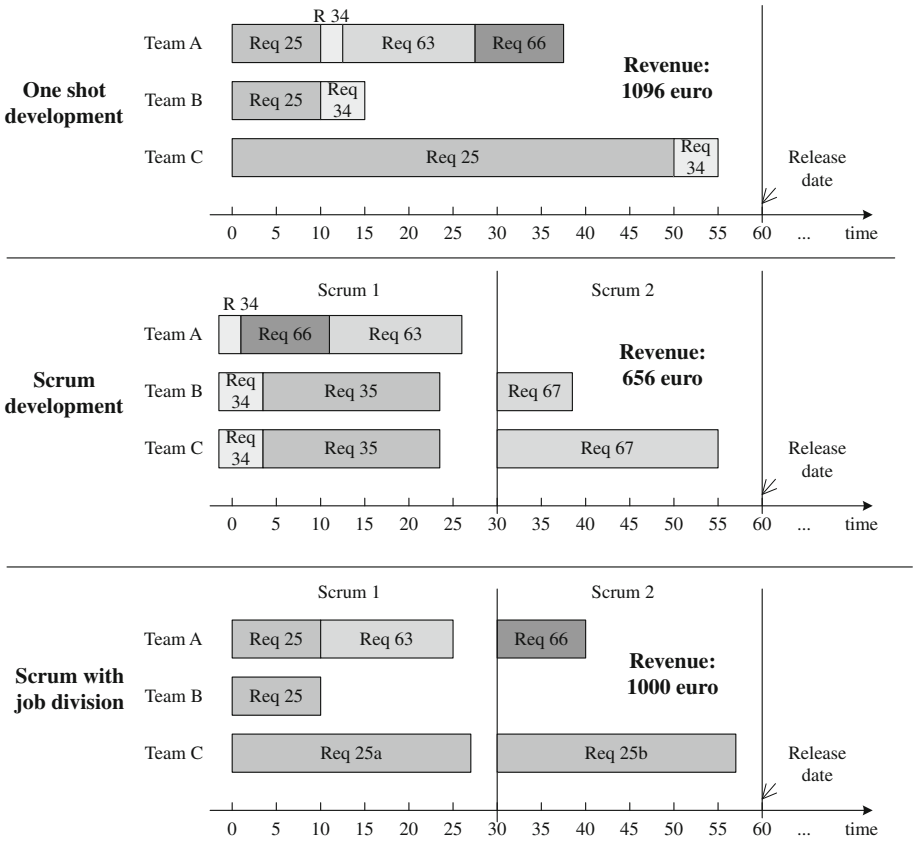

Fig. 8 Scrum development method

(Requirement 25a + Requirement 25b) will become longer (as shown in Fig. 8). Assume that we can split Requirement 25 evenly into Requirement 25a and Requirement 25b with 10% additional man days, i.e., each needs 28 man days and produces 400 Euro as revenue. As shown in Fig. 8, if we allow job division, it is then possible to select and schedule another group of requirements, so that the revenue and efficiency are increased. The revenue of the new collection of requirements increases from 656 to 1,000 Euro. Clearly, managers can split a requirement and estimate the cost and revenue according to the special feature of this requirement and the situations in the groups. Note that in a similar way, we can also split a requirement such that the jobs of different team can be assigned to different scrums.

When compared to the one-shot development process, adopting the Scrum development concept provides additional opportunities to adjust the project plans so that it is more flexible. Hence, we can better handle unforeseen changes. However, such flexibility will possibly reduce the revenue of the development project due to the strict deadlines for completing requirements at the end of each scrum. To alleviate this problem, we provide an approach that allows splitting jobs in order to improve the revenue of the development project. We can identify the candidate jobs for splitting by comparing the jobs selected in the one shot development approach but not in the scrum development process (e.g. Requirement 25 in Fig. 8).

From a scheduling point of view, we conclude the following. To achieve the benefits of scrum, it is crucial that the work can be split into relatively small pieces without a lot of overhead or transition cost (see [48] for an approach to refine big requirements into smaller ones). Such strategy of designing only small requirements (or dynamically divide big requirements into several smaller ones) can also be observed in several software companies, which typically work with requirements of maximal 2 to 5 days [48].

7 Conclusion and future research

As deciding on the requirements for upcoming software release is a complex and critical process, this paper aims at providing mathematical models to support such a decision. The main contributions of this paper include:

1. First, we have applied the RCPSP model to solve the requirement scheduling problem. This scheduling model can synchronize the development in different teams and is able to handle the precedence dependencies between requirements and the resource or expertise constraints in the company.

2. Second, we have designed a new ILP model that combines the requirement selection and scheduling. This combined approach not only can maximize the revenue of the selected requirements, but also provides an on-time-delivery project plan simultaneously. The requirement dependencies, which represent the relationship between requirements, can also be included in this model. To out best knowledge, this paper has provided the first model to combine requirement selection and scheduling together.
3. Third, we have investigated the influences of different factors and compared the different models by several simulations. The results indicate that the combined requirement selection and scheduling model can not only provide a on-time-delivery schedule, but is also more efficient than the traditional Knapsack model in terms of revenue per time unit spent on the project. In addition, we have indicated that requirement dependencies are more important than other factors, like revenue, release date, with regard to both requirement selection and scheduling.
4. At last, we provided two mechanisms allowing dynamically adjust the release after the project has started. The first mechanism allows run-time modification of the model, and the second one applies the concept of scrum method in agile software development.

The results look very promising, but more work still needs to be done. In the field of requirement engineering, our simulations show convincing results to combine the requirement selection and scheduling. However, more empirical research is needed to fully evaluate this process improvement opportunity. Applying our models to an actual project can also yield better understanding of their benefits. It is also interesting to investigate the uncertainty of the parameters [49], so that we can consider these uncertainty at the planning phase. In the fields of scheduling, our first simulation results suggest that the optimal schedule found by integer linear programming is not far away from the critical path lower bound. It will also be interesting to investigate whether there are faster algorithms, e.g., heuristics search, for scheduling that can get rather close to the optimum. At last, the scalability of the models is so far unknown (in this paper, we have tested the condition up to 100 requirements and 17 teams), more research is needed to test the scalability of the suggested ILP model and make it applicable for larger dataset.

References

1. Regnell B, Brinkkemper S (2005) Market-driven requirements engineering for software products. In: Aurum A, Wohlin C (eds) Engineering and managing software requirements. Springer, Dordrecht, pp 287–308
2. Novorita R, Grube G (1996) Benefits of structured requirements methods for market-based enterprises. In: International council on systems engineering sixth annual international symposium on systems engineering: practice and tools (INCOSE'96)
3. van de Weerd I, Brinkkemper S, Nieuwenhuis R, Versendaal J, Bijlsma L (2006) Towards a reference framework for software product management. In: RE '06. IEEE Computer Society, pp 312–315
4. Carlshamre P, Bjorn R (2000) Requirements lifecycle management and release planning in market-driven requirements engineering processes. In: DEXA Workshop, pp 961–965
5. Joachim K, Kevin R (1997) A cost-value approach for prioritizing requirements. IEEE Softw 14(5):67–74
6. Carlshamre P (2002) Release planning in market-driven software product development: provoking an understanding. Requir Eng 7(3):139–151
7. Firesmith D (2004) Prioritizing requirements. J Object Technol 3(8):35–47
8. Greer D, Ruhe G (2004) Software release planning: an evolutionary and iterative approach. Inf Softw Technol 46:243–253
9. Sawyer P, Sommerville I, Kotonya G (1999) Improving market-driven re processes. In: International conference on product focused software process improve-ment (PROFES'99)I
10. Ho-Won Jung (1998) Optimizing value and cost in requirements analysis. IEEE Softw 15(4):74–78
11. Ruhe G, Saliu M. Omolade (2005) The art and science of software release planning. IEEE Softw 22(6):47–53
12. van den Akker JM, Brinkkemper S, Diepen G, Versendaal J (2005) Flexible release planning using integer linear programming. In: REFSQ'05. Essener Informatik Beitrage, pp 247–262
13. van den Akker JM, Brinkkemper S, Diepen G, Versendaal J (2008) Software product release planning through optimization andwhat-if analysis. Inf Softw Technol 50(1–2):101–111
14. Wolsey LA (1998) Integer programming. Wiley-Interscience Series In Discrete Mathematics and Optimization
15. Leffingwell D, Widrig D (2000) Managing software requirements— a unified approach. Addison-Wesly
16. Ngo-The A, Ruhe G (2009) Optimized resource allocation for software release planning. IEEE Trans Softw Eng 35(1):109–123
17. Berander P, Andrews A (2005) Requirements prioritization. Springer, Dordrecht
18. Cusumano MA (2004) The business of software. Free Press, New York
19. Carlshamre P, Sandahl K, Lindvall M, Regnell B, Nattoch Dag J (2001) An industrial survey of requirements interdependencies in software product release planning. In: RE'10, IEEE Computer Society, pp 84–91
20. van der Aalst WMP, van Hee KM (2004) Workflow management: models, methods, and systems. MIT Press, Cambridge
21. Anda B, Dreiem H, Dag I, Sjoberg K, Jorgensen M (2001) Estimating software development effort based on use cases-experiences from industry. In: UML'01. Springer, UK, pp 487–502
22. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: ICSE'00—Future of SE Track, pp 35–46. ACM
23. Rising L, Norman JS (2000) The scrum software development process for small teams. IEEE Softw 17(4):26–32
24. Schwaber K, Beedle M (2001) Agile software development with scrum. Prentice Hall, USA

25. Li C, van den Akker JM, Brinkkemper S, Diepen G (2007) Integrated requirement selection and scheduling for the release planning of a software product. In: REFSQ'07, pp 93–108. Springer, LNCS 4542
26. Cormen TH, Leiserson CE, Riverst RL, Stein C (2001) Introduction to algorithms. MIT Press, Cambridge
27. Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley-Interscience Series In Discrete Mathematics and Optimization
28. Aristide M, Vittorio M, Salvatore R, Lucio B (1998) An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. Manag Sci 44(5):714–729
29. Artigues C, Demassey S, Neron E (2008) Resource-constrained project scheduling, models, algorithms, extensions and applications. Wiley-Iste
30. Blazewicz J, Lenstra JK, Rinnooy Kan AHG (1983) Scheduling projects subject to resource constraints: classification and complexity. Discrete Appl Math 5:11–24
31. Jorge LV, Balakrishnan R (1995) Strength and adaptability of problem-space based neighborhoods for resource-constrained scheduling. OR Spectr 17(1–3):173–182
32. Demeulemeester E, Herroelen W (1992) A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. Manag Sci 38(12):1803–1818
33. Wall MB (1996) A genetic algorithm for resource-constrained scheduling. PhD thesis, Massachusetts Institute of Technology
34. Frankola T, Golub M, Jakobovic D (2008) Evolutionary algorithms for the resource constrained scheduling problem. In: 0th international conference on information technology interfaces, pp 715–722
35. van den Akker JM, Diepen G, Hoogeveen JA (2007) A column generation based destructive lower bound for resource constrained project scheduling problems. In: CPAIOR'07, volume 4510 of LNCS. Springer, pp 376–390
36. Brucker P, Knust S (2006) Complex scheduling. Springer, Dordrecht
37. Dyer ME, Wolsey LA (1990) Formulating the single machine sequencing problem with release dates as a mixed integer program. Discrete Appl Math 26(2–3):255–270
38. van den Akker JM, van Hoesel CPM, Savelsbergh MWP (1997) A polyhedral approach to single-machine scheduling problems. Math Program 85(3):541–572
39. Li C, van den Akker JM, Brinkkemper S (2006) An integer linear programming approach to product software release planning & scheduling. Technical Report UU-CS-2006-065, Utrecht University, The Netherlands
40. ILOG Cplex. http://www.01.ibm.com/software/integration/optimization/cplex
41. ILP Product Comparison. http://www.lionhrtpub.com/orms/surveys/LP/LP-survey.html
42. Law AM (2006) Simulation modeling and analysis. McGraw-Hill Higher Education
43. Release Dataset. http://www.home.cs.utwente.nl/~lic/Resources.html
44. Regnell B, Höst M, Natt och Dag J, Beremark P, Hjelm T (2001) An industrial case study on distributed prioritisation in market-driven requirements engineering for packaged software. Requir Eng 6(1):51–62
45. Golenko-Ginzburg D, Gonik A, Laslo Z (2003) Resource constrained scheduling simulation model for alternative stochastic network projects. Math Comput Simul 63(2):105–117
46. Lambrechts O, Demeulemeester E, Herroelen W (2008) Proactive and reactive strategies for resource-constrained project scheduling with uncertain resource availabilities. J Sched 11(2):121–136
47. Brealey RA, Myers SC, Marcus AJ (2004) Fundamentals of corporate finance. McGraw-Hill/Irwin
48. Vlaanderen K, Brinkkemper S, Jansen S, Jaspers E (2009) The agile requirements refinery: applying scrum principles to software product management. In: IWSPM'09, USA
49. Klir GJ, Wierman MJ (1999) Uncertainty-based information. Physica-Verlag, Wurzburg