Software release planning under soft resource and dependency constraints

Mark Przepiora
Laboratory for Software Engineering Decision Support
University of Calgary
Calgary, Canada
mpmrzepi@ucalgary.ca

Abstract—The goal of incremental software release planning is to assign planning items to future releases so that value is maximized while various resource, technical and other constraints are satisfied. We propose a new approach to this problem called soft-EVOLVE II, in which a project is first modeled using a highly-general model with a nonlinear objective function, and then a set of high-value release plans is generated using a hybrid genetic algorithm. Our formalization naturally extends the established EVOLVE II model by Ruhe et al., but is capable of expressing a much larger variety of constraints between features, in which traditional, Boolean feasibility constraints are replaced entirely by soft constraints. Such a formulation allows for highly-detailed modeling while taking uncertainty into account. We discuss the primary hypotheses and goals of this research, as well as potential validity threats and their control.

Keywords—release planning; genetic algorithms; search-based software engineering

I. INTRODUCTION

In incremental software development, a software product is given sequential releases with a growing set of features or functionality with each release. The difficulty lies not only in finding a satisfactory assignment of features to releases (a release plan) that meets the constraints of the project as well as stakeholder demands, but also in being able to model the project and its constraints to a contextually-appropriate level of detail [1].

Although many approaches to this problem have been described, one common limitation in the models proposed is that they are not expressive enough to describe the complete range of constraints that may be present in a project. Another common limitation is that constraints (be they resource constraints, technical constraints or others) are modeled by Boolean values, in which a single violated constraint (no matter how loose this constraint may be in the real-world) makes an entire release plan infeasible. Such formulations give rise to an unrealistically-large infeasible region in the set of possible release plans, inside of which otherwise-high-value plans may reside.

We attempt to overcome these two common modeling limitations by presenting a new approach called soft-EVOLVE II. In Section II, we begin by defining a highly-general optimization problem in which the goal is to maximize some quantifiable notion of value. We argue that virtually any type of constraint may be modeled in our formalization, and violated constraints incur a “soft” value penalty rather than making a plan infeasible, (although the penalty may be $-\infty$ in the limiting case).

In Section III we analyze our formalization and review related literature to argue that our model is more expressive than existing approaches, and that such soft constraints allow for more realistic models than Boolean ones.

In addition to modeling a problem, the goal of release planning is of course to search for a suitable plan. Traditionally, this process is done solely by human beings in an ad-hoc, or on-the-fly manner. However, in the past decade there has been an effort to replace or augment human-based search with computer-based search. This effort has bred its own area of study called Search Based Software Engineering (SBSE) [2], in which software engineering problems are interpreted as search problems. This analysis has been applied to a wide field of problems—not only to planning problems such as requirement optimization [3] (what interests us) and resource allocation [4], but also others such as software design [5] and testing [6]. Since release planning problem lend themselves naturally to SBSE analyses, in Section IV we propose a genetic algorithm designed to solve problem instances by searching for high-value release plans.

In Section V we describe the soft-EVOLVE II method itself and how it is meant to be applied using the model and algorithm described in the preceding sections.

Finally in Sections VI and VII, we describe our current research plan, the experiments we are performing, and the threats to their validity.

II. PROBLEM DEFINITION

A. Basic Definitions

The objective of release planning is to assign each of a number of planning items (e.g. software features) to one or more future releases or else postpone it, while meeting various technical and resource constraints. To that end, we present the following general formalization.

Definition 1. We define an instance of the Release Planning problem (also called a project) to be a
collection of planning items \{1,2,\ldots,N\}, release periods \{1,2,\ldots,K\} (where \(K+1\) is a special release period named POSTPONED), and a set of constraints \(C\) (defined below). Individual planning items and release periods are denoted \(n,k\) respectively.

**Definition 2.** Given a project, a release plan is a map \(x:\{1,2,\ldots,N\} \rightarrow \{1,2,\ldots,K+1\}\), and the collection of all release plans is denoted \(X\). Semantically, \(x(n) = k\) means that feature \(n\) is released in release period \(k\).

**Definition 3.** A constraint in a project is a function \(c:X \rightarrow \mathbb{R} \cup \{-\infty\}\) where \(c(x)\) gives the value gained by implementing release plan \(x\). A positive value denotes value gained, whereas a negative one denotes value lost. A value of \(-\infty\) indicates that a plan is completely infeasible with respect to the constraint.

**Definition 4.** The value of a release plan is given simply by the sum

\[
F(x) = \sum_{c \in C} c(x).
\]

The goal, then, is to find a release plan with greatest value, or one with sufficiently large value.

The formalization we’ve described is general, as nearly any release planning scenario may be reduced to it. This fact can be seen in the simplest case when a project has a single constraint function \(F\), its objective function. (Note that we have imposed no restrictions of linearity, convexity, nor any other on \(F\).)

However, in practice the objective function will be a sum of easily-defined, common types of constraints, some of which we describe below.

**B. Basic Constraints**

Perhaps the most basic type of constraint we’d like to express is one imposed by a finite, quantifiable resource (e.g. money, effort) needed to complete each planning item in a project, a fixed amount of which is available in each release period. This class of constraints includes some resources for which it may not be possible to carry over unused units to future releases (an example of which may be effort in person-hours, if employees work full-time on a fixed schedule), as well as others (such as a budget in dollars) for which it may be possible to use unused units in the next release. We call the former a noncumulative resource and the latter a cumulative resource, and both fit within the following framework.

**Definition 5.** A resource constraint is defined using four auxiliary functions: (1) a consumption function \(con(n)\) denoting the number of units needed to complete planning item \(n\), (2) a capacity function \(cap(k)\) denoting the number of units available in each non-postponed release period \(k\), (3) an over-consumption function \(\Delta(x,k)\) denoting the number of units over-consumed in release \(k\) by implementing release plan \(x\), and finally (4) a non-positive penalty function \(p(z,k)\) denoting the cost in value of over-consuming \(z\) units in release \(k\). Then, the constraint itself is given by \(c(x) = \sum_k p(\Delta(x,k),k)\).

We illustrate the previous definition with an example. Consider a project with a single release period \(k = 1\), and suppose that implementing a release plan \(x\) requires 100 person-hours, but only 25 person-hours are available, with additional labour (e.g. hiring temporary workers or paying overtime) costing \$75 per hour. We may model this situation as a resource constraint with \(cap(1) = 25\) (since 25 units of resources are available), \(\Delta(x,1) = 75\) (since the plan requires 75 additional units), and \(p(z,1) = 75z\) (since each additional hour of labour costs \$75).

We may now generalize the preceding example. For the next two definitions, let \(\{\cdot\}^+ = \max(0,\cdot)\) denote the positive part of an expression.

**Definition 6.** A noncumulative resource constraint is a resource constraint whose over-consumption function is given by

\[
\Delta(x,k) = \left\{ \sum_{x(n)=k} con(n) - cap(k) \right\}^+.
\]

i.e. \(\Delta(x,k)\) is the total resource consumption in release \(k\) less the capacity, in case this number is positive.

**Definition 7.** A cumulative resource constraint is a resource constraint whose over-consumption function is given by

\[
\Delta(x,k) = \left\{ \sum_{x(n)=k} con(n) - \forall(x,k) \right\}^+ \]

where \(\forall(x,k)\) denotes the number of units available at the beginning of release \(k\). In case all unused units may be carried over to future releases, is can be showed that

\[
\forall(x,m,1) = cap(1)
\]

\[
\forall(x,k+1) = cap(k+1) + \left\{ \forall(x,k) - \sum_{x(n)=k} con(n) \right\}^+.
\]

Note the similarities between the two over-consumption function above. In both cases the over-consumption function is the difference between resources used and resources available in a release, the difference being that in the case of cumulative resources, the “resources available” term may be a function of the release plan itself.

Of course, satisfying resource constraints is not the only goal of release planning. In addition, it is usually possible to assign a value to releasing each individual planning item. This scenario naturally gives rise to a linear objective function, which can be modeled as a constraint in our formalization:
Definition 8. Let \( val(n, k) \) be the non-negative value of releasing planning item \( n \) in release \( k \), independent of any other release decisions. Then the constraint given by \( c(x) = \sum_n val(n, x(n)) \) describes a linear value function.

The combination of resource constraints and a linear value function alone allows us to formalize many simple release planning scenarios. For example, if it is known that each planning item in the project will bring in a certain amount of revenue, \( val(n, k) \) could represent this dollar value. However, in case more complicated technical constraints exist between planning items, we show how they may be modeled below.

C. Advanced Constraints

We define three kinds of technical constraints that may be present in a project:

Definition 9. A pre-assignment is a constraint of the form

\[
c(x) = \begin{cases} 0 & \text{if } x(n) \leq k \\ p & \text{else} \end{cases}
\]

where \( n \) is a planning item, \( ? \in \{\leq, =, \geq\} \), \( k \) is a release period, and \( p \) is a negative penalty value. Semantically, this specifies that \( x \) should satisfy either \( x(n) \leq k \), \( x(n) = k \) or \( x(n) \geq k \) respectively, and that the penalty for violating this constraint is \( p \).

Definition 10. A weak precedence constraint is a constraint of the form

\[
c(x) = \begin{cases} 0 & \text{if } x(n') \leq x(n'') \\ p & \text{else} \end{cases}
\]

where \( n' \) and \( n'' \) are planning items and \( p \) is a negative penalty value.

Definition 11. A strong precedence constraint is a constraint of the form

\[
c(x) = \begin{cases} 0 & \text{if } x(n') < x(n'') \\ p & \text{else} \end{cases}
\]

where \( n' \) and \( n'' \) are planning items and \( p \) is a negative penalty value.

The above two constraints allow us to model situations where the value or implementation of one item depends on another item being complete. Finally, we introduce a fourth kind of constraint,

Definition 12. Let \( S \) be a set of planning items. A logical constraint is a constraint \( c(x) \) that can be written solely as a function of the number of planning items from \( S \) that are not postponed by plan \( x \). That is, there is a function \( \phi(j) \) such that

\[
c(x) = \phi(\{ n \in S \mid x(n) \leq K \}).
\]

The motivating example of a logical constraint is given by the constraint that at least one item from a set \( \{n', n'', \ldots\} \) should be released, which describes a logical OR relation.

D. Discussion

Note that in this formulation, an instance does not strictly have an ‘infeasible region’ in general. The objective function \( F \) is defined for every plan \( x \), but depending on the problem instance and release plan \( x \), \( F(x) \) could be very small or even \(-\infty\). Thus, in general, constraints are ‘soft’ and may be broken if the payoff is worth the penalty.

In this formulation, there is no difference in principle between a value function and any other constraint function: both contribute (either positively or negatively) to plan’s value. This is in contrast to usual formulations in which the (usually, but not necessarily) linear objective function is a separate entity from constraints, which are Boolean in value and must be respected, i.e. in which \( F(x) \) may only either be positive or \(-\infty\). (In the following section we argue why such constraints do not properly model constraints in software development projects.)

III. ANALYSIS OF MODEL AND RELATED WORK

In [7] Svaenb erg, et al. reviewed 28 release planning models and evaluated, among other things, what requirement selection factors each is capable of handling (i.e., what kinds of constraints and other factors can be expressed in the model). In particular, the authors considered ‘hard’ and ‘soft’ factors. The former class consists of requirement dependencies, non-functional requirements, cost constraints, resource constraints, effort constraints, and time constraints. The latter consists of stakeholder influence factors, value factors, risk factors and resource consumption factors.

Since our model is described most generally by an arbitrary value function, it should be possible to model any of these factors. However, for the sake of comparison it is useful to compare the factors outlined above to the constraints templates we defined in Section II. In particular, all of the hard factors except for non-functional requirements can be naturally expressed in our model using only the constraint templates previously defined. It furthermore supports value factors and resource consumption factors explicitly, for a total of seven factors. No other model considered by Svaenb erg et al. supported this many factors, supporting our claim that the model presented here is highly expressive even in the default case.

Earlier we claimed that soft constraints (i.e. those with finite penalties) allow for more realistic models than strict, Boolean ones. Now we will discuss the evidence for this claim, both indirect and direct. Vasant and Barsoum note that objective and constraint functions tend to be non-linear, and that the data on which they are based are vague [8]. Furthermore, a survey by Carlshamre, et al. found that a large majority (80%) of planning items are interdependent on others [9]. The prevalence of uncertain data and a high degree of interdependence imply that, even if all constraints are well-defined and known to be Boolean, the infeasible regions they carve out, in reality, must include some
infeasible solutions and exclude some feasible ones, as the constraints are defined using uncertain data.

Indeed, there has been much effort to account for uncertainty in software development and other decision making scenarios. Seemingly the most popular method of doing so is through the application of fuzzy logic, in which constraint values, rather than being simply satisfied or violated, are replaced by the 0-to-1 degree to which they are satisfied or violated. In the field of software development, fuzzy logic has been applied to software evaluation [10], cost and effort estimation [11, 12], and requirement analysis [13], among other areas. This framework is widely used in manufacturing as well [14]. Within the domain of industrial planning, Vasant and Barsoum [8] use it to smooth out the endpoints of uncertain constraints.

However, this method is less applicable to the problem of release planning as described here. Fuzzy logic is applied best to real-valued constraints such as resource constraints, but these make up a small portion of the constraints found in release planning decisions. The rest (such as precedence constraints, coupling, pre-assignments, logical constraints and others) are essentially Boolean, but as we argued above, there is still a great measure of uncertainty surrounding them in most projects. Instead of forcing a plan to be infeasible if any single such constraint is broken, our model allows us to express the penalty for violating the constraint.

IV. ALGORITHM

Once a project is fully defined, a genetic algorithm can be applied to compute a set of high-value release plans. A hybrid approach is used here, in which a greedy heuristic is used to generate an initial population of plans, which are then evolved using the genetic algorithm. We briefly describe the two components, although this is still a work in progress.

A. Heuristic

It is well-known that the performance of a genetic algorithm can be improved greatly by using a heuristic to generate initial solutions and find local optima of existing ones [15]. To this end, the initial population used by the genetic algorithm is generated using a greedy algorithm that repeatedly attempts to select a planning item assignment $x(n) = k$ with a high change in value and low risk. Of course the notion of “risk” is crucial in order for this to work. Roughly, “risk” should be a measure of how negatively the current assignment could affect future assignments.

B. Genetic Algorithm

A genetic algorithm is used to evolve the initial population. Here there are a number of important parameters that must be selected, including the population size, maximum number of generations, stopping conditions, mutation and crossover rates as well as methods, and selection operator. These are all parameters we aim to optimize in our research.

C. Diversification

Once a population of high-quality solutions has been computed, a custom metric is used to select a small set of not only high-quality but also highly-diverse solutions. This metric takes into account not only objective value but also the vector distance between solutions. This step is key to the soft-EVOLVE II process (described in the following section).

V. RELEASE PLANNING METHOD SOFT-EVOLVE II

A. Background and Overview

EVOLVE II [1] is a formal release planning method in which stakeholder input and expert decisions are combined with computational methods via the paradigm of hybrid intelligence [16] in order to find high-quality release plans. The process is iterative and consists of three, repeating phases: Modeling, Exploration and Consolidation. First, an initial model is created to describe a project, using the expert opinions of project managers as well as stakeholders (Modeling). Next, computational methods are applied to reduce the problem space and find a set of high-quality release plans (Exploration). Then human expertise is once again applied to evaluate the solutions found and, if necessary, return to the Modeling phase for another iteration (Consolidation).

The goal of EVOLVE II is generally to determine the preferences of stakeholders and maximize their satisfaction while meeting the various resource and technical constraints of the projects in a strict way.

Our goal, on the other hand, is to maximize the value of a release plan in a single, quantifiable unit, whether or not high value in this sense necessarily corresponds to stakeholder satisfaction. Furthermore, we are not necessarily concerned with satisfying all constraints in a strict way, for the reasons argued in Section III.

Figure 1. An illustration of the EVOLVE II process [1].
These considerations suggest limitations with our approach, but they also carry clear advantages. For example, implementations of the EVOLVE II approach often suffer from the drawback that the objective value of a release plan has little or no meaning in absolute terms, being an aggregate value of scores across various, possibly unquantifiable criteria. Maximizing a singular notion of value, on the other hand, has a clear meaning to the stakeholders and managers of a project, and we hope to show that this aids in the Consolidation step of the process.

Our method, soft-EVOLVE II, follows these same three general steps, but Modeling and Exploration are carried out by implementing the model and algorithm from Sections II and IV respectively. We briefly describe these differences in the following subsections.

B. Modeling

In both processes, EVOLVE II and soft-EVOLVE II, the goal of the Modeling step is to take a complicated, real-world project—one with dozens or possibly hundreds of objectives, constraints, explicit concerns and implicit concerns—and model it mathematically to the level of detail appropriate for the scope of the planning. In particular, modeling amounts to determining (1) the planning items of the project, (2) the constraints of the project, and (3) the value of releasing each planning item.

The soft-EVOLVE II Modeling process mirrors that of EVOLVE II, but extra steps are needed for the former: namely, when defining constraints, penalties for violating each constraint must be defined as well, (which may require input from the project’s stakeholders in addition to the managers). The appropriate values for these penalties may be instance-specific, or they may fit into common templates. We give an illustrative example in the case of a precedence constraint

\[ n' \leq n'' \]

where such a constraint may belong to one or more general categories:

- **Item \( n'' \) has no value unless \( n' \) is released first**, in which case the penalty for violating this constraint may be

\[ -\text{val}(n', x(n'')) \]

- **In order to implement \( n'' \), \( n' \) is required as a prerequisite**. In this case, a violation may be fixed by forcing \( n' \) to be released in any release before \( n'' \), taking on resource over-consumption penalties if applicable.

- **Releasing \( n'' \) before \( n' \) has some other negative effect on value**. In this case, the penalty value may be defined entirely by the project manager.

C. Exploration

The Exploration phase of soft-EVOLVE II is performed by applying the algorithm from Section IV in order to generate a small set of high-value, highly-diverse release plans. Diversity of solutions is of key importance to both the EVOLVE II and soft-EVOLVE II processes, because of their iterative nature: it is assumed that every project, in addition to its explicit concerns such as limited resources, technical constraints, and so on, also suffers from a number of implicit concerns that are either difficult to model or identify. Exploration is carried out, in part, in order to identify the implicit concerns present in a project, and to evaluate their presence in the proposed release plans. [1] By presenting project managers with a variety of release plans, they are better able to choose a plan that addresses these concerns, or else return to the Modeling phase in order to address them explicitly.

VI. RESEARCH OBJECTIVES

A. Overview and Research Questions

The soft-EVOLVE II process outlined above is a comprehensive, hybrid approach to release planning, and so it naturally raises a number of questions, both computational and empirical in nature,

- Is the model valid? Is this an accurate way to model real-world projects?
- Is the model scalable? Is it too difficult (in human effort) to explicitly define the necessary details of a project?
- How well are solutions perceived by practitioners? Does “high value” as modeled by our process correlate with their idea of “high value”? Is the result of soft-EVOLVE II planning perceived better than the result of EVOLVE II or ad-hoc planning?
- Are there certain parameters to the genetic algorithm that work particularly well for most problem instances?
- Is the algorithm scalable? How well does the algorithm perform with respect to the number of planning items, constraints, and so on?

B. Current Focus and Research Plan

The first, second and third questions have been answered in part by previous research. The EVOLVE II process has been used extensively in industry settings in order to model projects and optimize plans of various sizes. Since our modeling process simply extends that of EVOLVE II, these three questions become somewhat simplified to asking whether the generalizations offered (but not imposed) by our process are relevant or necessary.

However, our current focus is on the fourth and fifth questions. In particular, we aim to optimize the performance of the genetic algorithm proposed in Section III for real-world projects of various sizes and complexity. Since such a task requires dozens or hundreds of input data, it is implausible for us to use solely real-world data to achieve this goal. Instead, we aim to algorithmically generate a test suite of problem instances with varying parameters. Although such data is artificial, we aim to generate projects whose “constitution” matches that of real-world projects, i.e. a realistic number of interdependencies between planning items, common kinds and distributions of technical constraints, realistic statistical distributions of resource constraints and capacities, and so on. If a sufficient number of these properties match those of real-world projects, the results gathered through a study of these artificial projects should be valid about real-world data with a high degree of certainty. Since our research is still young, we are still
investigating the best strategy for generating this kind of data.

The primary metric we use to measure improvements in the performance of our algorithm is the increase in objective value of the highest-quality solution obtained by the algorithm (within a fixed amount of time.)

VII. THREATS TO VALIDITY

A. Construct Validity

The main threat to the construct validity of our approach is the question of whether an increase in objective value truly reflects an increase in plan quality in any meaningful way. The way we control this threat is by generating project data that closely mimics that of real-world projects. This process is in essence a construct validity control, since if an artificial project accurately mimics a real-world one, then its objective value must also mimic a real-world notion of value whose increase is clearly valued.

B. Internal Validity

Due to the inherent stochastic nature of genetic algorithms, we must control for the possibility that the results we obtain are not the result of luck. However, this is easy to do simply by considering a sufficiently large number of projects, and by solving each project a sufficient number of times, such that any uncertainty in the results becomes negligible.

C. External Validity

To support external validity, we must generate data that is sufficiently varied in the ranges of parameters used (in number of planning items, number of resources, types of constraints, and so on) as well as in the kinds of projects represented, based on the analysis of real-world data used to generate the artificial data. The more varied the data, the more generalizable our results will be.

ACKNOWLEDGMENTS

This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Discovery grant no. 250343-07, and an Alexander Graham Bell Canada Graduate Scholarship (CGS-M). Thanks to my supervisor Günther Ruhe for all his support in preparing this paper and his support throughout the course of my research.

REFERENCES


