Finding Optimal Plans for Incremental Method Engineering

Kevin Vlaanderen, Fabiano Dalpiaz, and Sjaak Brinkkemper

Utrecht University,
Department of Information and Computing Sciences,
{k.vlaanderen,f.dalpiaz,s.brinkkemper}@uu.nl

Abstract. Incremental method engineering proposes to evolve the information systems development methods of a software company through a step-wise improvement process. In practice, this approach proved to be effective for reducing the risks of failure while introducing method changes. However, little attention has been paid to the important problem of identifying an adequate plan for implementing the changes in the company's context. To overcome this deficiency, we propose an approach that assists analysts by suggesting—via automated reasoning—optimal and quasi-optimal plans for implementing method changes. After formalizing the Process-Deliverable Diagrams language for describing the method changes to implement, we present a planning framework for generating plans that comply with different types of constraints. We also describe an implementation of the modeling and planning components of our approach.

Keywords: method engineering, evolution, planning, logic programming.

1 Introduction

Incremental method engineering [18,24] is a paradigm that proposes to change information systems development methods through a continuous improvement process. This strategy adheres to the common understanding that spreading changes over a time period is less risky and more efficient than introducing them all at once.

Empirical studies have provided significant evidence in favor of incremental method engineering [24,16,7], including cases on the introduction of Scrum [21], showing that it helps to cope with the major obstacles to change, including resistance to change, fear of ineffectiveness by the involved personnel, and resource constraints [2].

However, existing research has largely ignored the relevant problem of identifying a plan that specifies *when* to implement the changes. This requires defining which are the most (and least) urgent changes, how many changes can be implemented given time and budget constraints, and when a plan is better than another. See the following example.

Example 1. A software organization wants to improve customer satisfaction by introducing the Kano analysis [13]. However, introducing and learning the Kano analysis requires a significant effort, due to its complexity (Fig. 4). The management team is concerned with upfront costs and risks of resistance to change. A product manager suggests introducing it incrementally, but she cannot devise a proper plan. How can Kano analysis be embedded within the current process? Are there any variations possible? □

M. Jarke et al. (Eds.): CAiSE 2014, LNCS 8484, pp. 640–655, 2014.

In this paper, we address the problem of devising a plan for the incremental implementation of a set of method changes in an organization (as elaborated in Sec. 2). We use automated reasoning techniques for generating *optimal* and *quasi-optimal* plans. We propose a formalization of the Process-Deliverable Diagram (PDD) modeling language [23], that we use to describe the changes to be implemented.

We combine these elements into a method that, based on a description of the changes to be enacted, assists the analysts by suggesting possible plans, and helps to refine these plans to improve fitness with the organizational context. These plans have to satisfy mandatory constraints, and should *satisfice* [19] weak (nice-to-have) constraints.

While we are inspired by automated planning techniques, we develop a novel solution that copes with the specificities of method engineering. This includes defining sequencing based on deliverables rather than activities, and using weak constraints to derive a plan leading to an incremental maturity growth in the organization.

After stating our problem in Sec. 2, and discussing our research baseline in Sec. 3, we propose the following contributions:

- We formalize the PDD modeling language, adding clear semantics, so as to make it usable for automated reasoning. This formalization is described in Sec. 4.
- We define a formal framework that defines optimal and quasi-optimal plans with respect to an input PDD, an organizational context, and a set of time and budget constraints. The framework in Sec. 5.1 forms the basis for plan generation.
- We propose a process that guides the analysts in applying the framework to generate plans and to refine them by strengthening and relaxing constraints (Sec. 5.2).
- We develop tool support for our method: PDD models can be created via a graphical modeling environment, and automatically converted into input for a logic program in disjunctive Datalog [15] that generates (quasi-)optimal plans (see Sec. 6).

Sec. 7 illustrates our approach using the scenario in Example 1. Sec. 8 discusses related work, presents conclusions, and outlines future directions.

2 Problem Statement

Our research context is method evolution, i.e, the process through which an organization's methods change over time. We focus on incremental method engineering, a well-defined process for managing and incrementally introducing method changes.

This approach is illustrated in Fig. 1. Whenever the process is triggered (either by an occurred event, or every N months/years), the maturity of the current processes is assessed. If any process shows low maturity, the stakeholders' needs are considered to identify *what* to change, and, subsequently, by defining a plan that specifies *how* and *when* to deploy these changes in the organization. Finally, the changes are enacted as per the plan. This process is highly iterative, for organizations are in constant evolution.

In this paper, we focus on the important yet under-explored activity of *change planning*. The problem is that of determining a (quasi-)optimal scheduling for implementing the changes, based on all the constraints (hard and soft ones) of the stakeholders. To address this non-trivial problem, several sub-questions have to be answered:

- Which factors determine the optimality of a plan?
- Which elements describe the hard constraints that cannot be violated?

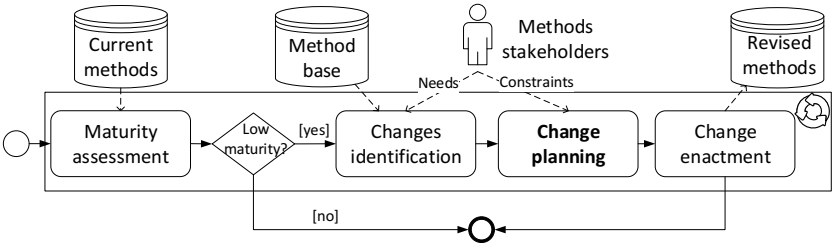

Fig. 1. Research context: systematic method evolution illustrated

- Which factors determine the priority of changes?
- How can one ensure that, even when partially deployed, the introduced changes can be effectively used in the organization?

In this paper, we consider the gradual introduction of new methods. In reality, it is more common to deal with changes to existing methods, which also requires to consider the removal and replacement of fragments. We leave dealing with the more complex case of decrements (as opposed to increments) for future research.

3 Baseline

We approach the challenges in Sec. 2 as part of the Online Method Engine (OME) [22], a knowledge management system for incremental method engineering. The OME consists of a method base that contains method fragments (generic descriptions of common approaches to software development tasks), rules for combining fragments, and organizational experience related to the fragments. These elements feed the four main functions of the OME: (a) disseminating method knowledge; (b) assessing the maturity of an organization's processes; (c) suggesting improvements based on method fragments and experience from similar organizations; and (d) enacting improvement proposals.

Process-deliverable Diagrams (PDDs) are a fundamental component of the method base: method fragments are modeled using a combination of UML activity diagrams—to describe the procedural aspects—and UML class diagrams—to express the data

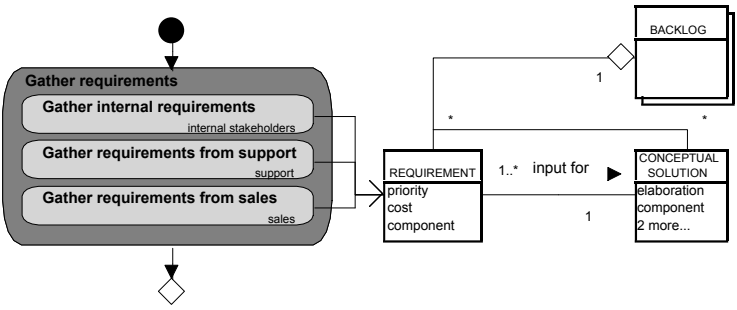

Fig. 2. Example of a Process-Deliverable Diagram (excerpt)

aspects (using classes called "deliverables"). These models are connected through a "results in" relationships from the activities to the deliverables. PDDs also distinguish between simple, closed, and open activities and deliverables [23]. Fig. 2 illustrates the core components of a PDD through an example. More details can be found in [23].

We also adopt the notion of focus area maturity matrix from the Situational Assessment Method (SAM) [4] (see Tab. 1). This matrix is filled in based on questions concerning situational factors as well as organizational capabilities. Each of these capabilities (with level A-F) contributes to the maturity of the organization in a specific focus/process area. For example, capability A in the focus area (row) "Requirements gathering" corresponds to a basic registration of the requirements, which contributes to maturity level 1, while capability F in the same area corresponds to the involvement of partners in the product management process, which contributes to maturity level 8.

By using a focus area maturity matrix instead of a fixed level maturity matrix, we are able to suggest more detailed improvement suggestions [4]. SAM enables assessing the maturity level of an organization and identifying the desired (target) maturity level.

Table 1. Example Capability Maturity Matrix (excerpt)

Focus Area		**Maturity Levels**										
Title	**Code**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Requirements Management												
Requirements Gathering	RG		A		B	C		D	E	F		
Requirements Identification	RI			A			B		C			D
Requirements Organizing	RO				A		B		C			

4 PDDs for Planning Method Changes

We formalize the relevant parts of the PDD language [23] so as to make it usable for automated reasoning. We state the requirements for our refinement in Sec. 4.1, and we present a revised metamodel and its semantics in Sec. 4.2.

4.1 Requirements

We want to leverage previous work: PDDs are a simple yet expressive means to model a method fragment's activities, deliverables, and their relationships. This choice is made to reuse the method base of fragments (modeled as PDDs) within the OME system.

PDDs were designed as a means to intuitively communicate methods and fragments to users. The price of this choice is that some constructs have ambiguous semantics that cannot be readily used for automated reasoning. Thus, we need to provide a clear semantics of the PDD metamodel, by fulfilling the three requirements ($\mathbf{R_1}$ to $\mathbf{R_3}$) below.

$\mathbf{R_1}$: Avoid Generic Associations. Consider a directed association between two deliverables: "customer wish" and "theme". The original PDD metamodel does not specify a detailed semantics for associations, which inhibits determining the nature and the strength of the link. Thus, we require our language to avoid generic associations.

R_2: Include Input and Output Relationships. PDDs do not express input relationships, i.e., that an activity needs to use a deliverable. This choice eases readability, but does not express when a deliverable is needed. This obstacles planning for change: an activity that requires a deliverable cannot be introduced until that deliverable is produced. Our language needs to unambiguously express input and output relationships.

R_3: Distinguish Deliverable Dependency Types. Consider the following dependency between deliverables: the priority of a requirement is based on the input from customers and partners, but one of them suffices. This can be represented as an association between a requirement and an aggregated *input* deliverable in PDDs, but the semantics are not sufficiently detailed to indicate the choice. We thus require our language to support more specialized deliverable relationships.

4.2 PDD Metamodel

In Fig. 3, we present a refinement of the part of the PDD metamodel [23] that relates with describing method change. The semantics of the language should enable describing *what* changes have to be introduced, and expressing the dependencies between deliverables that pose constraints on *when* the different activities are introduced.

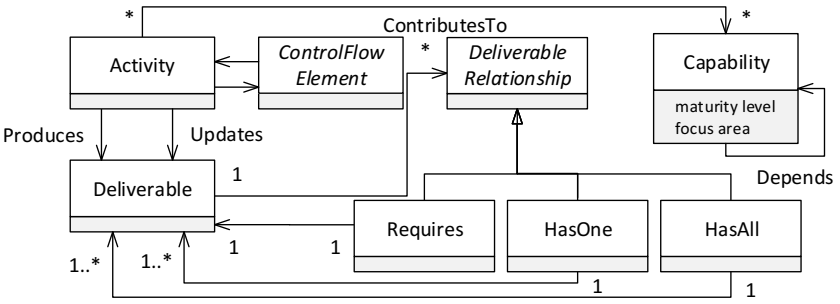

Fig. 3. Partial metamodel of the PDD language for describing method changes

Since we are interested in the implementation order of a set of activities, rather than in their execution order, our metamodel does not restrict the process side of the PDD: *ControlFlowElement* in Fig. 3 is a generic placeholder for all the control flow constructs of PDDs (sequence, decision point, fork and merge, etc.).

ContributesTo: this relationship indicates that an *Activity* contributes to a certain *Capability*, thus helping the organization to reach that capability's maturity level. An activity can contribute any number of capabilities, and a capability can be contributed by any number of activities. Graphically, contributions are textual annotations "FocusAreaCode:Capability" on the left of activities. In Fig. 1, e.g., activity "Analyze Product Environment" contributes to capability D of focus area "Requirements Gathering".

We implement several changes to fulfill $\mathbf{R_1}$–$\mathbf{R_3}$. We remove the distinction between simple, complex, and closed deliverables, as these concepts are not useful for our purposes. Also, we replace generic associations ($\mathbf{R_1}$) with the following relationships.

Requires: a transitive and asymmetric binary relationship between deliverables D_1 and D_2, indicating that the D_1 can be produced only when D_2 already exists. In Fig. 4, deliverable "Functional Questions" requires "Requirements".

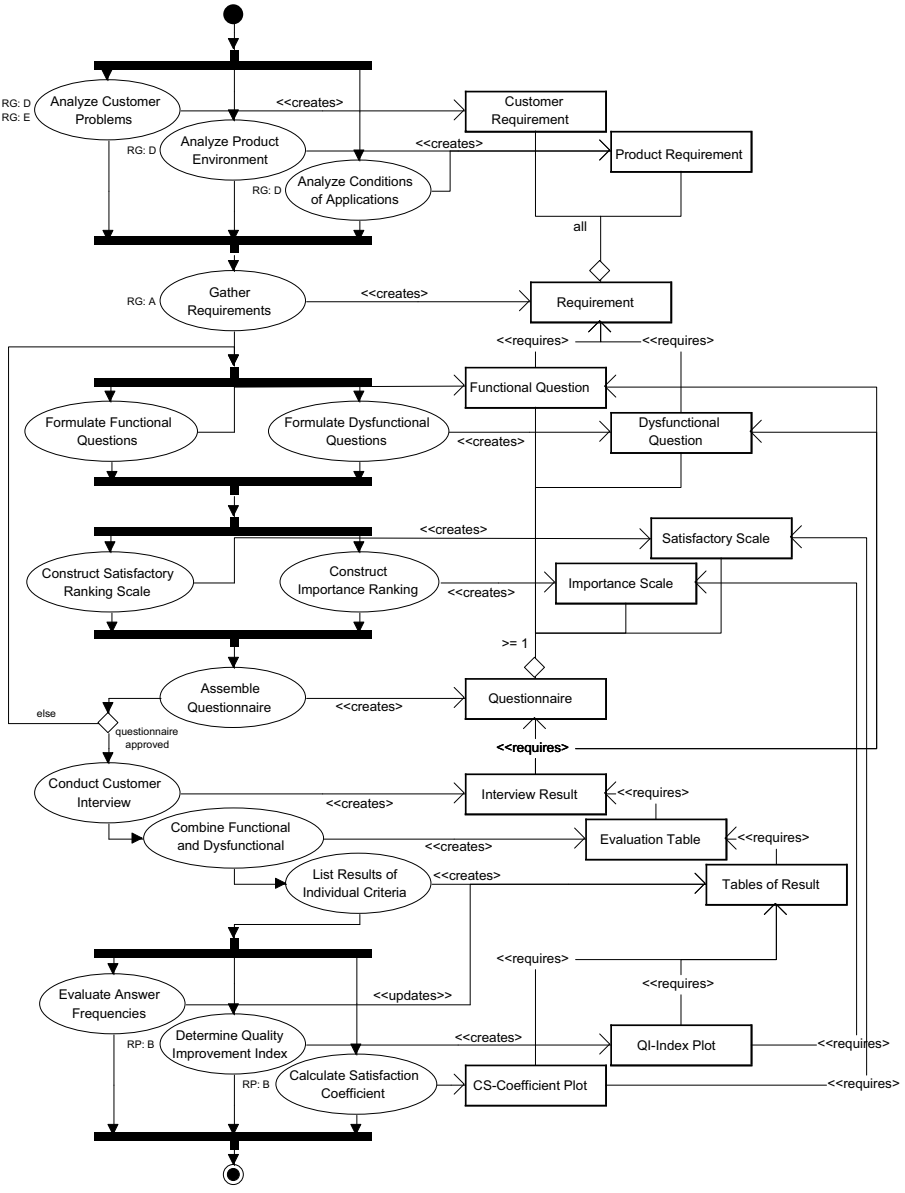

Fig. 4. PDD for the KANO Analysis

Produces/*Updates*: in a PDD, activities can be linked to deliverables only through the "results in" relationship between activity A and deliverable D. Here, we specialize this relationship into *Produces*, to denote that A creates a previously non-existent deliverable, and *Updates*, to indicate the modification of a previously available deliverable. In Fig. 4, activity "List Results of Individual Criteria" produces deliverable"Table of Results", while activity "Evaluate Answer Frequencies" updates it.

Together, the *Requires*, *Produces*, and *Updates* relationships satisfy $\mathbf{R_2}$: *Requires* (D_1,D_2) denotes that any activity that produces or updates D_1 needs D_2 as an input; *Produces*(A,D) indicates that A has output D; and *Updates*(A,D) specifies that D is both an input and output for A.

HasOne/*HasAll*: to allow for fine-grained deliverable dependencies ($\mathbf{R_3}$), we specialize aggregation into the *HasOne* and *HasAll* relationships. The former indicates that at least one of the parts of a deliverable shall be available in order to produce the deliverable itself, while the latter requires that all of its parts are available. In Fig. 4, "Requirements" requires both (*HasAll*) "Customer Requirements" and "Product Requirements"; on the contrary, "Questionnaires" requires at least one among "Functional Questions", "Dysfunctional Questions", "Importance Scales", and "Satisfactory Scales".

5 Planning for Method Evolution

We present the formal framework that derives plans for implementing a set of changes described in a PDD (Sec. 5.1). We introduce a process for determining the most adequate plan (Sec. 5.2), and illustrate our approach on the Kano analysis scenario (Sec. 7).

5.1 Formal Framework

We introduce the necessary elements to define the plan generation function $genplans$, which returns all plans for implementing a set of changes, given a set of constraints. We represent time instants via natural numbers: typically, 0 would denote the current time (CT), 1 would denote CT + X milliseconds, 2 would denote CT + 2·X milliseconds, etc.

Definition 1. ***Implementation cost*** *is defined by a function* $cst : \mathcal{ACT} \times \mathcal{ORG} \rightarrow \mathbb{R}^+$ *s.t.* $cst(\mathsf{A}, \mathsf{Org})$ *is the cost of implementing activity* A *in the organization* Org. □

The notion of cost here goes beyond monetary expenses, and it includes additional costs for the organization, including the learning effort, risk of failure, impact on motivation, time, etc. We measure cost in terms of abstract cost units; the definition of a mapping that reduces real cost to units is beyond the purpose of this paper.

Definition 2. *A* ***scheduling schema*** *is a list of time slots* $(\langle \mathsf{St}_1, \mathsf{End}_1, \mathsf{Bdg}_1 \rangle, \ldots, \langle \mathsf{St}_n, \mathsf{End}_n, \mathsf{Bdg}_n \rangle)$ *wherein changes can be implemented. The i-th time slot* $\langle \mathsf{St}_i : \mathbb{N}_0, \mathsf{End}_i : \mathbb{N}_0, \mathsf{Bdg}_i : \mathbb{R}^+ \rangle$ *starts at time* St_i*, ends at time* End_i*, and has budget* Bdg_i*. For all* i*, it is required that* $\mathsf{St}_i \leq \mathsf{End}_i$ *and* $\mathsf{End}_i < \mathsf{St}_{i+1}$. □

A scheduling schema defines the time slots within which changes are to be implemented. Each slot has a budget, i.e., an upper bound on the implementation cost in that

slot's time frame. The specification of a scheduling schema depends on the characteristics and strategy of the organization, and on the changes to implement. For example, an organization may define a linear schema where all slots have the same duration and budget, while another may start with low effort, and increase it in later slots.

Definition 3. *Given a PDD model* Mdl *and a scheduling schema* $(\langle \mathsf{St}_1, \mathsf{End}_1, \mathsf{Bdg}_1 \rangle, \ldots, \langle \mathsf{St}_n, \mathsf{End}_n, \mathsf{Bdg}_n \rangle)$*, a set of* ***scheduling constraints*** Cstr *defines temporal restrictions on the allocation of the activities and the production of the deliverables of* Mdl *with respect to a time* $\mathsf{T} : \mathbb{N}_0$*:*

- actBefore(A, T)*: activity* A *shall be scheduled in a slot* i*, s.t.* $\mathsf{End}_i < \mathsf{T}$*;*
- delBefore(D, T)*: deliverable* D *shall be produced in a slot* i*, s.t.* $\mathsf{End}_i < \mathsf{T}$*;*
- actAfter(A, T)*: like* actBefore*, but* $\mathsf{St}_i > \mathsf{T}$*;*
- delAfter(D, T)*: like* delBefore*, but* $\mathsf{St}_i > \mathsf{T}$*;*
- actAt(A, T)*: activity* A *shall be scheduled in a slot* i*, s.t.* $\mathsf{St}_i \leq \mathsf{T} \leq \mathsf{End}_i$*;*
- delAt(D, T)*: deliverable* D *shall be produced in a slot* i*, s.t.* $\mathsf{St}_i \leq \mathsf{T} \leq \mathsf{End}_i$*.* □

Scheduling constraints enable imposing fine-grained temporal restrictions on the allocation of activities and on the production of deliverables. In Sec. 7, we will show how these constraints are a useful tool in our method to identify the most suitable plan.

Definition 4. *Given a PDD model* Mdl *and a scheduling schema* Schema*, a* ***plan*** *is a set* $\mathsf{Pln} = \{\langle \mathsf{A}_1, \mathsf{T}_1 \rangle, \ldots, \langle \mathsf{A}_n, \mathsf{T}_n \rangle\}$ *such that:*

- $\mathsf{A}_1, \ldots, \mathsf{A}_n$ *are all and only activities in* Mdl*, and*
- *for each* i*,* $1 \leq i \leq n$*, there exists exactly one slot* $\langle \mathsf{St}_j, \mathsf{End}_j, \mathsf{Bdg}_j \rangle$ *in* Schema *such that* $\mathsf{St}_j \leq \mathsf{T}_i \leq \mathsf{End}_j$*.* □

A plan is an allocation of all and only the activities of a PDD model into a scheduling schema. The activities shall be allocated within exactly one slot.

We say that a plan is *feasible* if it respects budget constraints, i.e., if for each slot, the sum of the implementation costs of the activities in that slot does not exceed the budget. We say that a plan is *contiguous* when all slots have at least one allocated activity. In this paper, we are concerned with the generation of feasible and contiguous plans.

The function $genplans$ brings together the concepts defined above and generates the feasible and contiguous plans for implementing a set of changes described by a PDD model in an organization, according to a specified scheduling schema, additional temporal constraints, and considering a cost function for implementing the changes.

Definition 5. ***Plan generation*** *is a function that returns all feasible and contiguous plans for implementing a set of changes in an organization. Formally,* $genplans : \mathcal{PDD} \times \mathcal{ORG} \times \mathcal{CSTF} \times \mathcal{SS} \times \mathcal{CSTR} \times \rightarrow 2^{\mathcal{PLN}}$*, and* $genplans(\mathsf{Mdl}, \mathsf{Org}, cst, \mathsf{Schema}, \mathsf{Cstr}) = \{\mathsf{Pln}_1, \ldots, \mathsf{Pln}_n\}$ *is such that:*

- Mdl *is a description of the changes to implement in PDD;*
- Org *is the organization where the changes are implemented;*
- cst *defines the cost of implementing the activities of* Mdl *in* Org *(Def. 1);*
- Schema *is a scheduling schema (Def. 2);*

- Cstr *is a set of constraints on scheduling (Def. 3);*
- *for each* i, $1 \leq i \leq n$, Pln_i *is a feasible and contiguous plan (Def. 4) for* Mdl *and* Schema *such that (i)* Pln *respects all constraints in* Cstr*; and (ii)* Pln *complies with the deliverable and capability dependencies in* Mdl. □

While *genplans* deals with hard scheduling constraints that cannot be violated, it does not consider the optimality of a plan. In this paper, we conceive plan optimality in terms of *incremental maturity improvement*: the changes should be introduced in accordance with a growing maturity level of the organization. The activities that contribute to capabilities with lower maturity levels shall be introduced first, followed by the activities contributing to higher maturity levels, up to the highest maturity. Def. 6 introduces the notion of penalty for a plan, i.e., its distance from an optimal plan where activities are introduced with a monotonic increasing level of maturity.

Definition 6. ***Plan penalty*** *is a function that returns the distance between a given plan and an optimal plan that would introduce all activities in increasing order of maturity. Let the predicate* $preceeds(\mathsf{A}, \mathsf{A}', \mathsf{Pln})$ *indicate that activity* A *is scheduled before activity* A′ *in* Pln. *The maturity of an activity* $mat(\mathsf{A})$ *is the lowest maturity level among the capabilities that the activity contributes to. Formally,* $penalty : \mathcal{PLN} \times \mathcal{PDD} \rightarrow \mathbb{N}_0$, *s.t.* $penalty(\mathsf{Pln}, \mathsf{Mdl}) = \sum_{\mathsf{A},\mathsf{A}':preceeds(\mathsf{A},\mathsf{A}',\mathsf{Pln})} \max(mat(\mathsf{A}) - mat(\mathsf{A}'), 0)$. □

When an activity contributes to multiple capabilities, we consider the capability having the lowest maturity level, for that activity is important for the organization to achieve that level. The penalty is the number of "steps" that have been skipped in the plan: for instance, consider only activities "Gather Requirements" and "Analyze Customer Problems" in Fig. 4. The former activity has lowest maturity level 1 (it contributes to capability A in area requirements gathering), while the latter has lowest maturity level 6 (the contributed capability with lowest maturity is D in requirements gathering). If the former activity is introduced before the latter, the plan penalty would be 5.

We call a plan *optimal* when its penalty is zero, and *quasi-optimal* when its penalty is greater than zero but lower than Δ. This number Δ is domain-specific, and it depends on the number of activities in the plan, the organization, the cost of activities, etc.

In this paper, we limit ourselves to a special kind of scheduling schema, where for each slots i ($1 \leq i \leq n$), $\mathsf{St_i} = \mathsf{End_i}$, and the slots are such that $\mathsf{St_1} = 1, \ldots, \mathsf{St_n} = n$.

We do not consider constraints related to organizational resources (other than the abstract unit Cost) and human factors (such as worker resistance), as such factors are harder to attribute to single activities and deliverables. The method we propose is a support tool for analysts, and does not replace their role as decision makers.

5.2 A Method for Identifying and Refining Plans

We present an elaboration of the **Change planning** step in Sec. 2 that uses *genplans* (Def. 5) for identifying plans and for refining them to fit well with the organization at hand. This method, illustrated in Fig. 5, helps to restrict or widen the space of alternative plans, depending on the plans that the function *genplans* returns.

The process begins with two preparatory steps that provide the inputs to the *genplans* function: the planning context is defined (the changes to implement, the organization,

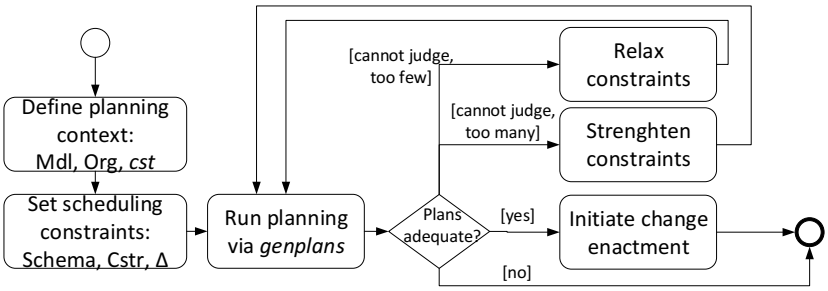

Fig. 5. A method for identifying and refining (quasi-)optimal plans

and the cost function), and an initial version of the scheduling constraints is created (the scheduling schema and constraints, and the quasi-optimality upper bound Δ). This input feeds the *genplans* function, which returns all optimal and quasi-optimal plans. The analyst then judges the adequacy of the returned plans:

- *Adequate*: a plan is identified and *change enactment* starts (see also Fig. 1);
- *Uncertain*: a final decision cannot be made at the moment, due to scarcity or excess of results. In order to identify a better plan, the analyst can modify the constraints:
  - *Relaxation*: when too few plans are returned, the scheduling constraints can be relaxed/weakened. Time slots can be merged to spread changes over a longer period with higher budget. Temporal scheduling constraints can be relaxed, or the quasi-optimality upper bound Δ increased. More intrusive options (from the organization's perspective) are to increase the number of slots or the budget.
  - *Strengthening*: if too many plans are returned, the analyst cannot make an informed choice, and the constraints can be tightened: the slots in the schema and their budget can be reduced, and temporal constraints can be added.
- *Inadequate*: no satisfactory plan can be devised in the current planning context, irrespective of the scheduling constraints. This terminates the process, which can be re-executed in a different context.

6 Realization

We have developed a graphical modeling tool for PDD within MetaEdit+ [20]. This tool is built around the Graph, Object, Property, Relationships, Role (GOPRR) metamodel, which we used to describe the PDD metamodel; instances of the latter metamodel are PDDs. The editor enforces syntactical rules, thereby ensuring the well formedness of the model. In addition, we have developed a set of code generators that transform any PDD model into a set of Datalog statements, which are needed to generate plans.

We have realized the mechanisms for identifying optimal and quasi-optimal plans via logic programming (specifically, via the disjunctive Datalog engine DLV [15]). The program consists of a set of inference rules (Tab. 2) that returns feasible and contiguous plans for a given input. The input consists of the Datalog statements that are generated from the PDD model, the temporal constraints, and the Δ quasi-optimality upper bound.

The following extensional predicates formalize in Datalog the primitives of our languages (Sec. 4 and Sec. 5): activity(A) states that A is an activity; produces(A,D) and updates(A,D) denote the relationships between activities and deliverables; requires(D1,D2) indicates that namesake relation between deliverables in Sec. 4. The predicate atLeastOnePart(D) (allParts(D)) says that D requires at least one (all) of its parts, each part being stated through hasPart(D,D1); cost(A,N) states that the deployment of activity A costs N (a natural number); slot(N1,N2) states that slot at time N1 has a budget of N2 cost units; actAt(A,N), delAt(D,N), actBefore(A,N), delBefore(D,N), actAfter(A,N), delAfter(D,N) state constraints telling that activity A or deliverable D shall be scheduled at/before/after slot N; contributes(A,C) states that activity A contributes to capability C; depends(C1,C2) says that capability C1 shall be implemented strictly after capability C2; maturity(C,N) states that capability C has maturity level N.

Tab. 2 presents the inference rules (syntax head :- tail.) to generate plans. By default, the program returns all possible plans that satisfy the constraints. A plan is defined as a set of predicates chosen(A,N), each stating that the deployment of activity A will be scheduled in slot N. Rules without a head (those starting with ":-") are integrity constraints: the condition expressed in the tail shall not become true in any model.

Table 2. Core disjunctive Datalog inference rules for deriving plans

Id	Rule definition
1	chosen(A,T) :- activity(A), slot(T,_), not chosenNotT(A,T), not missingReq(A,T).
2	chosenNotT(A,T) :- activity(A), chosen(A,T2), slot(T,_), slot(T2,_), T2!=T.
3	:- activity(A), 0=#count{T: chosen(A,T)}.
4	:- slot(T,_), 0=#count{A : chosen(A,T)}.
5	chosen(A,T) :- actAt(A,T).
6	:- delAt(D,T), T1=T-1, producedAt(D,T1).
7	:- delAt(D,T), not producedAt(D,T).
8	:- actBefore(A,T), chosen(A,T1), T1>=T.
9	:- delBefore(D,T), 0=#count{T1 : producedAt(D,T1), T1<T}.
10	:- actAfter(A,T), chosen(A,T1), T1<=T.
11	:- delAfter(D,T), 0=#count{T1 : producedAt(D,T1), T1>=T, not producedAt(D,T)}.
12	:- slot(T,B), #int(C), C=#sum{Cs,Act : cost(Act,Cs), chosen(Act,T)}, C>B.
13	missingReq(A,T) :- uses(A,D), #int(T), not producedAt(D,T), T2=#max{T1 : slot(T1,_)}, T2<=Z.
14	producedAt(D,T) :- chosen(A,T1), #int(T1), #int(T), T1<=T, produces(A,D), T3=#max{T2 : slot(T2,_)}, T3<=Z.
15	uses(A,D1) :- produces(A,D), requires(D,D1).
16	uses(A,D) :- updates(A,D).
17	requires(D1,D3) :- requires(D1,D2), requires(D2,D3).
18	hasOnePart(D) :- atLeastOnePart(D), produces(A,D), chosen(A,T), hasPart(D,D1), producedAt(D1,T).
19	:- atLeastOnePart(D), not hasOnePart(D).
20	:- allParts(D), produces(A,D), chosen(A,T), hasPart(D,D1), not producedAt(D1,T).
21	:- contributes(A1,C1), contributes(A2,C2), A1!=A2, depends(C1,C2), chosen(A1,T1), chosen(A2,T2), T1<=T2.
22	minmaturity(A,M) :- activity(A), M=#min{L : contributes(A,C), maturity(C,L)}.
23	penalty(Cs) :- #int(Cs), Cs=#sum{C,A1,A2 : chosen(A1,T1), chosen(A2,T2), T1<T2, minmaturity(A1,M1), minmaturity(A2,M2), M1>M2, C=M1-M2}.

Rules 1–3 ensure that all activities are assigned to exactly one slot. Rule 4 guarantees that all slots have at least one assigned activity. Rules 5–7 deal with actAt and delAt constraints: the activity (the deliverable) shall be scheduled (produced) in the specified slot. Rules 8–9 and rules 10–11 guarantee the fulfillment of the similar constraints in the before/after variant. Rule 12 ensures that the sum of the costs for implementing the activities in a slot does not exceed the slot budget. Rules 13–14 ensure that deliverables are produced before their use. Rule 15 states that an activity A uses a deliverable D1 if X produces D, and D requires D1. Rule 16 says that updating a deliverable implies using it. Rule 17 says that the requires relationship is transitive. Rules 18–20 take care of atLeastOnePart(D) (allParts(D)): at least one part of D (all parts) shall be available when D is produced. Rule 21 handles the depends(C1,C2) relationship between capabilities: all activities that contribute to C1 are scheduled strictly before any activity that contributes to C2. Rules 22–23 compute the penalty of the plan as in Def. 6.

The program can be run with different parameters to return only optimal and quasi-optimal plans. By adding the rule :∼ penalty(Cs). [Cs:1], only the plans with minimum penalty are listed (":∼" is a weak constraint that DLV optimizes by returning the models that minimize it). In order to return all quasi-optimal plans, the parameter "-costbound=Δ" can be specified when executing DLV.

7 Illustration

We apply our planning method to the scenario of Example 1 and Fig. 4. Kano analysis uses a two-dimensional quality model used for the analysis of customer requirements, which is useful to elicit customer needs about a service or product under design. It uses two types of questionnaires and an evaluation table for classifying the requirements into different categories [13]. We show how the company can use our method to identify an incremental plan to introduce the 15 activities and 13 deliverables of Kano analysis.

Step 1: Define the Planning Context. We begin with the creation of a PDD model that describes the changes to introduce (Kano analysis, as in Fig. 4), the organizational context (our example organization), and the cost function that returns the costs for each activity. In this example, we use a simple cost scheme, where we use natural numbers in the range [1,5] to describe the complexity of implementing and learning each activity. For each activity, we add a fact as input to our datalog program: cost(analyze_customer_problems, 4). cost(analyze_product_environment, 2)., etc.

Step 2: Set Scheduling Constraints. This activity involves the specification of the number of slots for implementing the plan, and the budget for each of them. Here, we define four slots, each with a budget of nine cost units (adopting the same unit as for activities cost). We do not define any temporal constraint, and we set $\Delta = 25$.

Step 3: Run Planning. When we run our planner with the settings above, we obtain 1,442 plans, with penalties between 8 and 22. Making a choice at this point would obviously be difficult; moreover, no optimal plan exists (no plan has penalty 0). Some constraints have to be introduced.

Step 4: Strengthen Constraints. The company wants to have a running implementation of Kano analysis in slot 0. This requires to have at least a table of results based on requirements from the customer and the product environment. More advanced tools, such as the QI-index plots and the CS-coefficient plots, can be introduced later. To such extent, the following temporal constraints are added:

delAt(tables_of_results, 0). delAfter(cs_coefficient_plots, 0). delAfter(qi_index_plots, 0). actAt(analyze_product_environment, 0). actAt(evaluate_answer_frequencies, 1).

When we re-run the planner, we obtain no results. By forcing the planner to schedule the activity that produces the table of results at slot 0, the require relationships between the deliverables imply that several other activities have to be schedules in slot 0 as well. Their total cost exceeds the budget of slot 0. This forces us to relax some constraints.

Step 5: Relax Constraints. To cope with the required effort for implementing the table of results in slot 0, the organization can either combine the effort of multiple slots, thereby lengthening the implementation time, or allocate more resources to slot 0. We assume the analyst opts for the latter: the budget of slot 0 is raised to 20, but a slight reduction in the overall budget is required (-4 units); the remaining units are allocated in two slots with budget 6, and the last slot is removed.

Table 3. Slot allocation for the activities of Kano analysis: the (quasi-)optimal plans in Step 5

Activity	Plan A	Plan B	Plan C	Plan D	Plan E
analyze_customer_problems	0	0	0	0	0
analyze_product_environment	0	0	0	0	0
analyze_conditions_of_applications	2	2	2	1	1
gather_requirements	0	0	0	0	0
formulate_functional_questions	0	0	0	0	0
formulate_dysfunctional_questions	0	0	0	0	0
assemble_questionnaire	0	0	0	0	0
conduct_customer_interview	0	0	0	0	0
combine_func._and_dysfunc._answers	0	0	0	0	0
list_results_of_individual_criteria	0	0	0	0	0
evaluate_answer_frequencies	1	1	1	1	1
construct_satisfactory_ranking_scale	1	1	2	2	1
construct_importance_ranking_scale	1	2	1	1	2
determine_quality_improvement_index	2	2	1	2	2
calculate_satisfaction_coefficient	2	1	2	2	2
penalty	8	8	8	12	12

When we re-run our planner, we obtain 5 plans. Tab. 3 shows them and outlines their differences via a gray background color. Two of them have a penalty of 12, the other three have a penalty of 8. The analyst is free to consider a restricted set of plans. The choice is between introducing both plots in slot 2, implementing QI index plots first and then CS coefficient plots, or vice versa.

8 Discussion and Future Directions

We have presented a method that assists analysts in the planning phase of method evolution, i.e., to identify plans for implementing a set of changes in an organization. Our method enables representing changes via PDD models, and is supported by our graphical modeling tool. The method includes the automated generation of plans that comply with scheduling constraints, and that maximize incremental growth in maturity, by trying to introduce changes according to an increasing maturity level. We also propose a process that guides analysts in refining plans by strengthening and relaxing constraints.

This work is performed within the context of the Online Method Engine, and it touches upon several related fields. We discuss our approach in the light of these fields.

Software Process Improvement. Research in the area of software process improvement has produced effective frameworks to determine what to change, including CMMI [6] and SPICE [8]. We complement these works by proposing a method for planning the implementation order of these changes.

Situational Method Engineering. This discipline deals with describing, constructing and adapting software development methods for a specific situational context, thus promoting reuse of standardized approaches while maintaining flexibility [11,12,3]. In this research, we employ the method fragment concept [5] for compatibility with the OME system; however, other notations can be used, as long as they satisfy the requirements of Sec. 4.1. There has been some work related to the notion of method evolution [18,17]. Most approaches in method evolution consist of manual activities, although some approaches support (semi-)automatic method construction [1].

Automated Planning. The problem of identifying a plan to reach a given goal is well-known in Artificial Intelligence [10]. Recent planners are able to deal with sophisticated planning constraints on state trajectory, preferences, soft constraints, and plan quality [9]. Our approach differentiates from existing solutions in that it employs a capability-driven planning policy that takes in to account deliverable-based constraints, as opposed the activity-based constraints that are typical of AI planning. We do not preclude that an extended version of our framework could employ PDDL.

Project Management. The implementation of a set of changes in the methods of an organization is usually executed in the form of a project. Project management is a very mature field, which offers effective mechanisms and tools to deal with change by planning, scheduling, and controlling it [14]. Our approach is inspired by this field, but focuses on a very specific type of scheduling that relates to method change.

We have focused only on introducing new fragments; the next step is to consider the removal and replacement of fragments. We will also explore the preceding step of *method construction*. Furthermore, we plan to convert our prototype into a comprehensive tool that supports the analysts in the plan refinement process by recommending possible refinements. We will evaluate the efficacy of our approach with case studies in the industry, and based on the feedback from practitioners, we will extend the supported constraints. Finally, we aim to assess the scalability of our reasoning techniques.

References

1. Aharoni, A., Reinhartz-Berger, I.: Semi-automatic composition of situational methods. Journal of Database Management 22(4) (2011)
2. Baddoo, N.: De-motivators for software process improvement: an analysis of practitioners' views. Journal of Systems and Software 66, 23–33 (2003)
3. Becker, J., Knackstedt, R.: Configurative method engineering – on the applicability of reference modeling mechanisms in method engineering. In: Proc. of AMCIS, pp. 1–12 (2007)
4. Bekkers, W., Spruit, M., van de Weerd, I., van Vliet, R., Mahieu, A.: A situational assessment method for software product management. In: Proc. of ECIS, pp. 22–34 (2010)
5. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information and Software Technology 38(4), 275–280 (1996)
6. CMMI Product Team: Capability Maturity Model Integration (CMMI). Tech. Rep. December 2001, Carnegie Mellon Software Engineering Institute, Pittsburgh (2002)
7. Diaz, M., Sligo, J.: How software process improvement helped Motorola. IEEE Software 14(5), 75–81 (1997)
8. Dorling, A.: SPICE: Software Process Improvement and Capability Determination. Software Quality Journal 2(4), 209–224 (1993)
9. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3 – the language of the fifth international planning competition. Tech. rep., University of Brescia (2005)
10. Ghallab, M., Nau, D., Traverso, P.: Automated planning: theory & practice. Elsevier (2004)
11. Harmsen, F., Brinkkemper, S., Oei, J.L.H.: Situational method engineering for informational system project approaches. In: Proc. of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle, pp. 169–194 (1994)
12. Henderson-Sellers, B., Gonzalez-Perez, C.: Granularity in conceptual modelling: Application to metamodels. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 219–232. Springer, Heidelberg (2010)
13. Kano, N., Seraku, N., Takahashi, F., Tsuji, S.: Attractive quality and must-be quality. The Journal of the Japanese Society for Quality Control 14(2), 39–48 (1984)
14. Kerzner, H.R.: Project management: a systems approach to planning, scheduling, and controlling. Wiley (2013)
15. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
16. Pino, F.J., Pedreira, O., García, F., Luaces, M.R., Piattini, M.: Using Scrum to guide the execution of software process improvement in small organizations. Journal of Systems and Software 83(10), 1662–1677 (2010)
17. Ralyté, J., Rolland, C., Ayed, M.B.: An approach for evolution-driven method engineering. In: Information Modeling Methods and Methodologies, pp. 80–202 (2005)
18. Rossi, M., Ramesh, B., Lyytinen, K., Tolvanen, J.P.: Managing evolutionary method engineering by method rationale. Journal of the Association for Information Systems 5(9), 356–391 (2004)
19. Simon, H.A.: Rational choice and the structure of the environment. Psychological Review 63(2), 129–138 (1956)
20. Tolvanen, J.P., Rossi, M.: MetaEdit+: Defining and using domain-specific modeling languages and code generators. In: Crocker, R., Steele Jr., G.L. (eds.) Proc. of OOPSLA (2003)

21. Vlaanderen, K., van Stijn, P., Brinkkemper, S., van de Weerd, I.: Growing into Agility: Process Implementation Paths for Scrum. In: Dieste, O., Jedlitschka, A., Juristo, N. (eds.) PROFES 2012. LNCS, vol. 7343, pp. 116–130. Springer, Heidelberg (2012)
22. Vlaanderen, K., van de Weerd, I., Brinkkemper, S.: On the design of a knowledge management system for incremental process improvement for software product management. International Journal of Information System Modelling and Design 3(4), 46–66 (2012)
23. van de Weerd, I., Brinkkemper, S.: Meta-modeling for situational analysis and design methods. In: Handbook of Research on Modern Systems Analysis and Design Technologies and Applications, ch. III, pp. 35–54 (2008)
24. van de Weerd, I., Brinkkemper, S., Versendaal, J.: Concepts for incremental method evolution: Empirical exploration and validation in requirements management. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 469–484. Springer, Heidelberg (2007)