Supporting a Project Leader within an Evolving Product Library

Edward Fischer
Institute of Computer Sciences, Software Systems Engineering
Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld
Germany
ef@tu-clausthal.de

Abstract: - One of a project leader’s central tasks is to schedule work products and keep track thereof. This is not a trivial issue as software systems are developed neither in a monolithic way, nor correct the first time. Each revision of a single work product may imply the need to check several consistencies with other work products, including various consequences: there might be, i.e., the need to produce new work products, change some specific parts of existing ones, or dismiss them in whole. Practice has shown that project leaders fail to continuously and manually check all dependencies on each single change a project’s product library (the set of all documents and versions thereof produced in a project) encounter. Underestimation of remaining workload and late detection of problems with costly resolutions are only two of resulting consequences. This paper identifies a number of typical planning issues, elaborates the underlying problem they have in common, and presents a solution therefore.

Key-Words: - Project Management, Project Controlling

1 Introduction

1.1 Motivation
In a project, controlling is one of the central tasks, a project leader has. This is about setting up what has to be done (target, i.e. work packages), comparing it to what has been done (performance), and order corrective measures to keep the project in time, budget and quality.

Creative work, as it is the case with software development, cannot be fully pressed into a strictly sequential scheme following the first-time-right approach. Next to iterations in the large searching for proper requirements (validation), we also have iterations in the small, addressing the verification issue, which covers quality assurance loops (i.e. testing) with corresponding corrective steps. As a result, a project leader is constantly faced with changing document contents, and various controlling issues like the need to adjust his schedule (target) or the assessment of the current situation (performance).

Practice has shown (i.e.[13]), that even in mid-sized Projects having 500 documents at the end, it is virtually impossible to keep a schedule up to date within occurring changes by hand: A project leader has not only to plan work products per system element, but also has to schedule work products for work products, too. For instance, a review document has to be produced for each specification document. This quantification (“for each”) might be restricted by a condition: i.e., a hazard analysis document might only be produced for each system element being rated as critical. This, “being critical”, requires a project leader to look at document contents while considering all kinds of consistency rules and other dependencies between work products.

Various tools do exist, mostly based on workflow concepts, which are able to instantiate actions (for restoring consistency, i.e.) out of current product library contents. However, they assume that actions, once instantiated, have to be finished before any changes can be made to those work products of the product library, they were instantiated from. This leads to long-term iterations replicating creative work in an unrealistic way.

Yet, tool support allows controlling not to go beyond a superficial level. Most commonly, revision- and consistency-related activities are missed to be kept up to date within an evolving product library. Underestimation of remaining workload and late detection of problems with costly resolutions are only two of resulting consequences.

1.2 Goal and Contribution
Eventual goal of this work is to provide better tool support for controlling a project. In that case, “better” means that a project leader should be able to keep his schedule (target) and assessment (performance) detailed down to the level of document contents, and up to date within an evolving product library. Evolving means, that existing work products in the product library may be changed any time, as a consequence of creative work.

In this paper, we will focus on the fundamental problem which has to be solved in order to be able to provide better tool support.

Therefore, as this paper’s contribution, the fundamental problem is outlined and solved in general. In addition, a set of concrete controlling problems is
gathered, along with a guideline to adopt the general solution to resolve them.

1.3 Structure of this Paper
Section 2 shows a row of controlling problems a project leader typically has. Section 3 explains the fundamental problem with an evolving product library those problems have in common, while section 4 makes clear, why current approaches did not yet solved that problem. In contrast, section 5 introduces a general solution to the fundamental problem and provides a guideline to adopt that solution to those various controlling problems in section 2. This paper comes to an end with a brief sketch of remaining further work in section 6 and concludes in section 7.

2 Concrete Controlling Problems
Work products are produced in every software development project. In the least structured case, this is just code. Adding more structure to a project, we see work products spanning from technical documents like test cases, over methodical like specifications and architecture documents, to more economic aspects like contracts and change request lists.

We call the set of all work products (and versions thereof) produced in a project the product library. Please note, that this notion abstracts from the technical organisation like a central repository or any distributed mechanism, as well as abstracts from the order, in which work products are produced or updated. In fact, dependencies between work products exist in any case, leading to various controlling problems, which are described next.

2.1 Scheduling new Work Products
At some point in time in a project, a work product is revised. For instance, a new system element is identified and enlisted in the system architecture document. Now, following consequent work products have to be scheduled:

- a software specification document for concretizing the new system element’s function, along with a review document to check that specification.
- a work product for the new system element itself (which would be code and/or binaries)
- test specification and test protocol documents, as well as test procedures later on – for each test case identified in the test specification.

In practice, so far, when team members create revisions, they “remember” to add new documents (or versions thereof) to the product library afterwards. A project leader might be informed in an abstract way during a meeting (i.e. “I’ve changed the architecture”). More likely, the change will go unnoticed. In fact, remembering is not one of people’s strengths. Thus, remaining workload is underestimated making it hard to maintain meaningful long term plans: currently forgotten work packages will pop out of the sudden later on. Enforcing a strict workflow process, however, will be ignored or will lead to opposition in respect to the organizational overhead.

2.2 Dismissing obsolete Work Products
As a counterpart to the previous problem, changes may render existing work products obsolete. Although removing seems to be a simple operation, it still requires a project leader to look into all document contents and track all dependencies between them. Especially when the project leader is not the only one who is authorized to make changes.

2.3 Check whether everything is up-to-date
Consider that an interface specification document $d$ has been revised by adding an optional parameter $p$ to some function therein. Now, there is the need for scheduling a new version for each existing test case, a new version for all interface’s implementations, along with a corresponding new test protocol version.

Right after introducing that new parameter $p$ to document $d$, that document can’t be rated as finished anymore – at least not until all the consequent versions of dependent work products have been produced. This requires a project leader to continually check all of those work products, before he can eventually set document $d$ as finished. Thus, calculating and keeping track of work product states creates an additional overhead.

2.4 Planning with unfinished Work Products
Taking a second look at work product states, we see that being “finished” is not the only one state, where consequent actions have to be started. In fact, quality assurance needs some kind of a “submitted” state. Thus, a project leader is not only faced with finished or out-of-date results, but with intermediate ones, too.

Until a document currently being revised, is accepted as finished, a project leader has to use the present finished version. This in turn, complicates the way he retrieves work products from the product library. In addition, some documents might relate their contents to different iterations (or releases if you prefer). So a document rated as “finished” might only imply, that the contents of the current iteration have been finished – without saying a word about the overall project plan.

2.5 Dealing with (adaptive) Process Models
Process models can be seen as a reminder to prevent important items of being forgotten to be scheduled. As each project is different, a process model can only enlist the types of work products – but not the concrete amount
of documents or versions thereof which have to be scheduled. However, some work product types are irrelevant for some projects, which implies that no documents needs to be produced of that type at all. I.e., a contract is not needed in an in-house-development. Modern process models, like V-Modell XT [5] or Hermes [8] provide a mechanism called “tailoring”, which allows a project leader to adapt a process model to his current project. This way, he can start a project with a minimal set of work product types he has to take care of. Changing requirements in that project, however, might demand him to readjust his tailoring. This typically result in new work product types added, along with various corresponding dependencies. Eventually, the project leader has to go through the whole product library to find out which of already existing work products have to be revised in order to fit the newly tailored process model.

3 Class of Controlling Problems

Instead of finding an own, narrow-fitting solution for each concrete controlling problem, we will first bring all of those problems (see section 2) to one common denominator, we call the problem class. This way, a solution for the problem class (see section 5) resolves all concrete controlling problems at once.

3.1 Common Denominator

Let us introduce a simple mechanism to describe how a project evolves: At each point in time, a project can be determined through its product library, which is a set of work products. A work product in turn is a set of versions. On that basis, every change made in a project can be described as adding a new version to the product library (see Fig. 1).

![Figure 1. Simple data model of a project](image)

As those controlling problems of section 2 cover document contents, working states and process models as well, we extend the denominator in Fig. 1 by respective constructs, leading to Fig. 2.

![Figure 2. Problem class](image)

Now, the structure of each concrete controlling problem can be described as an instance of this problem class. Fig. 3 shows an example therefor, for the case of scheduling new work products (compare section 2.1).

![Figure 3. A concrete controlling problem shown as an instance of the problem class](image)

In detail, white-filled objects represent the process model which is kept the same throughout the whole project. Grey-filled objects represent the product library at some time point t1 (black text & border) and time point t2 (black and red text & border). Grey-filled objects with orange text and border denote consequent measures for changes made between t1 and t2. In that case, adding UnitB to the critical list implies the need of scheduling a respective specification document, which in turn might imply a review protocol (due to a different dependency).

3.2 Fundamental Problem

Seeing that each change of a product library can be treated as adding a new version, we can outline two kind of changes: First, the loose one, which originates from creative work and thus is free to happen without any deterministic foreseeable cause. Second, the constrained one, which produces consequent results or resolves inconsistencies caused by loose changes. Thus, a constrained change originates from dependencies between work products.
As dependent work products may undergo numerous loose changes in parallel (remember the way creative work is actually done), dependencies have to be evaluated multiple times against the very same work products. This, however, leads to a redundancy problem as shown in Fig. 4: The first time dependencies are evaluated, at time point t1, a new specification work product for UnitA is scheduled. At time point t2, after the rated-as-critical decision was reconsidered, dependencies are evaluated a second time. Now, we only need one new specification work product to be scheduled – namely for UnitB. However, a straight-forward re-evaluation would schedule an additional (redundant) specification work product for UnitA. A mere duplication check does not help either, since that specification produced at time point t1 might have been changed meanwhile.

Now, team members manually synchronize their parts of the overall schedule with contents of their work products. Although this way, project leader’s workload is partly transferred to other team members, no real improvement in reducing the total workload of manual synchronization is achieved.

4.3 State Machines
In contrast to both previous approaches, state-machines provide the first simple mean of automatic instantiation. I.e., instead of creating two issues by hand – one for implementing a system element and one for testing it – a project leader needs only to create one instance of their combined process. Both issues would be treated as the two states of that process. An instance of that process would start with the implementation state, which would be displayed as an ordinary issue. After a team member has manually ticking off that issue, the state machine would go on to the next state, automatically showing a respective testing issue.

However, such state machines would prohibit the implementation to be changed until testing and any subsequent states are passed through. Second, a project leader would still have to manually instantiate the overall process for, i.e., each system element (remember that state machines cover the evolution of one object only). Third, a project leader can only see currently open issues. There is no support in planning the whole project including all steps already clear to come.

4.4 Workflows
Workflows can be seen as a generalization of state machines, providing more versatile means of automatic instantiation. I.e., a state (or step if you prefer) of a workflow process may

• denote an action which has to be ticked of manually

• invoke external applications or functions, in order to automatically, i.e., obtain and evaluate information from the product library, or add new work product versions into.

• Automatically instantiate (other) workflow processes

However, using automatic instantiation again implies the need to prohibit evaluated work products from being evaluated again or changed, until all thereof instantiated processes have been finished. Otherwise each further evaluation of a dependency would lead to either additional, redundant work products being scheduled, or anomalies if the work product in question was changed. Preventing at least the redundancy effect would require team members to resolve all newly instantiated scheduling items in one shot before the next evaluation. This isn't feasible in many ways: First, team members would be pressed into a clocked working process, expanding each clock to the longest single task. Second,
there would be no way to check dependencies until all team members have finished. And finally, third, even if they would resolve inconsistencies with the first shot, changes made therefor could trigger another effects. Thus, a “workflow-chain” is created, which in turn requires the team to finish all new issues immediately – all the way down to some stable state, which implies a longer time period. Then again, while they are at it, an unavoidable need for a loose change of an initial document (in that chain) might occur. This would have to wait until the chain is finished, producing a methodical dead lock.

In the end, workflow tools (like Serena Teamtrack [2]) are of great help supporting sequential processes (i.e. business transactions), but fail to support creative processes like a software development project.

4.5 Generating Work Products

Going into a quite different direction, there are approaches ([3], [4]) leaving aside actions and order in which work products are produced. More precisely, the idea is to have an integrated, central repository covering all contents of all work products only – but not work products themselves. A project leader uses a generator to generate a set of work products (consistent to a process model he wishes to use) out of that repository. As benefits, we can

- avoid parsing “office documents” for obtaining and evaluate work product contents.
- keep records of project states, through a set of work products being consistent to any supported process model.

However, this is where this approach ends, too. There is no support for prognostic planning by using a process model – we just get records of what we have done so far, seen through the eyes of a given process model. On the tool side so far (i.e. Innovator [14]), only system-related work products are covered yet (including architecture and specification documents, i.e.), excluding, among others, the project plan.

4.6 Upshot

As we have shown, none of existing approaches has a viable solution to manage an ever-changing product library of creative processes. Thus, none of those problems described in section 2 (or the converged problem class in 3.1) has been solved yet.

5 Solution

In section 5.1, we start informally by presenting the solution idea, which will be brought to a more formal representation as a pattern in section 5.2. Finally, section 5.3 shows how that pattern can be used to resolve controlling problems of section 2.

5.1 Idea

We treat all dependencies between work products as a pair of a trigger and a photofit picture. A trigger describes a pattern of when a consequent result is missing or an inconsistency has to be resolved, while a photofit picture defines which work products should be present, produced if missing, or updated by a new version if not meeting consistency rules or additional content requirements yet.

For example, imagine a dependency named “Architectural Influence” which, in few words, states that for each unit identified in a system architecture document, a corresponding specification document shall be produced, if that unit was also rated as critical. Fig. 5 shows how this dependency would look like through a trigger and a photofit picture: The trigger contains quantifiers X and Y, defining the space of interesting constellations of work products and their contents in a project state, as well as conditions for restricting that space to those where something should be present. More precisely, that space contains each combination of a (reference to a) system architecture document X and a (reference to a) unit Y. First condition removes all combinations (X,Y) where unit Y has not been identified in corresponding system architecture Y, while the second condition removes all combinations where unit Y was not rated critical. On the other side, the photofit picture states that a specification document should be present for each combination left, or produced if missing. For sake of simplicity, no additional content requirements should be met here.

Figure 5. Structuring dependencies through a trigger and a photofit picture

First step in evaluating a dependency against a given project state is to find each combination of valid assignments for the trigger’s quantifiers. As the second
step, we produce a new work product, according to the photonfit picture, for each assignment combination found. In the following, a pair of an assignment combination and a reference to the work product produced due to it, is called a dependency instance (see Fig. 6 for a scheme).

To deal with work product changes, we keep dependency instances as part of the project state. If a dependency (like our Architectural Influence) is re-evaluated, each possible assignment combination is skipped, if current project state has a dependency instance with the same assignment values already (see Fig. 7).

Thus, new work products can only be produced, if new assignment combinations are found. The other way around, work products are no longer needed, if assignment combination are no longer found during re-evaluation.

5.2 General Pattern and Field of Application

At first glance, the idea described in section 5.1 seems not to be restricted to the domain of software development. In fact, Fig. 8 and Fig. 9 show a more formal version of that idea as a pattern with domain-neutral names.

However, the nature of triggers and photonfit pictures restrict possible applications to processes. Further, this pattern add no value to standard processes, where no parallel changes occur (i.e. making a hotel reservation). Thus, in contrast to state machines, workflows or similar, we get only more value for the additional...
investment (of storing and comparing dependency instances), if dependent elements are changed in parallel. This again is very specific for creative work, where a sequential process is hard to be defined, or cannot be defined at all (see motivation).

5.3 Adoption Guideline

Having the pattern alone, we have not solved those concrete controlling problems of section 2 yet. So we have to integrate this pattern with the common denominator elaborated in section 3.1 (Fig. 2). This is accomplished in the following way: First, in Fig. 10, we identify which entities of the problem class we consider to be anytime changeable during a project. Each of them will be treated as a specialization of a quantifiable element.

Second, with a set of different quantifiable Elements, we also have different possibilities what a photofit picture might want. I.e., a new ProductInstanceVersion or a new ProductInstance might have to be added. As we have a relation between those two, deleting a not longer needed ProductInstance implies to delete all respective Versions, too. Thus the effect of a photofit picture can be different from case to case. Which cases are useful, is shown in Fig. 11, where we have identified a number of dependency kinds, with respective effects.

Concrete dependencies, like the “Architectural Influence” are supposed to be implemented as a specialization of one of those kinds, overwriting trigger and photofit picture, leaving the effect and the general evaluation semantics the same. Fig. 12 summarizes the dependency inheritance stack, and shows which part of the overall semantics is defined at which step.
6 Further Work: Tool Concept

Up to this point, we have discussed conceptual problems of the eventual goal of a better tool support for controlling a project (see section 1.2). As this was solved in the previous section 5, we will now present our first drafts of a tool concept making those results visible.

First, we discriminate between two kind of work products. We call work products representing system elements as “SE”s and corollary work products (like specifications) as “CWP”s.

![Figure 13. Concept for visualizing Dependencies](image)

**Figure 13.** Concept for visualizing Dependencies

Fig 13 shows, how this distinction can be used to arrange all work products in relation to dependencies between them: the backbone is made out of SEs: All work product contents, describing a single system element (including its hierarchy position as well as decisions whether it has to be tested, specified or whatsoever), are displayed right in the corresponding box (or “between” the boxes for the hierarchy as a special case). Any usual input type like dropdown, check box and so on, can be used to modify those work product contents.

For each SE, all CWPs “it has” produced (due to dependencies) are positioned directly around. Following this pattern, all CWPs a CWP “has caused”, are placed around the causer, too.

Missing work products, which should have been produced due to dependencies, but were not produced yet, are displayed through a contour. This contour functions as a button to create that work product. As a short cut alternative for creating work products manually, they can be created immediately during the dependency evaluation. However, work products which were not touched since, can be displayed “sketchy”. This way, a project leader can easily play around by modifying work products and their contents, while seeing all entailing consequences right away.

7 Conclusion

In this paper, we have examined the fundamental problem with controlling a project within an anytime-changing product library in section 3. Beforehand, in section 2, this paper showed in which various ways this fundamental problem appears in practice. However, those problems has not been solved by present approaches (see section 4), so, in section 5, this paper presented an universal solution therefore, as well as a guideline to adopt that solution to those problems in practice. However, it was also made clear, that the solution is restricted to creative processes with software development being a popular representative, though. In conjunction with a tool concept sketch, the foundation was laid to eventually achieve better tool support for controlling a creative (software development) project.

References:


