Towards Optimizing the Schedule of Software Projects with Respect to Development Time and Cost

Frank Padberg[*]
Fakultät für Informatik
Universität Karlsruhe, Germany
padberg@ira.uka.de

Abstract. A new probabilistic model for software projects is presented which explicitly takes a scheduling strategy as input. The model is a basis for applying stochastic optimization techniques to compute schedules which minimize the development time and cost. Process simulation is combined with the optimization techniques to cut down their computing time.

Keywords. Optimal Software Project Scheduling, Process Simulation, Probabilistic Process Models.

1 Introduction

Staff is the most valuable resource today in software development. There has been a dramatic increase in demand for software over the last decade. As a result, the supply of software engineers is not even close to meeting the current demand. Since the software business is still growing and it takes years of university education and training to become a good software engineer, the situation is likely not to change in the near future.

In view of the shortage of software developers, it is even more important than ever that software project managers plan and schedule their development projects in such a way that the available developers are deployed as effectively as possible. Scheduling means to bind developers to activities on a time scale, answering the question: who must do what, and when? Planning and scheduling a software project is especially difficult though, for a number of reasons.

- Software is an immaterial product. Thus, tracking the actual progress of a software project is difficult, making it hard for a manager to tell when it's time to take controlling action such as reassigning tasks.
- The time needed to complete a particular software development activity is known only roughly. The time needed depends on technical factors such as the complexity of the piece of code to develop, but also on human factors such as the skill and experience of the developers. The human factors are hard to measure.
- It is typical for software projects that activities which run in parallel interfere with each other. For example, when an interface between some of the components in a software system gets extended, all components which use that interface and which are under development must be reworked. Because of the unanticipated rework, the completion time for the components is delayed. It is extremely hard to foresee at what times during a project this sort of feedback between development activities will occur and how much impact on the progress of the project it will have.

When assigning tasks to the developers, the manager must also keep in mind that certain staff might be available only during certain periods of time. In addition,

[*] supported by a post-doctoral fellowship of the Deutsche Forschungsgemeinschaft DFG at the Graduiertenkolleg Informatik, Karlsruhe

there are various precedence relations among the tasks of a project. For example, a module must be designed before it can be coded.

Software engineering currently offers little help to software project managers how to find good schedules for their projects. On the one hand, effort estimation does not support scheduling. We get an estimate for the total development effort required for a project, expressed in man-days, and an estimate for the project duration. Some models also provide a distribution over time of the manpower needed for a project. Both the curve-fitting models and the more recent models, which use machine learning [17], neural networks [19], and analogy [16], do not show individual tasks and developers. Thus, deriving a schedule is not possible. For an overview of current effort estimation models see [7].

On the other hand, the automated software engineering environments which have emerged during the last decade do not support finding good schedules, too. Such an environment guides and supports project managers and developers during real software projects. Each software engineering environment comes with a process modelling language (often more than one) to formally describe the software development process in detail. The description captures the activities to be carried out, the staff involved, the products to be developed, the tools available, and the relationships between all those. Although it is possible for a project manager to assign tasks to developers, software engineering environments do not assist the manager in making that assignment best possible in order to meet a given deadline and budget. The manager also can specify a duration (and cost) for each activity, but the impact of feedback in the software process on the duration of activities is not modelled. An exception is [14]. One would like to see the assignment of tasks to the developers and the duration of tasks changed by the software engineering environment automatically in the best possible way whenever the state of the project has changed. For an overview of software engineering environments see [4][6].

What do we need in order to address the scheduling problem for software projects? First, we need a model for software projects which quantifies the impact of scheduling decisions on the development time and cost of a project. Therefore, scheduling actions such as stopping a task or reassigning a task must be part of the model. Feedback in the software process and its impact on activity durations must be modelled. The uncertainty inherent to the software process concerning the duration of activities and the occurence of events must be modelled. Thus, it is natural for the model to be *probabilistic.* In the model, events will occur only with a certain probability at particular points in time. Scheduling constraints such as precedence relations between tasks must be included in the model, too.

Second, we need techniques to compute optimal scheduling strategies for software projects. A scheduling strategy specifies which scheduling action to take in view of the current project situation. For example, a scheduling strategy might allocate most of the staff to those tasks which seem to be closest to completion, or, which have been worked on for the longest time. An optimal strategy minimizes the project duration (or cost). Since the underlying project model will be probabilistic, an optimal strategy will be *stochastically* optimal, minimizing the *expected* duration or cost. The optimization techniques must be computationally efficient; a full search for an optimal strategy in the huge set of all scheduling strategies is not feasible.

To accomplish the first step, this paper presents a new model for software projects which explicitly takes a scheduling strategy as input. No process modelling language is used, just standard mathematical notations. The model is probabilistic. If the scheduling strategy is fixed, the model outputs an estimate for the completion time. The model describes the software process at a high level of abstraction, talking about teams working on software components, not about individual developers performing activities such as coding a module. Still the model captures much of the dynamics of software projects. The intention is to keep the model as lean as possible for the time being.

The new software project model substantially extends a previous model described in [11]. The previous model made some simplifying assumptions with respect to scheduling. It was assumed that there are as many teams as there are components in the software, that the teams all start working at the same time, and that the teams keep working on their component until it is finished. Instead of scheduling, the previous model focused on the feedback which is so typical for software projects : changes in the software's design lead to rework. The new model is presented in section 2 of the paper.

The new model defines a *Markov decision process.* This setting points out how to accomplish the second step : we can apply stochastic optimization techniques from operations research. These techniques are collectively referred to as *stochastic dynamic programming* [1] [3] [15] . An outline of the basic techniques is given in section 3 of the paper. Although operations research provides optimization techniques that we can apply, the particular stochastic models studied there are not appropriate to describe the software process. Closest to what we need are stochastic project networks [8] [9] . A stochastic project network can model parallel execution of activities and repeated execution of activities. Yet the duration of an activity must not depend on any other activity which runs at the same time, nor on the duration of an activity which was performed earlier. In other words, in a stochastic project network different threads of execution are stochastically independent, as are different activities belonging to the same thread. These assumptions do not hold for software projects.

How can we exploit simulation of the software process for computing optimal schedules? The state space of the new software project model and hence the search space for the optimization procedures will be huge. The situation is similar to problems studied in machine learning. In machine learning, repeated simulation of the process is exploited successfully to cut down the computing time of the stochastic optimization techniques. The techniques used in machine learning are referred to as *reinforcement learning* or *neuro-dynamic programming* [1] [2] . The simulation-based solutions are known to converge to the optimal strategy. Therefore, how much accuracy is lost when using simulation depends on how much computing time is invested. An outline how to use simulation for the optimization of software project schedules is given in section 3 of the paper.

2 The Model

2.1 Software Projects

A software project consists of several development *teams* and a project manager. Based on some early high-level design, the software product is divided into *components.* At any time during the project, each team works on at most one component, and, vice versa, each component is being worked on by at most one team. It is not assumed that all teams work all the time, nor that there are enough teams to work simultaneously on all the uncompleted components. The assignment of components to the teams may change during the project. Thus, a team usually will work on several different components during the project. It is not required that a team has completed its component before it is assigned some other component to work on ; a team may be *interrupted* and re-allocated to another component by the manager.

Usually, several teams work at the same time, each on a different component. The teams do not work independently. From time to time a team might detect a *problem* with the software's high-level design. Since the components are coupled, for example, through common interfaces, such a problem is likely to affect other components and teams as well. To eliminate the problem, the high-level design gets revised. If there are additional problem reports by other teams while the design is being revised, they are taken into account, too. When the *redesign* is completed, some of the components will have to be *reworked* because of the design changes, while others are not affected. To sum up, the progress that a team makes developing its component depends on the progress of the other teams.

When all components have been completed, they are put together and the software gets integration tested. If errors are detected, a new development cycle begins. The model describes a development cycle probabilistically.

2.2 Time

Time is discrete in the model. The time axis gets subdivided into periods of equal length, called *time* *slices.* Think of a time slice as corresponding to, say, one week in real time. In addition, there is a *deadline* for completing the project. If the deadline is exceeded, the project will be cancelled as a failure.

2.3 Phases

In the model, a project advances through *phases.* Each phase lasts for some number of time slices which may vary from phase to phase. By definition, a phase ends ...

- when staff becomes available, or,
- when the software's high-level design changes.

Staff becomes available when some team completes its component. Staff also becomes available when some team completes all rework on a component which already had been completed earlier in the project but had to be reworked because of changes to the software's design. Changes to the software's design might be necessary to fix design errors or because of changes in the requirements.

Scheduling actions take place only at the end of a phase. Scheduling at arbitrary points in (discrete) time is not modelled. The rationale behind this restriction is that is does not make sense to re-schedule a project as long as nothing unusual happens. At the end of a phase though, staff is available again for allocation, or re-scheduling the project might be appropriate because of some design changes. At that time, the manager may also interrupt some of the teams and re-allocate them to other components. For example, the manager might decide to re-schedule a team to rework a central component which had been completed earlier but must be changed according to the revised design.

2.4 States

The *state* of a project changes at the end of each phase. The state ζ of a project by definition consists of four parts:

- a progress vector ζ^{p};
- a rework vector ζ^{r};
- an assignment vector ζ^{a};
- a countdown ζ^{c}.

The *progress vector* has one entry for each component. The progress ζ^{p}_k of component k is defined as the *net* development time that has been spent working on the component. The net development time is obtained from the total development time by substracting all rework times spent with adapting the component to high-level design changes. As a special case, the progress entry for a component is set to ∞ to indicate that the component is completed.

The progress made developing a component must be measurable in practice. A metric such as "x percent completed" would be hard to measure and thus is not suitable. Development times *can* be measured though.

The *rework vector* has one entry for each component, too. Rework is caused by changes to the software's high-level design. The rework time ζ^{r}_k for component k is the time that yet must be spent with adapting the component to high-level design changes. As soon as a component's rework time has been counted down to zero, "normal" development of the component can proceed. If the software's design is changed again while a component is being reworked, leading to even more rework for that component, the extra rework is added to the component's rework time. That is, the impact of high-level design changes on a component is assumed to add up. Once a component has been completed, only rework may occur for the component in the sequel.

The *assignment vector* also has one entry for each component. Entry ζ^{a}_k is the number of the team which has worked on component k most recently. As a special case, the entry equals 0 if none of the teams has worked on the component yet. Each entry in the assignment vector is given a leading plus or minus sign to indicate whether the specified team has been working on the component during the last phase or not.

If the work on a component is not yet completed, which can be seen from the progress vector and the rework vector, a leading minus sign in the assignment vector means that the specified team has been interrupted by the manager in an earlier phase while working on the component. A leading plus sign means that the last phase has ended while the team was still working on the component. In most cases, it will make sense for the manager to have the specified team continue working on the component during the next or some later phase. Any other team might need considerable time to become familiar with the component. For the same reason, it does not make much sense to record the numbers of *all* teams that have worked on the component at some time during the project, because the component will probably have changed considerably in the meantime.

The *countdown* is the time left until the project's deadline of, say, x_0 slices will be reached. The development cycle begins with the *initial state* σ which is defined by $\sigma_k^{\mathrm{p}} = \sigma_k^{\mathrm{r}} = \sigma_k^{\mathrm{a}} = 0$ for all k, and $\sigma^{\mathrm{c}} = x_0$. The development cycle ends when a *termination state* τ is reached. In a termination state, the deadline has not been exceeded ($\tau^{\mathrm{c}} \geq 0$), all components are completed ($\tau_k^{\mathrm{p}} = \infty$), and there is no rework left ($\tau_k^{\mathrm{r}} = 0$). The development cycle also ends when the deadline is passed.

2.5 Actions and Strategies

Scheduling takes place at the end of the phases. Possible scheduling actions are:

- assigning a component to a team;
- starting a team;
- stopping a team.

A scheduling action is modelled as an *action vector* which has one entry for each team. The action a_i for team i is the number of the component the team is scheduled to work on during the next phase. The entry is set to -1 if the team is stopped.

Actions may depend on the current state of the project and the number of the phase[†]. In most cases, several actions are possible for a given state and phase. A *scheduling strategy* or *policy* is a function which (deterministically) specifies a scheduling action for each project state and phase. A strategy is called *stationary* if the choice of the action depends on the state only, whence the strategy is a function mapping states into actions.

[†] There is some confusion in the literature here. Dependence of the action on the number of the phase is needed when studying *finite horizon* optimization. In that case, optimal actions may depend on the number of the phase. Optimal actions for *infinite* horizon problems do *not* depend on the number of the phase. On the other hand, is does not make much sense to have an action depend on the *entire history* of the project because the transition probabilities are always defined to depend on the current state alone once the action is specified, not on the entire history.

2.6 Constraints

A scheduling action is *admissible* only if it satisfies the *precedence relations* between the components. At the current level of abstraction, the model considers precedence relations between whole components only, not between single development activities such as designing and coding a module. The precedence relations resemble the task net of other models for the software process. The relations are specified as a graph or a table, which serves as input to the scheduling strategies. It is assumed that the relations contain no cycles. The precedence relations can force some re-scheduling at the end of a phase if a component has to be reworked which must precede another component that is currently under development.

An action must satisfy additional constraints to be admissible. Each team must work on a different component. An action must schedule at least one team to work. If a component has been completed during an earlier phase and there is no rework for that component, no team may be scheduled to work on the component. The set of all actions which are admissible if the project is in state ζ is denoted by $\mathrm{A}(\zeta)$.

Currently, the model assumes that all teams are available during the whole project. Any team may be scheduled to work on any component, but the teams need not be equally well-prepared for that. Different skill levels of the teams are modelled using the base probabilities of the teams, see subsection 2.8.

2.7 Transition Probabilities

Given a project state ζ and a scheduling action a, the next state η of the project is not completely determined since the different events which will end the next phase will occur only with a certain probability. Define the *transition probability*

$$\mathrm{P}(\zeta, a;\ \eta)$$

to be the probability for ending the next phase with state η given that the previous phase had ended with state ζ and scheduling action a was taken.

The transition probability $\mathrm{P}(\zeta, a; \eta)$ does not depend on any information about the project's history except its current state and the action chosen. For such a modelling to make sense the state must contain all relevant information about the project's past. The resulting process

$$\zeta(0), \quad a(0), \quad \zeta(1), \quad a(1), \quad \ldots$$

is called a *Markov decision process* [1][3][15]. If the scheduling strategy is fixed, the process

$$\zeta(0), \quad \zeta(1), \quad \ldots$$

is a Markov process. Since the actions may depend on the number of the phase, that Markov process is stationary only if the scheduling strategy is.

To compute the transition probabilities, the following input data are required:

- statistical data about past projects;
- high-level design data.

The statistical data are probability distributions specifying for each team how likely it is that the team will complete its component, report a high-level design problem, or adapt its component to a design change within a given time, see subsection 2.8. The design data are a measure for the strength of the coupling between the software's components, see subsection 2.10.

2.8 Statistical Data

The statistical data required as input to the model are a measure of the pace at which the teams have made progress in past projects. Define the *base probabilities*

$$\mathrm{P}(\mathrm{E}_k^i(t)) \quad \text{and} \quad \mathrm{P}(\mathrm{D}_k^i(t))$$

to be the probabilities that team i will report a problem (event $\mathrm{E}_k^i(t)$) or will complete its component (event $\mathrm{D}_k^i(t)$) after a *net* development time of t slices when working on component k. For later use, set

$$\mathrm{B}_k^i(t) \;=\; \overline{\bigcup_{s=1}^{t} \Big(\mathrm{E}_k^i(s) \,\cup\, \mathrm{D}_k^i(s) \Big)} \,.$$

Event $\mathrm{B}_k^i(t)$ corresponds to a situation where team i does not complete component k nor report a problem with the design for a net period of t slices.

As another statistical input to the model, define the *probability of rework time*

$$\mathrm{P}(\mathrm{R}_k(t))$$

to be the probability that it will take t slices to adapt component k to the latest design changes.

The base probabilities depend upon various human and technical factors, for example, the software process employed by the team, the complexity of the component, and the skills and the experience of the team. The base probabilities are computed from empirical data collected during past projects. If the database is sufficiently large, a manager will distinguish between different team productivity levels and component complexity classes when computing the base probabilities. For a particular team and component, a manager will ...

- look at all components developed by the team in past projects;
- classify the components according to their complexity, using a complexity measure of his choice;
- in each complexity class, look at the records to find out the development times and rework times;
- in each complexity class, compute the net times and the probability distributions;
- choose the probability distributions which fit best to the given component.

Note that the database will automatically adjust to the local environment in a company.

An example how to compute the base probabilities and the probabilities of rework time from empirical data is given in [13].

2.9 Advance of a Team

Suppose that team i is scheduled to work on component k during the next phase ($a_i = k$). To compute the transition probabilities, we must know how likely it is that the next phase will end after d slices because team i finishes its component or detects a problem with the high-level design. The corresponding probabilities can be computed from the team's base probabilities. There are two cases to consider:

- Component k has not been completed during an earlier phase ($\zeta_k^{\mathrm{p}} < \infty$).
- Component k has been completed during an earlier phase ($\zeta_k^{\mathrm{p}} = \infty$), but there is some rework for the component to be worked off ($\zeta_k^{\mathrm{r}} > 0$).

In the first case, the probability that team i will complete component k after d more slices is equal to the conditional probability

$$\mathrm{P}\,(\mathrm{D}_k^i(\zeta_k^{\mathrm{p}} + d - \zeta_k^{\mathrm{r}}) \mid \mathrm{B}_k^i(\zeta_k^{\mathrm{p}}))\,.$$

Conditioning by the event $\mathrm{B}_k^i(\zeta_k^{\mathrm{p}})$ takes into account that some teams already have been working on the component for a net time of ζ_k^{p} slices. Similarly, the probability that there will be a redesign of the software after d more slices because team i detects a problem is equal to the conditional probability

$$\mathrm{P}\,(\mathrm{E}_k^i(\zeta_k^{\mathrm{p}} + d - \zeta_k^{\mathrm{r}}) \mid \mathrm{B}_k^i(\zeta_k^{\mathrm{p}}))\,.$$

The rework time ζ_k^{r} must be substracted in both formulas because the first ζ_k^{r} slices during the phase will be spent by team i reworking the component, which yields no net progress. By assumption, teams will not report problems while they are doing rework.

In the second case, team i will complete the rework after ζ_k^{r} slices, no earlier. Therefore, the probability that the team will cause the phase to end after d slices is equal to one if $d = \zeta_k^{\mathrm{r}}$ and zero otherwise.

We also must know how likely it is that team i will *not* cause the next phase to end for a period of d slices when working on component k. In the first case, the probability that team i will not complete the component nor cause a redesign for a period of d slices is equal to the conditional probability

$$\mathrm{P}\,(\mathrm{B}_k^i(\zeta_k^{\mathrm{p}} + d - \zeta_k^{\mathrm{r}}) \mid \mathrm{B}_k^i(\zeta_k^{\mathrm{p}}))\,.$$

In the second case, the probability that team i will not complete the rework for a period of d slices is equal to one if $d < \zeta_k^{\mathrm{r}}$ and zero otherwise.

2.10 Design Data

The design data required as input to the model are a measure for the strength of the coupling between the software's components. The stronger the coupling is the more likely it is that problems originating in one component will lead to rework in other components. For example, when an interface offered by some component is extended, all components which use that interface must be reworked. Often there is more than one link between two components in a design.

For nonempty subsets K and X of the set of components, the *dependency degree*

$$\alpha\left(K,\, X\right)$$

by definition is the probability that changes in the software's design will extend over exactly the components X given that the problems causing the redesign were detected in the components K. At least one component must be changed if design problems occur. Thus, X must be nonempty. For example,

$$\alpha\left(\{3\},\, \{1,\, 2,\, 3\}\right)$$

is the probability that a problem detected in the third component will lead to changes in the first three components of the software.

The dependency degrees are computed from the high-level design of the software. This way, the model *explicitly* takes the design of the software as input, which allows to *quantify* the impact of design decisions on the delivery date and cost of a project, see [12].

2.11 Impact of Design Changes

Suppose that a problem with the software's high-level design occurs during a phase. The problem leads to a revision of the design, whence some of the components must be changed. To compute the transition probabilities, we must know how likely it is that the revision will have a certain impact on the components.

If one or more design problems occur during a phase in the set of components K, the probability that there will be changes to the components in the set X is given by the dependency degree $\alpha\left(K,\, X\right)$. For each component k in X, the probability that the amount of rework to be added to the component's rework time will be t slices is given by the probability distribution $\mathrm{R}_k(t)$ of rework times. Therefore, the probability that a design revision will have a certain impact on the progress of the project is computed by multiplying the corresponding dependency degree and probabilities of rework times.

2.12 Transition Probabilities (Continued)

Suppose that a state ζ and an admissible action $a \in \mathrm{A}(\zeta)$ are given. The next state then is partially determined. For example, if a team is scheduled by the action to work on a particular component, the entry for the component in the next state's assignment vector must be set accordingly. Many combinations of ζ and a with a state η as the next state therefore will be inconsistent. As a result, many transition probabilities will be equal to zero and need not be considered in computations.

To compute the transition probability $\mathrm{P}(\zeta, a; \eta)$ for some state η take the following steps:

- compute the length d of the phase which passes between ζ and η
- check whether the action a and the two assignment vectors ζ^{a} and η^{a} are consistent
- check whether the progress vector η^{p} and the rework vector η^{r} are valid
- compute the set X of components which must be changed as part of the latest redesign, and the amount of additional rework for these components
- determine the set K of components where the redesign comes from
- multiply the right base probabilities, dependency degrees, and probabilities of rework time.

The length d of the phase which passes between states ζ and η is equal to the difference $\zeta^{\mathrm{c}} - \eta^{\mathrm{c}}$ of the countdowns. The length of the phase must be greater than zero, otherwise the transition probability is set to zero.

Since the action a is admissible, it is consistent with the assignment vector ζ^{a}. The new assignment vector η^{a} is completely determined by the action a and the old assignment vector ζ^{a}. If team i is scheduled to work on component k by the action, the entry for component k in the next state's assignment vector must be $\eta_k^{\mathrm{a}} = i$. If no team is scheduled to work on component k, the entry must be $\eta_k^{\mathrm{a}} = \zeta_k^{\mathrm{a}}$. If the new assignment vector η^{a} is any different from that, the transition probability is set to zero.

Several checks must be performed on the progress vector and the rework vector of state η to see whether η is a valid next state for ζ when action a is taken. For example, only those components k which are worked on during the phase can have net progress or their rework time counted down. There are several basic cases to consider:

- there already is some net progress for component k ($\zeta_k^{\mathrm{p}} > 0$) or not
- some team works on component k during the phase ($\eta_k^{\mathrm{a}} > 0$) or not
- component k had already been completed in some earlier phase ($\zeta_k^{\mathrm{p}} = \infty$) or not
- there is enough time to achieve net progress for component k during the phase ($d > \zeta_k^{\mathrm{r}}$) or not.

The following table shows which values of η_k^{p} and η_k^{r} in the next state's progress vector and rework vector are valid. Just disregard the t_k column for the moment.

case	η_k^{p}	η_k^{r}	t_k
$\eta_k^{\mathrm{a}} \le 0$ $\zeta_k^{\mathrm{p}} = 0$	0	0	0
$\eta_k^{\mathrm{a}} \le 0$ $\zeta_k^{\mathrm{p}} > 0$	ζ_k^{p}	$\ge \zeta_k^{\mathrm{r}}$	$\eta_k^{\mathrm{r}} - \zeta_k^{\mathrm{r}}$
$\eta_k^{\mathrm{a}} > 0$ $\zeta_k^{\mathrm{p}} = \infty$	∞	$\ge \zeta_k^{\mathrm{r}} - d$	$\eta_k^{\mathrm{r}} - (\zeta_k^{\mathrm{r}} - d)$
$\eta_k^{\mathrm{a}} > 0$ $\zeta_k^{\mathrm{p}} < \infty$ $d > \zeta_k^{\mathrm{r}}$	∞ or $\zeta_k^{\mathrm{p}} + (d - \zeta_k^{\mathrm{r}})$	≥ 0	η_k^{r}
$\eta_k^{\mathrm{a}} > 0$ $\zeta_k^{\mathrm{p}} < \infty$ $d \le \zeta_k^{\mathrm{r}}$	ζ_k^{p}	$\ge \zeta_k^{\mathrm{r}} - d$	$\eta_k^{\mathrm{r}} - (\zeta_k^{\mathrm{r}} - d)$

If the new state η deviates from any of the values or bounds specified in the table, the transition probability is set to zero.

The table also shows by how much additional rework t_k component k will be set back because of the latest design changes. The set X of components which must be changed is

$$X = \{ k \mid t_k > 0 \}.$$

There might be additional rework even for components on which no team works during the phase. The set X may be empty. If X is nonempty, at least one team reports a design problem during the phase.

The set M of all teams which are scheduled to work during the phase is computed from the action a as

$$M = \{ i \mid a_i > 0 \}.$$

The set M is a disjoint union of several subsets M_1 through M_4. The subset M_1 consists of all teams which just complete their component at the end of the phase and is computed as

$$M_1 = \{ i \mid \eta^{\mathrm{p}}_{a_i} = \infty \text{ but } \zeta^{\mathrm{p}}_{a_i} < \infty \}.$$

The subset M_4 consists of all teams which just complete all rework on a component that had already been completed in an earlier phase and is computed as

$$M_4 = \{ i \mid \zeta^{\mathrm{p}}_{a_i} = \infty \text{ and } d = \zeta^{\mathrm{r}}_{a_i} \}.$$

The subset M_2 consists of all teams which report a design problem during the phase. The set M_3 consists of all teams which are still working at the end of the phase.

In most cases, the sets M_2 and M_3 are *not* determined by the action a and the states ζ and η. For example, think of two scenarios where a different team reports a design problem in each scenario. Both problem reports may lead to design changes in the same set of components and to the same amount of rework for each of the changed components. Therefore, if not already set to zero because of inconsistencies, the transition probability $\mathrm{P}(\zeta, a; \eta)$ is a *sum*

$$\mathrm{P}(\zeta, a; \eta) = \sum_{Z} \mathrm{P}(Z)$$

where the sum runs over all partitions Z of the set $M \setminus (M_1 \cup M_4)$ into two sets M_2 and M_3. For each partition Z, the probability $\mathrm{P}(Z)$ which contributes to the transition probability is computed as a product of several factors:

- For each team in M, one factor comes from the base probabilities, as described in subsection 2.9.
- One factor comes from the dependency degree for K and X, as described in subsection 2.11. Here,

$$K = \{ a_i \mid i \in M_2 \}.$$

- For each component k in X, one factor comes from the distribution $\mathrm{R}_k(t)$ of rework time for that component, as described in subsection 2.11. The value of t_k from the table given above is used as the parameter t.

2.13 Proof

The transition probabilities $\mathrm{P}(\zeta, a; \eta)$ yield a Markov decision process only if for each state ζ and each action a the probabilities for all possible transitions to some other state η sum up to one. That is,

$$\sum_{\eta} \mathrm{P}(\zeta, a; \eta) = 1.$$

The proof of this equation runs along the same lines as the proof given in [10] for the previous model [11].

3 Optimization Using Simulation

3.1 Cost Functions

Associate with the transition from a state ζ to some state η the *transition cost*

$$g(\zeta, a; \eta).$$

The cost of a transition depends on the scheduling action a taken. For example, in the software project model the transition cost may be the length $d = \zeta^{\mathrm{c}} - \eta^{\mathrm{c}}$ of the phase which passes between ζ and η. The transition cost also may be the staffing cost for the phase, which depends on the length of the phase, the set of teams scheduled to work during the phase, the cost per week for teams which work, and the cost per week for teams which wait to be scheduled.

Each state ζ is also assigned a *terminal cost*

$$g(\zeta)$$

which is incurred when ending the process in state ζ. For example, in the software project model the terminal cost of a state which corresponds to the project being cancelled as a failure might be some financial penalty. Termination states, which correspond to a successful outcome of the project, have zero terminal cost.

Using the transition costs, one can assign to any finite sequence

$$\omega \;=\; \zeta(0),\ a(0)\ \ldots \\ \ldots\ \zeta(m-1),\ a(m-1),\ \zeta(m)$$

of state-action pairs its cost

$$g(\omega) \;=\; \sum_{i=0}^{m-1} g(\zeta(i),\ a(i);\ \zeta(i+1))$$

by summing up the costs of all the transitions in the sequence[†]. The sequence ω can be viewed as the *path* or the *course* of the project when observed for a period of m phases from state $\zeta(0)$ on. For the software project model, the first state $\zeta(0)$ in a sequence ω need not be equal to the initial project state σ.

Given a state ζ and an action a, the next state of the process will be η only with a certain probability. The *expected cost* for the next transition is

$$\sum_{\eta} \mathrm{P}(\zeta, a;\ \eta) \cdot g(\zeta, a;\ \eta).$$

The probability that the process will proceed from state ζ according to a sequence ω is equal to the product

$$\mathrm{P}(\omega) \;=\; \prod_{i=0}^{m-1} \mathrm{P}(\zeta(i),\ a(i);\ \zeta(i+1))$$

of the corresponding transition probabilities. Thus, given a strategy π the *expected n-stage cost-to-go* for state ζ is computed as

$$G_n^{\pi}(\zeta) \;=\; \sum_{\Omega_n^{\pi}(\zeta)} \mathrm{P}(\omega) \cdot \big(g(\omega) + g(\zeta(n))\big).$$

[†] We are a bit sloppy with the notation here, using g with transitions, states, and state-action sequences as parameters.

The set $\Omega_n^{\pi}(\zeta)$ consists of all sequences ω which start with state ζ, have n stages, and are controlled by the strategy π, that is, for which $a(i) = \pi(i, \zeta(i))$. The functions G_n^{π} are called the *cost-to-go functions* of the strategy π.

For the software project model, a stage is the same as a phase. Since a project must succeed before the deadline of x_0 time slices is exceeded, a project will last for at most x_0 phases. The *expected project cost* when scheduling according to π then is

$$\mathrm{E}_{cost} \;=\; G_{x_0}^{\pi}(\sigma).$$

It is understood here that a sequence which terminates successfully before the deadline is exceeded has zero transition costs afterwards.

3.2 Optimal Strategies

Optimizing the schedule of software projects with respect to development time or cost amounts to solving the following *stochastic optimization problem:*

> Find a scheduling strategy which has minimal cost-to-go functions in the Markov decision model for software projects.

A strategy π has minimal cost-to-go functions if

$$G_n^{\pi}(\zeta) \;\le\; G_n^{\mu}(\zeta)$$

for all strategies μ, stages n, and states ζ. An optimal strategy will be *stochastically* optimal, minimizing the *expected* cost. The cost function g for the software project model is either the development time function or the staffing cost function described at the beginning of subsection 3.1.

The search space for the optimization problem consists of all possible scheduling strategies. The search space is far too huge to perform a full search. The key to finding an optimal strategy is the observation that an optimal action for state ζ with n stages to go must minimize the sum of

- the expected cost for the next transition and
- the expected *optimal* cost with $n-1$ stages to go.

Denote by $G^{\star}_{n}(\zeta)$ the *optimal expected cost* for state ζ with n stages to go. Formally, the observation says:

$$G^{\star}_{n}(\zeta) = \min_{a \in \mathrm{A}(\zeta)} \sum_{\eta} \mathrm{P}(\zeta, a; \eta) \cdot \big(g(\zeta, a; \eta) + G^{\star}_{n-1}(\eta) \big).$$

This is *Bellman's equation* of stochastic dynamic programming [1] [3] [15]. The proof of Bellman's equation relies on the Markov property of the transition probabilities for the underlying stochastic process.

Once the optimal cost-to-go functions have been computed, an optimal strategy is obtained by choosing the actions in such a way that the minimum in Bellman's equation is attained for all stages n and states ζ. The optimal cost-to-go functions are unique, of course, but there might be more than one optimal strategy.

Bellman's equation gives an iterative algorithm to compute the optimal cost and an optimal strategy for a Markov decision process. The algorithm is known as *backwards dynamic programming*. Start with the terminal costs of the states as the optimal zero-stage costs,

$$G^{\star}_{0}(\zeta) = g(\zeta).$$

Then, compute the optimal one-stage costs from Bellman's equation for all states ζ. Then, compute the optimal two-stage costs, and so on. For the software project model, the terminal cost of a state which corresponds to the project being cancelled as a failure should be set to some high value to make that state look bad to the optimization algorithm as a last state.

3.3 Policy Iteration

Computing the optimal expected cost and an optimal strategy using backwards dynamic programming is increasingly expensive as the number of states grows. For the software project model, the number of states will be huge, growing exponentially with the number of components.

Based on Bellman's equation, another algorithm for computing an optimal strategy has been developed, called *policy iteration,* which computationally is more efficient [1] [3] [15]. Policy iteration generates a sequence

$$\pi_1, \pi_2, \ldots$$

of policies and terminates after finitely many iterations with an optimal policy. The sequence of policies generated is *improving* in the sense that the cost-to-go functions improve with each iteration:

$$G^{\pi_{k+1}}_{n}(\zeta) \leq G^{\pi_k}_{n}(\zeta)$$

for all stages n to go, states ζ, and iterations k. The policy iteration algorithm alternates between a *policy evaluation* step and a *policy improvement* step.

- Policy Evaluation Step. Evaluate policy π_k by computing all its cost-to-go functions $G^{\pi_k}_{n}$.
- Policy Improvement Step. Obtain the next policy π_{k+1} by performing the minimization of Bellman's equation, but using the cost-to-go functions $G^{\pi_k}_{n}$ of the last policy π_k instead of the yet to be determined optimal cost-to-go functions $G^{\star}_{n}$.

Formally, the improvement step determines the actions of the next policy π_{k+1} in such a way that the equation

$$G^{\pi_{k+1}}_{n}(\zeta) = \min_{a \in \mathrm{A}(\zeta)} \sum_{\eta} \mathrm{P}(\zeta, a; \eta) \cdot \big(g(\zeta, a; \eta) + G^{\pi_k}_{n-1}(\eta) \big)$$

holds for all n and ζ. The equation means that π_{k+1} chooses the action with n stages to go best possible when assuming that the following actions will be chosen according to π_k. The algorithm stops if the new policy does not improve the last one for at least one state, whence both policies are optimal. The algorithm gets initialized by choosing some policy π_0. The closer the cost of the initial policy π_0 is to the optimum, the fewer iterations are necessary before the algorithm terminates with an optimal strategy.

3.4 Simulation

To evaluate a given strategy in the evaluation step of the policy iteration algorithm, all of the strategy's cost-to-go functions must be computed. This computation is expensive if the number of states is large. In machine learning, where the number of states usually is large, *simulation* has been successfully employed to *approximate* the cost-to-go functions G^{π}_{n} of a given strategy π, see [2]. The idea is ...

- to generate *sample trajectories* of the process, starting from state ζ and scheduling according to π;
- to compute the cost for each sample trajectory;
- to take the mean of the sample costs as an approximation of the strategy's cost-to-go for state ζ.

The sample mean is

$$\widetilde{G}_n^{\pi}(\zeta) = \frac{1}{S} \cdot \sum_{m=1}^{S} s_n(\zeta, m)$$

where $s_n(\zeta, m)$ denotes the cost of sample number m for state ζ with n stages to go, and S denotes the number of such samples obtained from the simulation. The sample mean is taken as an approximation for the cost-to-go:

$$\widetilde{G}_n^{\pi}(\zeta) \approx G_n^{\pi}(\zeta).$$

The sample mean is known to converge to the cost-to-go function with probability one [2]. Therefore, the accuracy of the approximation depends only on the number of simulation runs.

It is possible to use a given trajectory to obtain a cost sample for *each* state visited by the trajectory. For each intermediate state $\zeta(k)$ of the trajectory, consider the segment $\zeta(k),\ \zeta(k+1),\ \ldots$ to get cost samples for state $\zeta(k)$ with 1 up to $n = N - k$ stages to go, where a simulation trajectory is stopped after N stages. Doing so is feasible because of the Markov property of the process. Once a trajectory gets to some state, the statistics of the future only depend on that state, not on the sequence of states visited before. Therefore, during a simulation an *update formula* [2] can be used if state ζ occurs in a trajectory at stage n for the m-th time:

$$\widetilde{G}_n^{\pi}(\zeta) := \widetilde{G}_n^{\pi}(\zeta) + \frac{1}{m} \cdot \left(s_n(\zeta, m) - \widetilde{G}_n^{\pi}(\zeta)\right).$$

The approximate cost-to-go $\widetilde{G}_n^{\pi}(\zeta)$ is initialized with zero. The update formula is applied to *each* state visited by a simulation trajectory.

A given strategy may tend to stay in a *subset* of all possible states. In that case, states from the subset will be more likely to occur during simulation than others. Thus, the cost-to-go estimates for states in the subset will be more accurate than for states outside. This is a problem when using simulation for the evaluation step in the policy iteration algorithm. When using simulation, the improvement step is

$$G_n^{\pi_{k+1}}(\zeta) = \min_{a \in \mathrm{A}(\zeta)} \sum_{\eta} \mathrm{P}(\zeta, a;\ \eta) \cdot \left(g(\zeta, a;\ \eta) + \widetilde{G}_{n-1}^{\pi_k}(\eta)\right).$$

One can see that cost-to-go estimates are needed for all states η with $\mathrm{P}(\zeta, a;\ \eta) \neq 0$. Suppose that some state η is needed during the improvement step and that the simulation for the last strategy π_k did not reach state η enough times to obtain a reasonable cost-to-go estimate for the state. One can then postpone the improvement step and go back to simulating the process with π_k as the strategy and η as the first state. This approach is called *iterative resampling* [2]. Iterative resampling has the advantage that states which are not needed for the policy improvement step do not use up simulation time.

For the software project model, a phase is simulated by "throwing a dice" which behaves according to the base probabilities, dependency degrees, and probabilities of rework time. No state can occur twice in a sample trajectory since the countdown is part of the project state. This avoids some technical problems with multiple visits to the same state in a single trajectory. Experience with simulating the previous project model suggests that the simulations of the new model can be both accurate and fast.

4 Research

The next step is to implement a computer program which is based on the new software project model and combines policy iteration with process simulation to compute optimal schedules. Once the program is available, a number of interesting research questions can be tackled, including:

- How far away from the optimum are the strategies that managers use in real projects?
- Should we develop components which are strongly coupled to several other components early or late in the project?

- Does the model reflect the results obtained by other researchers about schedule compression?

References

1. Bertsekas: *Dynamic Programming and Optimal Control I+II.* Athena Scientific 1995
2. Bertsekas, Tsitsiklis: *Neuro-Dynamic Programming.* Athena Scientific 1996
3. Derman: *Finite State Markovian Decision Processes.* Academic Press 1970
4. Derniame, Ali Kaba, Wastell: *Software Process: Principles, Methodology, and Technology.* Lecture Notes in Computer Science 1500, Springer 1999
5. El Emam, Madhavji: *Elements of Software Process Assessment and Improvement.* IEEE Computer Society Press 1999
6. Finkelstein, Kramer, Nuseibeh: *Software Process Modelling and Technology.* Research Studies Press 1994
7. Gray, MacDonell: "A Comparison of Techniques for Developing Predictive Models of Software Metrics", Information and Software Technology 39 (1997) 425-437
8. Neumann: *Stochastic Project Networks.* Lecture Notes in Economics and Mathematical Systems 344, Springer 1990
9. Neumann: "Scheduling of Projects with Stochastic Evolution Structure", see [18] 309-332
10. Padberg: *Schätzung der Erfolgsaussichten, der Dauer und der Kosten von Softwareprojekten mit strengen wahrscheinlichkeitstheoretischen Methoden.* Dissertation (in German), Universität Saarbrücken 1998
11. Padberg: "A Probabilistic Model for Software Projects", Proceedings ESEC/FSE 7 (1999) 109-126, Lecture Notes in Computer Science 1687, Springer 1999
12. Padberg: "Linking Software Design with Business Requirements – Quantitatively", Second International Workshop on Economics-Driven Software Engineering Research, see Proceedings ICSE 22 (2000) 811
13. Padberg: "Estimating the Impact of the Programming Language on the Development Time of a Software Project", submitted
14. Raffo, Kellner: "Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives", see [5] 297-341
15. Ross: *Introduction to Stochastic Dynamic Programming.* Academic Press 1983
16. Shepperd, Schofield, Kitchenham: "Effort Estimation Using Analogy", Proceedings ICSE 18 (1996) 170-178
17. Srinivasan, Fisher: "Machine Learning Approaches to Estimating Software Development Effort", IEEE Transactions on Software Engineering 21-2 (1995) 126-137
18. Weglarz: *Project Scheduling. Recent Models, Algorithms, and Applications.* Kluwer 1999
19. Wittig, Finnie: "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort", Australian Journal of Information Systems 1 (1994) 87-94