Scheduling Software Projects to Minimize the Development Time and Cost with a Given Staff

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

Abstract

A probabilistic scheduling model for software projects is presented. The model explicitly takes a scheduling strategy as input. When the scheduling strategy is fixed, the model outputs a probability distribution for the project completion time or cost. By applying stochastic optimization techniques, schedules for software projects can be computed which minimize the development time or cost with a given staff.

1 Introduction

Staff is the most valuable resource today in software development. 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, 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 models do not support scheduling. They only provide 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 [19], neural networks [22], and analogy [18], do not show individual tasks and developers. Thus, deriving a schedule is not possible. For an overview of effort estimation see [8].

On the other hand, the process-centered 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 devel-

oped, 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 [16]. One would like to see the assignment of tasks to the developers and the duration of tasks automatically changed by the software engineering environment in the best possible way whenever the state of the project has changed. For an overview of software engineering environments see [2][5][7].

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, using the software project model. A scheduling strategy specifies which scheduling action to take in view of the current project situation. 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 generic 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. When the scheduling strategy is fixed, the model outputs a probability distribution for the project completion time and a completion time estimate. The model describes the software process at a high level of abstraction: teams work on software components. The intention is to keep the model as lean as possible for the time being. Classical process phases such as coding or testing are not modelled. By modelling individual components the model does go below the level of system dynamics models [1][9][20]. It captures much of the dynamics of software projects, representing varying staff skill levels, rework caused by design changes, component coupling, and task assignments.

The software project model presented here substantially extends a previous model described in [12]. 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 a team keeps working on its 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 describes feedback the same way, but adds scheduling. An earlier version of the scheduling model has been presented at a workshop [14]. The model is described in section 2 of the paper.

The model defines a *Markov decision process.* This mathematical 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* [3][4][17]. An outline of how to apply these techniques to the software project scheduling model 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 [10][11]. A stochastic project network can model parallel execution of activities and repeated execution of activities; but 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 clearly do not hold for software projects. Thus, a new approach to the scheduling problem for software projects is required.

2 Process 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 problems reported 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.

[†] not to be confused with the development phases in other process models such as the waterfall model

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 for 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, development of the component can be resumed as planned. 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 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 specified team was still working on the component. For example, an entry $\zeta_2^{\,\mathrm{a}} = -5$ means that team number five was the last team to work on the second component and has been re-scheduled to work on some other component in the second-last or some earlier phase. 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* $\zeta^{\,\mathrm{c}}$ 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 cycle ends when a *termination state* is reached. In a termination state τ, 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 termination states differ in the value of the countdown and the values in the assignment vector. 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, but also on the number of phases completed so far[†]. In most cases, several actions are possible for a given state and phase. A *scheduling strategy* or *policy* is a function which (deterministically) specifies an action for each project state and phase. A strategy is called *stationary* if the choice of the action depends on the state only; in that special case, the strategy is a function mapping states into actions.

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 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.

Besides possible precedence relations among components, 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 the project has not yet terminated.
- 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 $\mathrm{A}(\zeta)$.

An example of a simple scheduling strategy is to use a fixed priority list for the components. An available team is allocated to the next unprocessed component in the list. If several teams are available at the same time, the next unprocessed component in the list is assigned to the available team with the highest team number, the second next unprocessed component in the list is assigned to the available team with the second highest team number, and so on. The priority list must satisfy all precedence relations. A team works on its component without interruptions until the component is completed, except when the team has to rework one of its previous components because of a design problem. If a team has to rework several of its previous components, the components are reworked according to their priorities. Immediately after having finished all rework, the team resumes working on the uncompleted component. Note that which team has completed a particular component can be read off the project state's assignment vector.

Currently, the model assumes that all teams can work during the whole project. Any team may be scheduled to work on any component, but the teams

[†] Dependence of the action on the number of phases completed makes sense for "finite horizon" optimization, because in a late phase the action must take into account that time is running out.

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 2.7. The base probabilities may be taken into account when choosing a scheduling action. For example, in the strategy given in the preceding paragraph the available team with the shortest expected net development time for the next unprocessed component in the list could be allocated to that component (instead of the available team with the highest team number). The expected net development time for team i and component k is equal to $\mathrm{E}[\mathrm{P}(\mathrm{D}_k^i(t))]$, see subsection 2.7.

2.6 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, the action chosen, and the number of phases completed so far (since the action depends on this). For such a modelling to make sense the state must contain all relevant information about the project's past. The resulting process

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

is called a *Markov decision process* [3][4][17]. If the scheduling strategy is fixed, the process

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

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

To compute the transition probabilities, statistical data about past projects and high-level design data are required as input, as described in the next two subsections.

2.7 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. 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 are best suited for the given component.

The empirical database reflects the specific development environment and process in a company, since the data are taken directly from a company's software projects. An example how to compute the base probabilities and the probabilities of rework time from empirical data is given in [15].

2.8 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*

$$\alpha(K, X)$$

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 when 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 [13].

2.9 Transition probabilities (continued)

Suppose that a state ζ and an admissible action $a \in \mathrm{A}(\zeta)$ are given. The next state of the project 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 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 the following steps must be taken:

- compute the length d of the phase which passes between ζ and η as the difference $\zeta^{\mathrm{c}} - \eta^{\mathrm{c}}$ of the countdowns;
- 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.

For details on how the base probabilities, probabilities of rework times, and dependency degrees enter the formula for the transition probabilities, see [14] and the previous model [12].

3 Optimization

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

$$\begin{aligned}\omega \quad = \quad & \zeta(0),\ a(0)\ \ldots \\ & \ldots\ \zeta(m-1),\ a(m-1),\ \zeta(m)\end{aligned}$$

of state-action pairs its cost

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

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. Therefore, 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

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

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

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

[†] 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 of state-action pairs 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. A project starts in the initial state σ. 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, number of 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.

The optimal expected cost is achieved when always choosing an optimal action in the remaining stages. Denote by $G_n^\star(\zeta)$ the *optimal expected cost* for state ζ with n stages to go. Formally, the observation says:

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

This is *Bellman's equation* of stochastic dynamic programming [3][4][17]. 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, but there might be more than one strategy achieving the optimal cost.

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,

$$G_0^\star(\zeta) \;=\; g(\zeta).$$

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 [3][4][17]. 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:

$$G_n^{\pi_{r+1}}(\zeta) \;\le\; G_n^{\pi_r}(\zeta)$$

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

- Policy Evaluation Step. Evaluate policy π_r by computing all its cost-to-go functions $G_n^{\pi_r}$.
- Policy Improvement Step. Obtain the next policy π_{r+1} by performing the minimization of Bellman's

equation, but using the cost-to-go functions $G_n^{\pi_r}$ of the last policy π_r instead of the yet to be determined optimal cost-to-go functions $G_n^\star$.

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

$$G_n^{\pi_{r+1}}(\zeta) = \min_{a \in A(\zeta)} \sum_{\eta} P(\zeta, a; \eta) \cdot \left(g(\zeta, a; \eta) + G_{n-1}^{\pi_r}(\eta) \right)$$

holds for all n and ζ. The equation means that π_{r+1} chooses the action with n stages to go best possible when assuming that the following actions will be chosen according to π_r. 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.

4 Conclusions

We are currently implementing the model and the optimization techniques. Using the model and techniques presented in the paper, a number of interesting research questions can be tackled, including:

- How far away from the optimum are the scheduling strategies that managers use in real projects?
- Should we develop components which are strongly coupled to several other components early or late in a project?
- Can we derive good, practical scheduling "rules" from the properties of optimal strategies?

The model's use can be extended beyond schedule optimization. By using the staffing cost for a phase instead of the phase duration as the transition cost (see subsection 3.1), the model can be used to analyse tradeoffs between schedule and effort. This is another topic for future research using the model.

References

1. Abdel-Hamid, Madnick: *Software Project Dynamics.* Prentice Hall 1991
2. Ambriola, Conradi, Fuggetta: "Assessing Process-Centered Software Engineering Environments", ACM Transactions on Software Engineering and Methodology 6:3 (1997) 283-328
3. Bertsekas: *Dynamic Programming and Optimal Control.* Athena Scientific 1995
4. Derman: *Finite State Markovian Decision Processes.* Academic Press 1970
5. Derniame, Ali Kaba, Wastell: *Software Process: Principles, Methodology, and Technology.* Lecture Notes in Computer Science 1500, Springer 1999
6. El Emam, Madhavji: *Elements of Software Process Assessment and Improvement.* IEEE Computer Society Press 1999
7. Finkelstein, Kramer, Nuseibeh: *Software Process Modelling and Technology.* Research Studies Press 1994
8. Gray, MacDonell: "A Comparison of Techniques for Developing Predictive Models of Software Metrics", Information and Software Technology 39 (1997) 425-437
9. Madachy: "System Dynamics Modeling of an Inspection-Based Process", Proceedings ICSE 18 (1996) 376-386
10. Neumann: *Stochastic Project Networks.* Lecture Notes in Economics and Mathematical Systems 344, Springer 1990
11. Neumann: "Scheduling of Projects with Stochastic Evolution Structure", see [21] 309-332
12. Padberg: "A Probabilistic Model for Software Projects", Proceedings ESEC/FSE 7 (1999) 109-126, Lecture Notes in Computer Science 1687, Springer 1999
13. Padberg: "Linking Software Design with Business Requirements – Quantitatively", 2nd International Workshop on Economics-Driven Software Engineering Research EDSER (2000)
14. Padberg: "Towards Optimizing the Schedule of Software Projects with Respect to Development Time and Cost", International Software Process Simulation Modeling Workshop ProSim (2000)
15. Padberg: "Estimating the Impact of the Programming Language on the Development Time of a Software Project", Proceedings International Software Development and Management Conference ISDM/AP-SEPG (2000) 287-298
16. Raffo, Kellner: "Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives", see [6] 297-341
17. Ross: *Introduction to Stochastic Dynamic Programming.* Academic Press 1983
18. Shepperd, Schofield, Kitchenham: "Effort Estimation Using Analogy", Proceedings ICSE 18 (1996) 170-178
19. Srinivasan, Fisher: "Machine Learning Approaches to Estimating Software Development Effort", IEEE Transactions on Software Engineering 21:2 (1995) 126-137
20. Tvedt, Collofello: "Evaluating the Effectiveness of Process Improvements on Software Development Cycle Time via System Dynamics Modeling", Proceedings COMPSAC 19 (1995) 318-325
21. Weglarz: *Project Scheduling. Recent Models, Algorithms, and Applications.* Kluwer 1999
22. Wittig, Finnie: "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort", Australian Journal of Information Systems 1 (1994) 87-94