SOFTWARE PROCESS IMPROVEMENT AND PRACTICE
Softw. Process Improve. Pract. 2006; **11**: 77–91
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spip.254

A Study on Optimal Scheduling for Software Projects

Frank Padberg*,†
Fakultät für Informatik, Universität Karlsruhe, Germany

Research Section

Software projects often suffer from unexpected rework and delays. Therefore, project scheduling remains a difficult task for the managers. In this article, we compute optimal scheduling strategies for a set of sample software projects and simulate their behavior. The computations are based on a stochastic Markov decision model for software projects, which focuses on capturing the feedback between concurrent development activities. Since the underlying process model is stochastic, the strategies are stochastically optimal, that is, they minimize the expected project duration. The ultimate goal of this research is to develop guidelines for managers to schedule their software projects under uncertainty in the best possible way.

The sample projects are similar, but differ in certain characteristics of the project or product, such as the strength of the coupling between the components or the degree of specialization of the teams on the tasks. By using a set of related projects, we can study how the project characteristics influence the optimal scheduling decisions in a project. After computing the optimal scheduling policies for the sample projects, we use extensive discrete-event simulations to study the behavior of the optimal policy for each given setting and compare the performance of the optimal policy against the possible list policies. List policies are a simple and commonly used class of scheduling policies. For our sample projects, the simulations show that the best list policy in general is not optimal. The higher the degree of specialization of the teams, the larger the performance gap is. On the other hand, the stronger the coupling between the components, the smaller is the improvement that the optimal policy achieves over the best list policy. Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: project scheduling; optimal schedules; process simulation; rework modeling; stochastic process models

1. INTRODUCTION

The planning of software projects takes place under considerable uncertainty. For various reasons, the time needed to complete some development activities in a software project is hard to estimate. In addition, feedback between the various activities often causes rework and delays. These circumstances make project scheduling a notoriously difficult task for the managers of software projects.

In this article, we begin a systematic study of how the good scheduling decisions in a software project depend on certain characteristics of the project or the product. Examples of such characteristics are the strength of the coupling between the components or the degree of specialization of the teams on different tasks. More specific research questions

* Correspondence to: Frank Padberg, Fakultät für Informatik, Universität Karlsruhe, Germany
†E-mail: padberg@ira.uka.de

Contract/grant sponsor: Deutsche Forschungsgemeinschaft, DFG

that we address in this study are:

- Can we save cost and time in software projects by applying *dynamic* scheduling strategies instead of the commonly used simple heuristics?
- If yes, which are the *key factors* that are taken into account by the dynamic scheduling strategies to yield this improvement?
- How does the inherent *uncertainty* about the task-durations and project-path influence the scheduling decisions and cost in a software project?
- How much impact does the *coupling* in a software system have on the cost and schedulability of the development project?

The ultimate goal of this research is to develop *practical guidelines* that indicate to managers how to best schedule their software projects under uncertainty.

Under conditions of uncertainty, the future path of a project is not known in advance. Therefore, the best a manager can do is to schedule the project in such a way that cost or duration will *likely* be minimized. To model the uncertainty in the software process we use a *stochastic* scheduling model, which we have presented earlier in detail (Padberg 2001, Padberg 2002a, 2002b). In this stochastic setting, scheduling is optimal if it minimizes the *expected* project cost (or duration).

In our model, the software is developed by teams who work in parallel on the software's components. At any time during the project, each team works on, at the most, one component only. Scheduling is dynamic in the model: The assignment of the components to the teams may change several times during the project, depending on the *scheduling policy* followed by the manager. The net progress of a team when working on a component is described by *probability distributions*. By definition, the net progress for a component excludes any rework performed on the component.

Our process model focuses on modeling the *feedback* between concurrent activities in a software project. From time to time, some team may detect a defect, an inconsistency, or some other 'problem' in the software's high-level design. Since the components are coupled, through common interfaces for example, the necessary redesigning will force a number of components to be *reworked*. As a result, the teams do not work independently; the progress of one task not only depends on the productivity of the team undertaking the task but also on the progress of the other teams on their respective tasks.

To study the relationship between scheduling and a project's characteristics, we evaluate the performance of different scheduling strategies on a set of related sample projects. The projects are hypothetical, but have realistic input data. More specifically, we fix a set of two teams and four components. The teams have different productivities and the components have different complexities and risk-levels. We then systematically vary the strength of the coupling between the four components, the degree of specialization of the two teams on the components, and the scheduling strategy.

We study four different couplings (minimal, uniform, maximal, asymmetric) in combination with 21 different cases of specialization of teams on components. This procedure yields a set of 84 different yet related projects. Specialization of a team on a certain component is expressed in our model by using a probability distribution for the team's net progress, which has a smaller mean and variance as compared to a nonspecialized team – see Subsection 4.4 for details. The coupling strength is expressed in our model by suitable change propagation probabilities. The stronger the coupling between a software's components, the more likely it is that the design changes will propagate through the software, as described in more detail in Subsection 4.5.

For each of the 84 sample projects, we evaluate 25 different scheduling policies: all possible *list policies* and the optimal policy. List policies use a fixed priority list for assigning the components to the teams – see Subsection 3.3 for more details. With four components, there are 24 different list policies for each project. We have already seen in a previous study (Padberg 2002b), that the list policies are a good starting point for analyzing scheduling decisions in our process model.

For the purpose of this study, cost is defined to be the project duration. Hence, an optimal policy minimizes the expected project length. The optimal policy for any given project setting is computed using a particular dynamic programming algorithm called value iteration – see Section 5.

Process simulation is an important technique in this study. To evaluate the performance of the various scheduling policies, we simulate 10,000 project paths (trajectories) for each possible combination of a project-setting with a policy. We observe the project cost (duration) for each trajectory and use the resulting frequency table and mean value as an approximation of the true cost distribution and the true expected cost of the policy. For a given setting, we use the simulation results to determine the best list policy. We also use the simulated mean cost to compare the performance of the best list policy against the optimal policy.

For our sample projects we find that the best list policy, in general, is not optimal. The performance gap between the best list policy and the optimal policy is proportional to the degree of team specialization. A dynamic scheduling policy can take advantage of team specialization better than the relatively inflexible list policies.

On the other hand, a strong coupling between the components in the sample project results in a small difference between the optimal policy and the best list policy. With a strong coupling, the redesigns that occur from time to time during the project are highly likely to cause rework in many components. Hence, the actual completion times for the different components deviate considerably from the values that can be expected from the teams' probability distributions, making it more difficult for dynamic policies to turn their knowledge of these distributions into a cost advantage.

2. RELATED WORK

Scheduling is not currently supported by effort estimation techniques in software engineering. Both the classical curve-fitting models (see (Gray and Mac Donell 1997) for an overview) and the more recent estimation models, which use techniques such as machine learning (Srinivasan and Fisher 1995), neural networks (Wittig and Finnie 1994), and analogy (Shepperd *et al*. 1996), do not show individual tasks and developers. Hence, deriving a detailed schedule with these models is not possible.

In software process modeling we are aware of only one simulation model, (Raffo and Kellner 1999) which is similar to our model in that it shows individual activities and allows feedback in the process to have an impact on the stochastic durations of the tasks. This model uses statecharts to describe the code error detection and correction loop in the software process. The duration of the activities in the loop is stochastic, depends on the number of residual errors in the code, and decreases stochastically with each iteration through the loop. Yet, the model does not aim at scheduling and deals with only part of the software process. By explicitly modeling individual components and individual scheduling actions, our scheduling model is also much more fine-grained than system dynamics models that operate at the level of total workforce and overall schedule length (Abdel-Hamid and Madnick 1991, Collofello and Houston 1998, Madachy 1996, Tvedt and Collofello 1995).

Stochastic scheduling models are being studied in operations research, but these models are not appropriate for describing the software process. Closest to the dynamics of software engineering projects are 'stochastic project networks' (Neumann 1990, 1999). 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 that runs at the same time, nor on the duration of an activity that was performed earlier. This assumption does not hold for software projects. Therefore, our own scheduling model describes the rework and the feedback between activities in a project in a way that is novel in the literature on scheduling (Neumann 1999, Möhring 2000, Weglarz 1999).

This article is a revised and extended version of a conference article (Padberg 2004). The article builds upon our project scheduling model presented earlier (Padberg 2001, 2002a, 2002b). As compared to our previous work, we are now able to compute the *exact optimal policy* for each sample project. In addition, we now provide simulations for a whole *set of examples*. For each example, we compare the performance of the best list policy against the optimal policy and study the impact of important project characteristics on the scheduling decisions in a project.

3. MARKOV DECISION MODEL

This section describes how the software process is modeled as a Markov decision process (MDP). Additional details are provided in (Padberg 2001, 2002b).

3.1. States and Transitions

In the Markov decision model the software process moves between *states* over time. In each state a scheduling *action* is chosen by the manager. Once an action has been chosen, the transition to the next state is governed by the *transition probabilities*. Hence, the software process appears as a stochastic system.

In the software process MDP a transition to the next state takes place if a *phase* ends. By definition, a phase ends when a redesign occurs or a team finishes its current component. Both events are subject to chance. The project is completed if all the components are completed. The project is cancelled if the given *deadline* is exceeded.

For the software process, the state of the project carries information about the *net* progress (that is, excluding any rework) that has been made for each component up to the current point in time. Progress is measured in time units. In addition, the latest task assignment and the amount of rework yet to be completed for each component are included in the state. Finally, the project state shows the time left for the deadline to expire.

To illustrate the MDP mechanism, Figure 1 shows a cutout from a sample software process MDP with two teams and four components. The tree is to be read from the bottom to the top. The state variables ζ and η in the figure are *simplified* versions. The state only shows the net progress made on the components so far (this is the vector) and the time left until the deadline expires (this is the number); rework information has been left out. Please refer to (Padberg 2001, 2002a) for more details on the elements of the project state in the software process model.

Suppose that the current state of the process is ζ and that one time slice in model time corresponds to 1 month of work in real time. Up to this state a net progress of 2 months has been made on component 1, a net progress of 4 months on component 2, a net progress of 3 months on component 3, and a net progress of 1 month on component 4. There are 10 months left until the deadline is reached.

The dotted line shows part of a possible path of the project from state ζ onwards. In this state the manager chooses the scheduling action $a = (1,\ 3)$. This action assigns the first team to component 1 and the second team to component 3. Following some other policy the manager might have made a different scheduling decision, this is indicated in the figure by the other branch of the tree originating from state ζ.

After the manager has assigned the teams the next development phase begins. The outcome of this phase, that is, the next state η of the project, is subject to chance (the transition probabilities are not shown in the figure).

In our example the phase lasts for 4 months (the remaining time has decreased to 6 months in state η) and the net progress on components 1 and 3 has increased. In particular, component 1 has been completed in this phase, which is indicated by an infinity symbol ∞ for its net progress. Hence the first team can be reassigned by the manager, with his next scheduling action, to one of the unfinished components. Then the next phase begins. The project could have taken a different path by chance, and the next state could have been different from η. This is indicated in the figure by the other edges originating from the action node.

3.2. Input Data

To compute the transition probabilities for the software process or to simulate a project, some input data are required: the *base probabilities* and the *dependency degrees*.

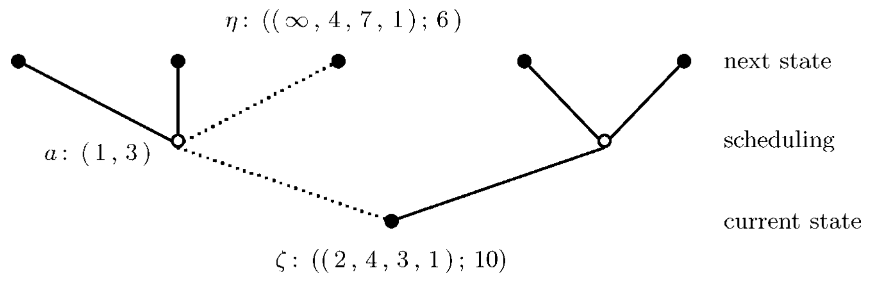

Figure 1. Cutout from a software process MDP

The base probabilities are a measure for the pace at which the teams have made progress in past projects. The dependency degrees are a probabilistic measure for the strength of the coupling between the components. We give examples for the input data in the next section.

3.3. Scheduling Policies

A rule or table that specifies which scheduling action is to be taken for each possible state is called a *strategy* or *policy*. When the scheduling strategy is fixed, the software process turns into a Markov chain.

A simple class of policies are the *list policies*. A list policy uses a fixed priority list for the components to prescribe an order in which they must be developed. When a team finishes its current component it is allocated to the next unprocessed component in the list. Hence the list policies keep all teams busy all the time. List policies are not very flexible but are common in practice.

In a stochastic setting, the task completion times are not known in advance. Thus, even for list policies the actual schedule depends on the order in which the teams finish their tasks, which is subject to chance.

3.4. Cost and Optimization

Each transition bears some *cost*. The cost usually equals the duration, but may also equal the personnel cost or some other cost metric. A *path* of the project consists of a sequence of state-action pairs. The cost of a path is computed by summing up the costs of all transitions in the path. The probability of a path is the product of the corresponding transition probabilities. For a fixed policy the *expected project cost* is the expected value for the cost of a full path from the project start to the project end.

Since the process model is stochastic, the best that one can achieve is a policy that minimizes the expected project cost. How to compute such an optimal policy is well known in operations research. The basic algorithms in use are backwards dynamic programming, value iteration, and policy iteration (Bertsekas 1995, Ross 1983).

The effort for computing an optimal policy grows exponentially with the number of states of the MDP. Hence, exact optimization is feasible only if the instance under study is not too large, that is, if the instance does not have too many states and transitions in each state (branching factor). This applies to the software process model as the number of states grows exponentially with the number of components in the software.

4. INPUT DATA FOR THE SAMPLE PROJECTS

4.1. Components and Teams

The sample project has four components (A, B, C, and D) of varying complexities and risk-levels. Differences between the components with respect to their complexities and risk-levels are modeled as follows:

- The probability distributions for the net development times of the components have different means and variances – see Table 1 in Subsection 4.3 for the values. The expected net effort for A and B is smaller than for C and D. The distributions for C and D have a higher variance than those for A and B.
- The probability that design problems will originate from a component varies from component to component – see Subsection 4.3 for details. The risk is lowest for component A and highest for component D.

According to their complexity and risk level, the components are ranked $A < B < C < D$.

Two teams (`One` and `Two`) work simultaneously on the project. A team may be specialized on certain components. Specialization is expressed by using base distributions as the input, which for the specialized teams have a smaller mean as compared to nonspecialized teams – see Subsection 4.4 for details.

4.2. List Policies

Since the sample project has four components, there are 24 different list policies, denoted as ABCD, ABDC, ... DCBA. For example, list policy ABCD initially assigns component A to team `One` and component B to team `Two`. Whichever team finishes first is allocated to component C. Finally, the next team to finish is allocated to component D.

4.3. Base Probabilities

The *base probabilities* $P(D^i_A(t))$ specify how likely it is that team i will finish component A after t time units. These probabilities are taken from a binomial distribution with parameters $n = 2$ and $p = 0.5$. The binomial distribution has been shifted to the right by 3 units and then scaled by a factor of 0.9. Scaling was applied in order to model in a way that there would be a 10% risk that a redesign will originate from component A during development.

Accordingly, the *base probabilities* $P(E^i_A(t))$ specify that there is a 5 percent risk that team i will report a design problem (and hence trigger a redesign) after 3 (4) time units while working on component A. These probabilities sum up to the risk of 10% that component A will cause a redesign.

The base distributions for component B have the same shape as for component A. The difference is that there is a higher risk that design problems will originate from component B (20% for B instead of 10% for A).

Without specialization, team `One` and `Two` have the same base distributions for the components A and B.

Figure 2 shows the base probabilities for components A and B as histograms.

The base probabilities for components C and D again are taken from a binomial distribution (with parameters $n = 3$ and $p = 0.5$). The binomial distribution has been shifted to the right by 5 units. The distribution was scaled by a factor of 0.8 for component C and 0.7 for component D. Therefore, component D is most likely to trigger a design change during development (risk of 30%).

Again, without specialization, team `One` and `Two` have the same base distributions for the components C and D. Figure 3 shows the base probabilities for components C and D as histograms.

The mean and variance of the base distribution for each team and component (without specialization) are given in Table 1. The table also shows the risk level for each component, that is, the probability that a design problem will originate from the component.

For each component there is also a probability distribution for the *rework time* required should the component be affected by a design change. For components A and B the rework time always is one time unit. For components C and D, one and two time units of rework occur with equal probability. The rework time is assumed to be independent of the team that is currently working on the component.

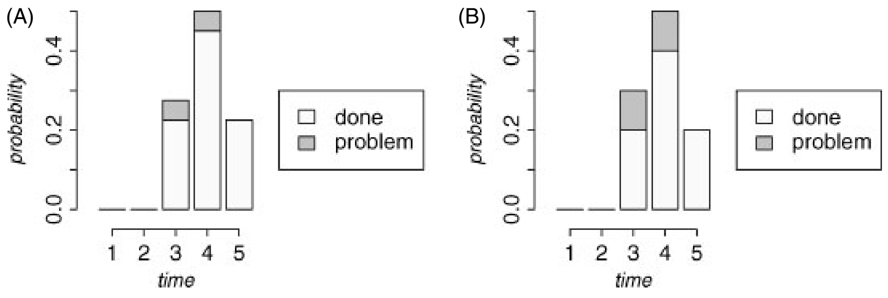

Figure 2. Base probabilities for components (A) and (B) (no specialization)

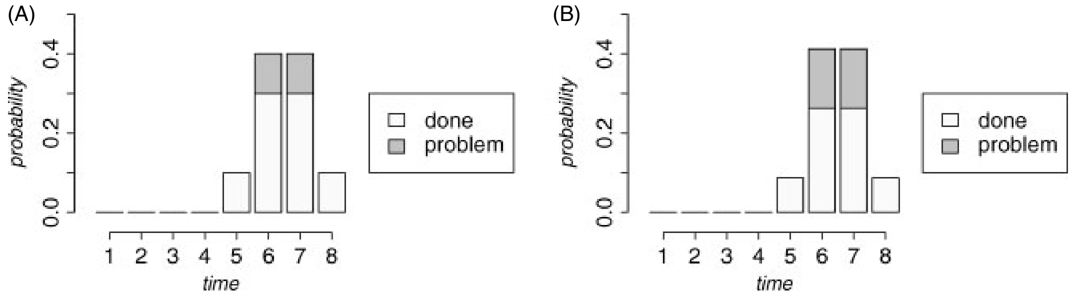

Figure 3. Base probabilities for components (C) and (D) (no specialization)

Table 1. Mean, variance, and risk level of base probabilities (no specialization)

Component	Team	Mean	Variance	Risk
A	One, Two	4.0	1.0	0.10
B	One, Two	4.0	1.0	0.20
C	One, Two	6.5	1.5	0.20
D	One, Two	6.5	1.5	0.30

4.4. Specialization

If a team is specialized on a particular component, its base distribution for that component is shifted to the left; the shape of the distribution remains unchanged. For example, if a team is specialized on component A, its corresponding base distribution is shifted to the left by 2.0 units. Owing to the shift, the expected net development time for a specialized team working on component A equals 2.0 units instead of 4.0 units for a nonspecialized team.

Similarly, with specialization the base distributions for the components B, C, and D are shifted to the left by 2.0, 3.0, and 3.0 time units, respectively. Figure 4 shows as histograms the base probabilities for a specialized team working on component A or C.

To codify if and how teams are specialized on certain components, we use a 4-digit notation (one digit for each component). 0 indicates no specialization, 1 indicates that team `One` is specialized on the component, and 2 indicates that team `Two` is specialized on the component. For example, a case of no specialization at all in any component is coded as 0000. As another example, the case of team `One` being specialized on C, team `Two` being specialized on D, and no team being specialized on A or B is coded as 0012.

For any given component, we use the same base distributions for team `Two` as for team `One`, except when one team is specialized on that component and the other team is not. Therefore, the case 1122 has the same input data and dynamics as 2211; only the roles of the two teams are switched. Hence, we need to study only half of the theoretically possible specialization codings in our simulations.

In this study there is only one specialized team for each component, if any. Having two specialized teams for the *same* component does not make much sense for this study: for that particular component, no performance difference between the two teams could be exploited by a dynamic scheduling policy.

Furthermore, we focus on having zero, two, or four cases of specialization for a team on components. The other cases are left for future study.

4.5. Coupling

The stronger the coupling between the components, the more likely it is that the design problems that originate in one component will propagate to other components and lead to rework there (ripple effects). The strength of the coupling between the components is measured by the *dependency degrees* $\alpha(K, X)$. Here, K and X are sets of components.

For example, $\alpha(\{B\}, \{A, B, D\})$ is defined as the probability that any changes in the design will extend exactly over the components A, B, and D given that the redesign was triggered by component B.

In this study, we vary the strength of the coupling by using four different sets of dependency degrees:

- Minimal coupling means that only those components in which a design problem occurs will have to be reworked, but no other components. With minimal coupling, tasks are being worked

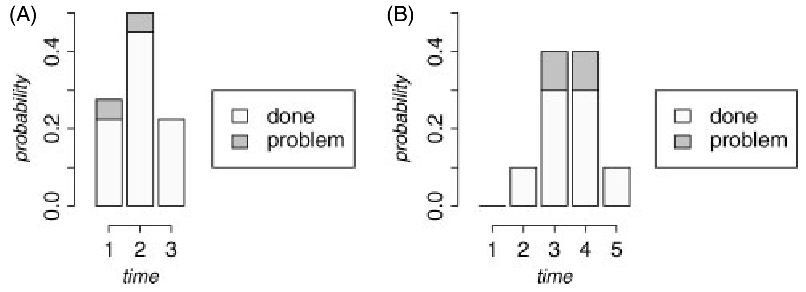

Figure 4. Base probabilities for components (A) and (C) with specialized team

on independently of each other. This is the best-case scenario with no feedback between different tasks.

- Maximal coupling means that all the components will always have to be reworked in case of a redesign, no matter in which component a design problem occurs. This is the worst-case scenario with maximal feedback between different tasks.
- Uniform coupling assumes no prior knowledge about the coupling between the components. Each set of components bears the same risk of being affected by rework in case of a redesign, no matter from which component the design problem arises.
- Asymmetric coupling assumes strong coupling between certain pairs of components. Please refer to (Padberg 2002a) or (2002b) for the specific values of the dependency degrees in this case.

5. OPTIMIZATION AND SIMULATION

The simulations in this study are based on a re-implementation of the stochastic simulation model presented earlier (Padberg 2002a, 2002b). The new implementation is written in C# under .NET instead of the ModL language that comes with the simulation environment EXTEND. The C# code is much faster, of course.

In addition to the process simulation code, we have implemented an algorithm to compute optimal scheduling policies in our model. The algorithm implements value iteration (Bertsekas 1995, Ross 1983) but also takes advantage of the fact that our software process MDP has no cycles. This yields a fast algorithm that combines value iteration with depth-first search for accessible states. The algorithm is exact, up to the precision of the floating point arithmetic.

For each set of dependency degrees (minimal, uniform, maximal, and asymmetric) and each team specialization coding (0000 through 2221) we computed the optimal policy for the given project setting, simulated all possible list policies, and finally simulated the optimal policy.

During optimization, we recorded the CPU time required for computing the optimal policy and the number of different states of the MDP. We also recorded the expected project cost under the optimal policy, which is usually called 'the optimal cost.'

During the simulations, we simulated 10,000 full project trajectories for each policy and project setting. We recorded the simulated cost distribution and corresponding mean cost. A comparison of the optimal cost against the simulated mean cost for the optimal policy shows that 10,000 simulation runs are enough to get a reliable picture for our sample project.

In general, the effort for computing an optimal policy grows exponentially with the number of states of an MDP. Table 2 shows the maximum number of MDP states and the maximum CPU time (min: sec) required for computing the optimal policy in our sample project settings of dependency degrees.

The computations were carried out on a Pentium III processor with 700 MHz clock rate and 256 MB of main memory.

6. ANALYSIS OF RESULTS

6.1. Optimal Cost

Cost equals project duration in this study. Recall that the optimal cost is defined as the expected cost of the optimal policy. Figure 5 shows the optimal cost for all the combinations of team specialization and coupling strength. The bars are arranged in groups of four, corresponding to the four different couplings studied in this article.

The stronger the coupling between the components, the higher is the optimal cost. This is as expected because the risk of change propagation and widespread rework increases with the strength of the coupling.

The optimal cost is highest if none of the teams are specialized; see the group of bars for the specialization coding 0000. The optimal cost, in general, is lowest if for each component there is a

Table 2. Number of MDP states and CPU time for schedule optimization

Coupling:	Minimal	Uniform	Maximal	Asymm
Min MDP states	2685	293,037	87,809	228,183
Max MDP states	2833	336,615	103,477	254,651
Max CPU time	0:01	8:19	0:59	6:13

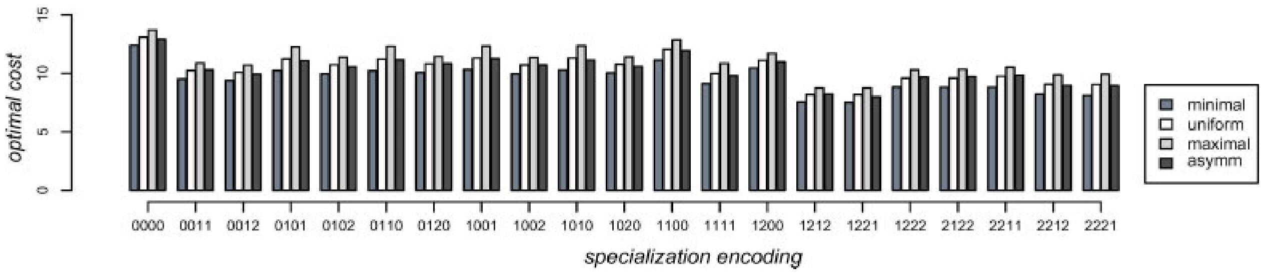

Figure 5. Optimal cost for varying coupling strength

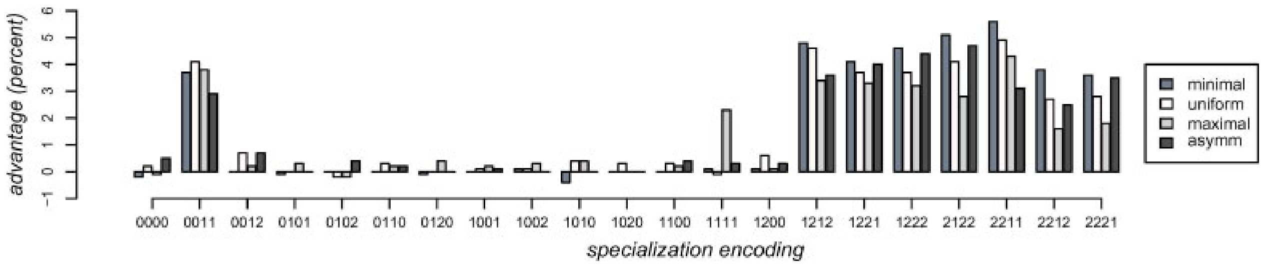

Figure 6. Relative cost advantage of optimal policy over best list policy for varying coupling strength

team that is specialized on that component; see, e.g. the group of bars for coding 1212. Again, this is as expected because specialized teams have a shorter expected net development time for the components than nonspecialized teams. An optimal policy takes advantage of this fact by assigning components to specialized teams.

Figure 6 shows the relative cost advantage (in percent) of the optimal policy over the best list policy for each project setting. For the optimal policy we use the expected cost computed during optimization; for the best list policy we use the simulated mean cost. We analyze this figure in the following subsections.

The remainder of this section is organized along the dimension 'degree of specialization'. We shall call the components A and B 'small' because they have a shorter expected net development time than the 'large' components C and D.

6.2. No Specialization of Teams

This setting refers to the specialization coding 0000. In this case the list policies fall into two broad categories depending on whether a policy manages to achieve a *balanced task assignment* (one large and one small component assigned to each team) or not; see Table 3.

Table 3. Optimal cost with no specialization

Coupling:	Minimal	Uniform	Maximal	Asymm
List policy median (balanced)	12.5	13.2	13.8	13.1
List policy median (unbalanced)	14.1	15.2	16.2	15.3
Best list policy (simulated)	12.4	13.1	13.7	13.0
Optimal policy (exact)	12.4	13.1	13.7	12.9

In the balanced case the simulated project cost comes close to the optimal cost, no matter how strong the coupling between the components is. Consequently, without specialization of teams the best list policy is close to the optimal in the sample project.

6.3. Specialized Team for Each Component

These settings refer to the specialization codings 1111, 1212, 1221, 1222, 2122, 2211, 2212, and 2221. For each component there is a specialized team, but in some cases the same team is specialized on more than half of the components.

In all these cases, except 1111, there is a significant performance gap between the best list policy and the optimal policy; see Figure 6. The cost advantage of the optimal policy ranges between 2 and 5.5%.

These percentages may look small. One must bear in mind, though, that our sample projects are small. Any policy will schedule two of the four components as its first action. If a list policy mimics the optimal policy by choosing the same first scheduling action as the optimal policy, then not too much can go wrong when assigning the remaining two components later in the project. We expect to observe a greater cost advantage for the optimal policy over the best list policy in future studies of larger projects.

We shall analyze in Subsection 6.5 why there is no performance difference between the best list policy and the optimal policy in case of the specialization coding 1111; the analysis requires a more detailed look at the schedules that actually occur for the optimal policy and the best list policies in that particular case.

Figure 6 also shows that the stronger the coupling between the components the smaller is the improvement that the dynamic scheduling achieves over the best list policy. A possible explanation is that because of the increased process feedback and rework that comes with a stronger coupling, the future behavior of the process is less predictable from the expected net component completion times. Therefore, it is difficult for a dynamic strategy to make better scheduling decisions than the list policies based on their knowledge of the base distributions.

Table 4 shows that the optimal cost is smallest if one of the teams is specialized on one large and one small component, whereas the other team is specialized on the remaining large and small component (cases 1221 and 1212). The optimal cost increases if the specialization is unbalanced, that is, if one team is specialized on more components than the other team (see, e.g. case 2212). The optimal cost also increases if one team is specialized on the two large components and the other team on the two small components (see, e.g. case 2211). These observations are independent of the coupling strength.

6.4. Specialized Team for Two Components

These project settings fall into three groups. The first group contains the cases 0011 and 0012, where (one or two) teams are specialized on the two large components but no team is specialized on the two small components. The second group contains the cases 0101, 0102, 0110, 0120, 1001, 1002, 1010, and 1020, where teams are specialized on one large and one small component but no team is specialized on the remaining two components. The third group contains the cases 1100 and 1200, where teams are specialized on the two small components but no team is specialized on the two large components.

Table 4. Optimal cost with one specialized team for each component

Coupling:	Minimal	Uniform	Maximal	Asymm
1212	7.5	8.2	8.7	8.2
1221	7.5	8.2	8.7	8.0
2212	8.2	9.1	9.9	9.0
2221	8.1	9.0	9.9	8.9
1222	8.8	9.6	10.3	9.7
2122	8.8	9.6	10.3	9.7
2211	8.8	9.7	10.5	9.8
1111	9.1	10.0	10.8	9.8

We say that the specializations *complement* each other if one team is specialized on one component and the other team on the other component, as opposed to one and the same team being specialized on both components.

In all three groups the best list policy always is close to the optimal; see Figure 6. An analysis of the initial action chosen by the optimal policy indicates that the best list policy takes advantage of the team specialization in the same way the optimal policy does: if the specialization is complementary, allocate the teams to its favorite components; if the specialization is noncomplementary, allocate the specialized team to the larger of its favorite components (and the other team to a component on which neither team is specialized).

For example, in case 0102 with uniform coupling, the optimal policy and the best list policy allocate team `One` to component B and team `Two` to component D as the first scheduling action. In case 0101 with uniform coupling, the optimal policy and the best list policy first allocate team `One` to component D (which is larger than component B) and team `Two` to component C (on which neither team is specialized).

Table 5 shows the optimal cost in each group. In all the three groups, complementary specialization shows a lower optimal cost than noncomplementary specialization (see, e.g. case 0101 *versus* 0102). The difference increases with the strength of the coupling.

Table 5. Optimal cost with specialized teams for two out of four components

Coupling:	Minimal	Uniform	Maximal	Asymm
0011	9.5	10.2	10.9	10.3
0012	9.4	10.1	10.7	9.9
0101	10.2	11.2	12.3	11.1
0102	10.0	10.7	11.4	10.6
0110	10.2	11.2	12.3	11.1
0120	10.1	10.8	11.4	10.4
1001	10.3	11.3	12.3	11.3
1002	10.0	10.7	11.3	10.7
1010	10.3	11.3	12.4	11.1
1020	10.0	10.8	11.4	10.6
1100	11.1	12.0	12.9	11.9
1200	10.5	11.1	11.7	11.0

Suppose that the strength of the coupling is fixed. If we focus on the project settings with the complementary specialization, the optimal cost is smallest in the first group and largest in the third group (see, e.g. case 0012 *versus* 1200). In addition, the optimal cost shows little variation within the middle group; the corresponding median lies between the optimal cost of the first and the third group. Similar statements hold good if we focus on the project settings with the noncomplementary specialization (see e.g. case 0011 *versus* 1100).

Given our particular choice of the base probabilities, the expected net development time for a component is about 50% shorter with a specialized team than with a nonspecialized team. Therefore it is better to have specialized teams working on the large, expensive components than on the small ones.

6.5. Typical Schedules

To visualize the schedules that actually occur for an optimal policy we use special Gantt charts called 'average schedules' that we first introduced in (Padberg 2002b). These charts are computed from the *mean* net development times and the *mean* rework times for each component, which in turn are collected from the simulation traces for the policy. Recall that for each project setting we have simulated 10,000 full project trajectories for the policies under study, including the optimal policy.

No Specialization

Figure 7 shows a typical average schedule for the optimal policy in the sample project, with no specialization and uniform coupling. The optimal cost for this project setting is 13.1 time slices. The numbers below the bars are the mean development times for the components (including rework).

In this typical scenario, the optimal policy first assigns component C to team `One` and component D to team `Two`. Team `One` finishes earlier than team `Two`, thus the optimal policy assigns component B to team `One`. The next team to finish is team `Two`, which finally is assigned to the last unfinished component, A. The simulations show that the project will take this path with about 61% probability. The expected cost in this scenario is 12.9 time slices.

There is another typical path that the project can take under the given input data; see Figure 8. The optimal policy assigns the components C and D as described before; but now the first team to finish its current task is team `Two`. Therefore, the optimal policy assigns component B to team `Two` instead of team `One`. Finally, component A is assigned to team `One`. The project will take this alternative path with about 38% probability. The expected cost in this scenario is slightly higher than before, 13.3 time slices.

Taking a second look at the behavior of the optimal policy it becomes apparent that the list policy CDBA will yield the *same* task assignments as the optimal policy, no matter whether component C is completed before component D, or vice versa. In fact, the simulations show that list policy CDBA is the best list policy in this setting and achieves the same expected cost as the optimal policy.

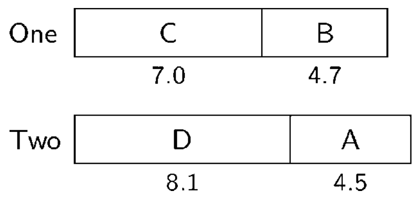

Figure 7. First typical schedule for the optimal policy (no specialization, uniform coupling)

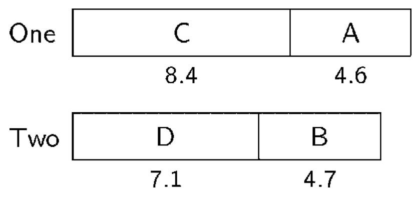

Figure 8. Second typical schedule for the optimal policy (no specialization, uniform coupling)

Specialization 1212
In case of projects with specialized teams, the Gantt charts show that the optimal policy tries to assign the teams to their favorite components. For example, if team One is specialized on components A and C and team Two is specialized on components B and D, the optimal policy *always* assigns the components accordingly; see Figure 9. Clearly, the project completion time with specialized teams is much shorter than in the previous cases, about 8.2 time slices.

One might ask whether this behavior can also be achieved with a list policy, similar to the previous project setting with no specialization. The answer is 'no'. To understand why, we briefly study several candidate list policies in the next few paragraphs.

The simplest idea is to mimic the optimal policy by applying list policy ABCD. Yet, this list policy will assign component C not to team One but to team Two in case component B is completed earlier than component A. The simulations show that this assignment will occur with a probability of about 30%, despite the fact that *on average* component A has a shorter completion time than component B. This assignment is unfavorable because it leads to a higher mean project completion time of about 11.2 time slices in the end.

Recall that the performance of a policy is roughly a weighted sum of the mean project completion times of its possible assignments, the weights being the probabilities with which the assignments occur. Hence, the unfavorable assignment drags down the total performance of list policy ABCD. Contrary to list policy ABCD, the optimal policy will assign component C to team One in any case, knowing that the specialization of team One on component C will payoff.

To avoid the problems with list policy ABCD and make sure that the large components will be assigned to their specialized teams, one might try list policy ADCB. Since component A is much smaller than component D, one would expect that team One will complete component A faster than team Two takes to complete component D; by definition, list policy ADCB would then assign component C to team One. This idea actually works, but has some other drawbacks. The simulations show that in about 20% of the cases team One will be so fast that it will finish A *and* C before team Two has finished D. Hence, list policy ADCB will assign the remaining component B also to team One. This assignment is unfavorable as it leads to a mean project completion time of about 10.4 time slices. As a result, list policy ADCB is not optimal.

Finally, one might try to properly assign the two large components C and D immediately, using list policy CDAB. However, this list policy will lead to the optimal assignment in only 47% of the cases. In another 34% of the cases, components A and B will be assigned to nonspecialized teams, which will result in a higher mean project completion time of about 10.2 time slices. In the remaining cases, both components A and B will be assigned to team One, leading to an unbalanced distribution of the workload. The mean project completion will then be about 10.4 time slices. Hence, list policy CDAB also is not optimal.

Specialization 1111
Figure 10 shows an average schedule for the optimal policy in the sample project with the specialization coding 1111 and uniform coupling. The simulations show that this scenario occurs with about 72% probability and an expected cost of 9.2 time slices. Because of the specialization of team One, this team (on average) can complete the set of components C, B, and A in about the same time as the nonspecialized team Two takes to complete component D.

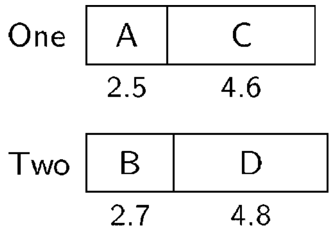

Figure 9. Average schedule for optimal policy (specialization 1212, uniform coupling)

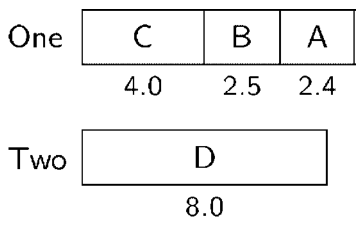

Figure 10. First typical schedule for specialization 1111 with uniform coupling

The other typical scenario in this setting is shown in Figure 11. This scenario occurs with about 22% probability.

In this latter scenario, team `Two` is faster in completing component D than before, whereas team `One` needs longer for component C than before. As a result, team `Two` will get allocated to component A and the expected cost for this scenario will be much higher, 11.8 time slices. Similar to the project setting with no specialization and uniform coupling, this scheduling behavior can also be achieved by a list policy, namely, policy CDBA. Again, simulations for list policy CDBA show that this is in fact true. Hence, the best list policy is optimal in this project setting, despite the fact that there is specialization of a team on certain components. This explains why specialization 1111 appeared as an exception in Section 6.3

An average schedule is an analysis tool that gives only an approximate picture of the corresponding project scenario. In particular, in many cases the sum of the mean development times for the individual components, as specified in the average schedule, differs somewhat from the overall mean project completion time, as given in Tables 4 and 5.

7. CONCLUSIONS

In this article we studied the performance of different scheduling strategies on a set of related sample software projects. For each sample project we simulated a large number of project trajectories under all possible list policies and under the optimal policy. The optimal policy in each setting was computed using stochastic dynamic programming; the best list policy in each setting was determined from the simulated cost distributions. All computations and simulations are based on our Markov decision model for software projects.

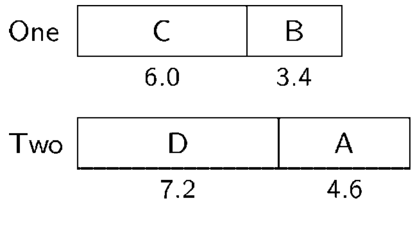

Figure 11. Second typical schedule for specialization 1111 with uniform coupling

The sample projects differ in certain characteristics of the project and software product. In the analysis of the simulation results for the different project settings, we focused on the cost difference between the best list policy and the optimal policy. We studied the impacts of the strength of the coupling between the components and of the degree of team specialization on the project cost. In the sample projects, we tried to take a representative sample from the large input space along the dimensions 'specialization' and 'coupling'.

In our sample projects, the cost advantage of the optimal policy over the best list policy ranges between 2 and 5.5% of the expected project cost. These figures may look small, but policy optimization will yield a much larger improvement over simple strategies as the size of the project grows because a dynamic strategy will then have more opportunities to take advantage of its knowledge of the component sizes, team productivities, component coupling, and current project state. This effect is being studied in our ongoing work.

Even our small project-instances have upto several hundred thousand states, and the optimal policy stores an optimal action for each state. Hence it is hard to understand the scheduling decisions made by an optimal policy from the state-action table. In other words, it is difficult to grasp how the optimal policy actually *behaves*. This problem will become more urgent for larger examples in future studies. Process simulation is a very helpful tool here as it quickly provides valuable information about a policy. It is easy to collect and visualize data such as the actual task assignments and the cost for each component while simulating a project trajectory.

To sum up, we have achieved the following results in this article:

- Small project instances in our model actually can be optimized in reasonable computing time.
- The best list policy in general is not optimal.
- The higher the degree of specialization of the teams, the larger is the cost advantage of the optimal policy over the best list policy.
- The stronger the coupling between the components, the smaller is the improvement of the optimal policy over the best list policy.

These results have important implications for both researchers and practitioners. For researchers, our results show that systematic schedule optimization for software projects is viable, even if we allow our scientific models to reflect the complicated dynamics of software projects, such as the strong feedback between activities and the uncertainty about the future path of a project. Optimization studies for such complex scheduling models seemed out of reach up to now. Doing research on software project scheduling is also worthwhile because theoretical results achieved in this area promise substantial improvements in software project management practices.

For practitioners, our results indicate that much cost can be saved by moving from simple scheduling heuristics, such as list policies, to more dynamic scheduling policies. For example, postponing the assignment of some component until a team with more expertise on that type of work is available can prove beneficial, even if some other team is ready and would remain idle in the meantime. The fact that a high uncertainty about the task durations is natural in software development might make detailed planning appear as a useless task, but our results indicate that better planning is possible even under conditions of uncertainty. In addition, our results underline that a strong overall coupling within a software not only is problematic from a design and maintenance perspective, but also makes scheduling much harder.

It is well known that the computational effort for the exact optimization of an MDP, in general, grows exponentially with the size of the problem. In our case, the computational effort grows with the number of components in the software. We succeeded in computing an exact optimal policy for the small sample projects studied in this article, but it is unlikely that realistic project instances with dozens of components can be exactly optimized.

From a computational point of view, at least two countermeasures can be taken. First, the size of the state space depends on the choice of the unit for the time axis in the model. If we choose a time slice corresponding to 1 month instead of 1 week, the input probability distributions have a much smaller support and the state space shrinks considerably. Since we have modeled the process at a more coarse-grained level in this case, the results of our computations will provide less information, but will still be useful.

Second, we can use simulation-based techniques to *approximate* the optimal policies for large instances. We have described such techniques in (Padberg 2001). Although we will lose some information as compared to the exact optimization, we will still be able to increase our understanding of scheduling under uncertainty by studying the approximate solutions. This is a subject of ongoing research.

Having just reflected on the scalability of an exact optimization approach, we would like to emphasize that exact optimization of large project instances is *not* our primary research goal. Our research agenda actually is:

- to study more project instances in detail through a combination of exact optimization and process simulation;
- on the basis of these examples, to understand and explain the mechanisms and key factors that drive the scheduling process in our model;
- to use our insight to derive useful, practical guidelines that indicate how to best schedule a software project under conditions of uncertainty;
- to validate these guidelines through simulation of large instances and through empirical case studies in real software projects.

ACKNOWLEDGEMENTS

The simulation and optimization program was written by Max Horstmann as part of his diploma thesis (Horstmann 2004). This research is financially supported by the Deutsche Forschungsgemeinschaft DFG (project title: OASE).

REFERENCES

Abdel-Hamid TK, Madnick SE. 1991. *Software Project Dynamics*. Prentice Hall.

Bertsekas DP. 1995. *Dynamic Programming and Optimal Control*. Athena Scientific.

Collofello J, Houston D. 1998. A system dynamics simulator for staffing policies decision support. Proceedings of the 31st Annual Hawaii International Conference on System Sciences, 103–111.

Gray A, Mac Donell S. 1997. A comparison of techniques for developing predictive models of software metrics. *Information and Software Technology* **39**: 425–437.

Horstmann M. 2004. Untersuchung verschiedener Methoden zur Berechnung optimaler Strategien bei der Planung von Softwareprojekte. Diplomarbeit, Universität Karlsruhe, (in German).

Madachy R. 1996. System dynamics modeling of an inspection-based process. Proceedings of the 18th International Conference on Software Engineering ICSE, 376–386.

Möhring RH. 2000. Scheduling under uncertainty: Optimizing against a randomizing adversary. Proceedings of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, LNCS 1913. Springer: 15–26.

Neumann K. 1990. Stochastic Project Networks, Lecture Notes in Economics and Mathematical Systems 344. Springer.

Neumann K. 1999. *Scheduling of Projects with Stochastic Evolution Structure*. Kluwer: 309–332.

Padberg F. 2001. Scheduling software projects to minimize the development time and cost with a given staff. Proceedings of the 8th Asia-Pacific Software Engineering Conference APSEC, 187–194.

Padberg F. 2002a. Using process simulation to compare scheduling strategies for software projects. Proceedings of the 9th Asia-Pacific Software Engineering Conference APSEC, 581–590.

Padberg F. 2002b. A discrete simulation model for assessing software project scheduling strategies. *International Journal on Software Process Improvement and Practice SPIP* **10**(3–4): 127–139.

Padberg F. 2004. Computing optimal scheduling policies for software projects. Proceedings of the 11th Asia-Pacific Software Engineering Conference APSEC, 300–308.

Raffo DM, Kellner MI. 1999. Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives. *Elements of Software Process Assessment and Improvement* EL Emam K, Madhavji NH. IEEE Computer Society Press: 297–341.

Ross SM. 1983. *Introduction to Stochastic Dynamic Programming*. Academic Press.

Shepperd M, Schofield C, Kitchenham B. 1996. Effort estimation using analogy. Proceedings of the 18th International Conference on Software Engineering ICSE, 170–178.

Srinivasan K, Fisher D. 1995. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering TSE* **21**(2): 126–137.

Tvedt JD, Collofello JS. 1995. Evaluating the effectiveness of process improvements on software development cycle time via system dynamics modeling. Proceedings of the 19th International Computer Software and Applications Conference COMPSAC, 318–325.

Weglarz J. 1999. *Project Scheduling. Recent Models, Algorithms, and Applications*. Kluwer.

Wittig GE, Finnie GR. 1994. Using artificial neural networks and function points to estimate 4GL software development effort. *Australian Journal of Information Systems* **1**: 87–94.