A Discrete Simulation Model for Assessing Software Project Scheduling Policies

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

Abstract

Good project scheduling is an essential, but extremely hard task in software management practice. In a software project, the time needed to complete some development activity is difficult to estimate. Often, the completion of activities is delayed due to unanticipated rework which is caused by feedback in the process.

In this paper, we show how process simulation can be used to support managers in finding good schedules for their software projects. We present a novel, stochastic simulation model which is tailored to the special dynamics of software projects, and which explicitly takes a scheduling strategy as input. The model represents task assignments, staff skill levels, component coupling, and rework caused by design changes. The simulation model is implemented in the ModL language of the general-purpose graphical simulation tool EXTEND.

As an illustration of our simulation model, we study the performance of various list policies for a small sample project. The simulations quickly show the impact that the choice of the list policy will have on the progress and completion time of the sample project. To explain the performance difference between the list policies, we use the simulation traces to provide a detailed analysis of the task assignments which actually occur in the simulations.

Keywords. *Process Simulation, Project Scheduling, Rework Modeling, Stochastic Process Modeling.*

1. Introduction

To cut development cost and meet tight deadlines in short staffed software projects, it is essential that managers optimize the project plan and schedule. Good software project scheduling is an extremely hard task in practice, though. The time needed to complete a software development activity is difficult to estimate since it depends not only on technical factors, but also on human factors such as the experience of the developers. Even worse, it is typical for software projects that the completion of tasks is delayed because of unanticipated rework; such rework is caused by feedback in the development process.

In this paper, we show how to apply *software process simulation* to the scheduling problem. We present a discrete-time, stochastic simulation model which is tailored to the special dynamics of software development projects. The model explicitly takes a scheduling strategy (also called a policy) as input. A strategy specifies for each possible state of the project which action to take, such as reassigning or stopping some task. The model is stochastic in order to account for the uncertainty inherent to the software process with respect to the duration of activities and the occurence of events.

The simulation model is an *implementation* of the probabilistic scheduling model for software projects which we have presented earlier [14]. In the scheduling model, teams work in parallel on the software's components. Unplanned changes in the system design can occur at any time and lead to rework. Since the components are coupled, for example, through common interfaces, changes which originate in one part of the system can propagate to other parts of the system. As input to the model, statistical data collected during past projects and high-level design data about the current project are required.

In each simulation, one possible full path of the software project is simulated. When the policy is fixed, the output of a series of simulations can be used to measure the performance of that policy. For example, the output of the simulations for a fixed policy can be used to compute and display the probability distribution for the project completion time, as well as the corresponding expected value. In addition to the project completion time, other project features can be observed in the simulations, such as the total amount of rework in the project or the development times of the individual components.

The simulation model makes it easy to experiment with different scheduling strategies for a given project. A manager can see quickly how the expected completion time of his project changes when he changes the strategy or some other part of the project setting, such as the number of available teams. This way, a manager can evaluate and compare different strategies and choose the one which he can expect to work best for his next project.

As a sample application of our simulation model, we study all possible *list policies* for a small hypothetical project. A list policy prescribes an order in which the components must be worked on, but the final schedule depends on which path the project actually takes: the next team to finish must work on the next component in the list. List policies are commonly used and reflect management practice.

The sample project consists of four components and two teams. Although the project is small, it is not clear aforehand which list policy the manager should prefer because of the probabilistic nature of the development process. Based on a detailed analysis of the performance of the task assignments which actually occur in the simulations, we are able to identify three different ways to achieve a good average completion time for the sample project. In addition, we find that the performance of a task assignment can be sensitive to the project context in which the assignment occurs.

Software engineering currently offers little help to software project managers as far as scheduling is concerned. The existing *effort estimation models* do not support scheduling; they only provide an estimate for the total effort required for a project, expressed in man-days. Some models also provide a distribution over time of the manpower needed for a project. Both the classical curve-fitting models and the more recent models [19, 20, 23] do not show individual tasks and developers. Thus, deriving a detailed schedule is not possible.

Process-centered software engineering environments [2, 4, 7] guide project managers and developers during real software projects, but they do *not* support finding good schedules. Each environment comes with a process modeling language (often more than one) to formally describe the software development process in detail, capturing the activities to be carried out, the staff involved, the products to be developed, and the tools available. Managers and developers use a process model by "enacting" it; that means, they step through the model in accordance with the progress of the real project, using an interpreter for the process modeling language. Although a manager can explicitly assign tasks to developers, the environments do not assist the manager in making that assignment best possible with respect to meeting a given deadline and budget. The manager also can specify a fixed duration (and cost) for each activity, but the strong impact of feedback in the software process on the duration of the activities is not modeled.

An exceptional stochastic simulation model for part of the software process is presented in [17]. The model uses statecharts to describe the code error detection and correction loop. The duration of the activities in the loop is stochastic and depends on the number of residual errors in the code. With each iteration through the loop, the number of residual errors decreases according to some probability distribution. The process iterates until a prescribed quality level is reached. Although this model does not aim at scheduling, it is similar to our model in that it shows individual activities and allows feedback in the process to have an impact on the stochastic activity durations of the tasks.

Operations research provides stochastic scheduling models, but these models are not appropriate for describing the software process. Closest to the dynamics of software engineering projects are "stochastic project networks" [10, 11]. A stochastic project network can model parallel execution of activities and repeated execution of activities. Yet, the duration of an activity must not depend on any other activity which runs at the same time, nor on the duration of an activity which was performed earlier. In other words, in a stochastic project network different threads of execution are stochastically independent, as are different activities belonging to the same thread. These assumptions do not hold for software projects. The particular way in which our own model describes the feedback between activities is novel in scheduling [9, 11, 15].

Our scheduling model describes the software process at a high level of abstraction. Classical process phases such as coding or testing are not modeled. Still, the scheduling model captures much of the dynamics of software projects, representing varying staff skill levels, rework caused by design changes, component coupling, and changing task assignments. By explicitly modeling individual components and individual scheduling actions, our model is more fine-grained than *system dynamics models* which operate at the level of total workforce and overall schedule length [1, 3, 8, 21].

This paper is a largely revised and extended version of a conference paper [16].

2. Scheduling Model

2.1. Project dynamics

The simulation model is an implementation of the stochastic scheduling model for software projects which we have presented earlier [14]. The scheduling model captures much of the dynamics of software projects, representing varying staff skill levels, design changes, component coupling, rework, and changing task assignments.

In the model, the software product is developed by several *teams.* The teams work in parallel. Based on some early high-level design, the software 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 required that there are enough teams to work simultaneously on all uncompleted components. The assignment of the components to the teams may change during the project.

The teams do not work independently. From time to time some team might detect a problem with the software's high-level design. To eliminate the problem, the high-level design gets revised. Since the components are coupled, for example, through common interfaces, such a design change is likely to affect more than one component and team. This is the way how feedback between the different activities in the project occurs in the model: all components which are affected by the design change will have to be *reworked,* not only the component where the problem was detected.

2.2. Scheduling actions

In the model, a software project advances through a sequence of *phases.* 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 a team completes all rework on a component which already had been completed earlier in the project but had to be reworked because of a design change. Note that our definition of phases is different from classical waterfall models.

Scheduling *actions* take place only at the end of the phases. Possible scheduling actions are: assigning a component to a team; starting a team; stopping a team. Scheduling at arbitrary points in (discrete) time is not modeled. 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 change.

At the end of a phase, the manager may also interrupt some of the teams and re-allocate them to other components. It is not required that a team has completed its current component before being re-allocated. The decision which team to allocate to which component at the end of a particular phase is based on the manager's scheduling *strategy* or *policy.*

2.3. Probabilities

The scheduling model is probabilistic: events will occur only with a certain probability at a particular point in time. In particular, the following events are subject to chance:

- the point in time at which some component is completed;
- the points in time at which design changes occur;
- the set of components which must be reworked due to a design change;
- the amount of rework caused by a design change.

Mathematically, the scheduling model is a Markov decision process [18]. For more details and the explicit formulas see [12, 14].

2.4. Input data

In order to compute the probabilities in the scheduling model, respectively, simulate a project path, the model requires the following input data: the base probabilities, the dependency degrees, and the scheduling strategy. We explain the input data in detail in the next section and give examples.

3. Sample Project

3.1. Architecture

We use the following small, hypothetical project as a running example throughout this paper.

The sample project is a client-server system which has four components. The client contains a front-end, component A, and a large application part which does some pre-processing, component C. The server contains an administrator front-end, component B, and a large application kernel, component D. The client and the server are coupled only through the application components C and D. In particular, there is no direct link between the two front-end components.

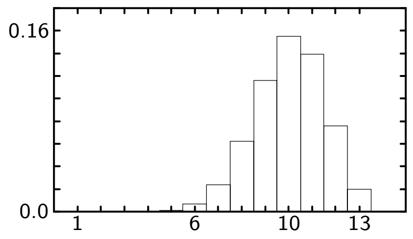
There are two teams in the project, team One and team Two. The teams work in parallel. The complexity of the components and the productivity of the teams are reflected in the probability distributions which are used as input for the simulations, see below.

3.2. Base probabilities

The *base probabilities* are a measure for the pace at which the teams have made progress in previous projects. For each team and component, there is a separate set of base probabilities. Probability $\mathrm{P}(\mathcal{D}_k^i(t))$ specifies how likely it is that team i will finish component k after having worked on it for t time units. These probabilities refer to the *net* development times, excluding any rework. Similarly, probability $\mathrm{P}(\mathcal{E}_k^i(t))$ specifies how likely it is that team i will report a design problem after having worked on component k for t time units. By definition, the base probabilities for a given team and component must add up to one.

The base probabilities must be discrete distributions with finite support. Therefore, we use *skewed binomial* distributions as the base probabilities for the sample project. The parameters n and p for the binomial distributions are listed in TABLE 1. The distributions were rounded to one decimal digit and shifted to the right by one (the simulations need at least one time unit to do anything). Then, the distributions were scaled (see the scale factors given in TABLE 1) in order to achieve that for a fixed team and component the base probabilities sum up to one. The binomial distributions are similar in shape to Rayleigh distributions which have been shifted to the right and whose (otherwise infinite) tail has been compressed, see FIGURE 1.

Figure 1. The base probabilities $\mathrm{P}(\mathcal{D}_{\mathsf{C}}^{\mathsf{One}}(t))$ for team One completing component C after t units.



The larger the scale factor used with $\mathrm{P}(\mathcal{D}_k^i(t))$ the less likely it is that component k will trigger a design change during the project. For example, if component C is assigned to team One, the probability that the component will be completed without triggering a design change is 60 percent, see the entry for the scale factor of $\mathrm{P}(\mathcal{D}_{\mathsf{C}}^{\mathsf{One}}(t))$ in TABLE 1. Note that the scale factors in TABLE 1 depend only on the component, not on the team which is allocated to the component.

Table 1. Base probabilities for the sample project.

distribution	n	p	peak	scale
$\mathrm{P}(\mathcal{D}_{\mathsf{A}}^{\mathsf{One}}(t))$	7	0.7	6	0.8
$\mathrm{P}(\mathcal{E}_{\mathsf{A}}^{\mathsf{One}}(t))$	6	0.5	4	0.2
$\mathrm{P}(\mathcal{D}_{\mathsf{A}}^{\mathsf{Two}}(t))$	10	0.7	8	0.8
$\mathrm{P}(\mathcal{E}_{\mathsf{A}}^{\mathsf{Two}}(t))$	10	0.5	6	0.2
$\mathrm{P}(\mathcal{D}_{\mathsf{B}}^{\mathsf{One}}(t))$	7	0.8	7	0.8
$\mathrm{P}(\mathcal{E}_{\mathsf{B}}^{\mathsf{One}}(t))$	6	0.55	4	0.2
$\mathrm{P}(\mathcal{D}_{\mathsf{B}}^{\mathsf{Two}}(t))$	10	0.8	9	0.8
$\mathrm{P}(\mathcal{E}_{\mathsf{B}}^{\mathsf{Two}}(t))$	10	0.55	7	0.2
$\mathrm{P}(\mathcal{D}_{\mathsf{C}}^{\mathsf{One}}(t))$	12	0.75	10	0.6
$\mathrm{P}(\mathcal{E}_{\mathsf{C}}^{\mathsf{One}}(t))$	13	0.45	7	0.4
$\mathrm{P}(\mathcal{D}_{\mathsf{C}}^{\mathsf{Two}}(t))$	16	0.75	13	0.6
$\mathrm{P}(\mathcal{E}_{\mathsf{C}}^{\mathsf{Two}}(t))$	20	0.45	10	0.4
$\mathrm{P}(\mathcal{D}_{\mathsf{D}}^{\mathsf{One}}(t))$	14	0.75	12	0.4
$\mathrm{P}(\mathcal{E}_{\mathsf{D}}^{\mathsf{One}}(t))$	15	0.45	8	0.6
$\mathrm{P}(\mathcal{D}_{\mathsf{D}}^{\mathsf{Two}}(t))$	20	0.75	16	0.4
$\mathrm{P}(\mathcal{E}_{\mathsf{D}}^{\mathsf{Two}}(t))$	26	0.45	13	0.6

The parameters for the binomial distributions are chosen in such a way that *on average*

- team Two has a 30 percent lower productivity than team One;
- the net development times for the front-end components A and B are about the same for a given team;
- the net development times for the core application components C and D are much longer than for the front-end components;
- the risk that high-level design problems will occur is much higher for the application components than for the front-end components;
- design problems can be expected to occur mainly

after two thirds of the net development time of a component;

- component D has the largest expected effort and has the highest risk of triggering design changes.

In addition, for each component k there is a probability distribution $\mathrm{P}(\mathcal{R}_k(t))$ which specifies the amount of *rework time* t required if that component has to be reworked because of a design change. Again, we use skewed binomial distributions which were shifted to the right by one, see TABLE 2.

Table 2. Rework time probabilities for the sample project.

distribution	n	p	peak
$\mathrm{P}(\mathcal{R}_\mathsf{A}(t))$	1	0.25	1
$\mathrm{P}(\mathcal{R}_\mathsf{B}(t))$	1	0.25	1
$\mathrm{P}(\mathcal{R}_\mathsf{C}(t))$	4	0.5	3
$\mathrm{P}(\mathcal{R}_\mathsf{D}(t))$	4	0.55	3

3.3. Dependency degrees

The *dependency degrees* are a probabilistic measure for the strength of the coupling between the components. The stronger the coupling is the more likely it is that high-level design problems which originate in one component will propagate to other components, leading to rework. The dependency degree $\alpha(K, X)$ by definition is the probability that changes in the software's design will extend over exactly the set X of components given that the problems causing the redesign were detected in the set K of components.

The dependency degrees for the sample project are listed in TABLE 3. Blank entries correspond to zero. For example, the entry in row C and column CD of TABLE 3 corresponds to $\alpha(\mathsf{C}, \mathsf{CD})$ and specifies that there is a 35 percent probability that design problems detected in component C will lead to design changes which affect the components C and D, but no other components.

For the sample project, the dependency degrees are chosen in such a way that

- the core components C and D are strongly coupled;
- each front-end is strongly coupled to its application component;
- changes must propagate along the interfaces between the components;
- there is only a limited risk that a design change which originates in a front-end, say component A, will propagate to the *other* application component, in this case component D;
- design changes in one front-end can have an impact on the other front-end only if the intermediate components C and D are affected.

3.4. List policies

The *scheduling strategy* specifies for each possible state of the project which scheduling action to take. There is a huge number of possible different strategies that can be applied to a project. The scheduling model and, hence, the simulation model make no assumptions about the strategy, except that the information used by the strategy when choosing a scheduling action must be contained in the project state (refer to section 4 for the definition) and the model input data.

In this paper, we focus on a well-known class of scheduling strategies, *list policies.* A list policy uses a fixed priority list for the components to prescribe an order in which the components must be developed. When a team finishes its current component, it is allocated to the next unprocessed component in the list. If several teams become available at the same time, the team with the smallest id is allocated to the first unprocessed component in the list, the team with the second-smallest id is allocated to the second unprocessed component in the list, and so on.

A list policy keeps all teams busy all the time. As opposed to policies which prescribe for each component which team exactly must work on this component, a list policy does not have to wait for "the right" team to become available before development of the next component can start.

In a probabilistic setting, the task completion times are not known in advance. Thus, a priority list does not completely pre-determine to which team a particular component will actually get assigned; the actual schedule (task assignments and their timing) depends on the order in which the teams finish their tasks, which is subject to chance.

Since the sample project has four components, there are $\mathrm{fac}(4) = 24$ different list policies for the sample project. For example, the list policy CDAB initially assigns component C to team One and component D to team Two. Whichever team finishes its task first will work on component A. Finally, the next team to finish will work on component B.

With list policies, a team works on its component until completion without interruption. An exception

Table 3. Dependency degrees for the sample project (in percentages).

	A	B	C	D	AB	AC	AD	BC	BD	CD	ABC	ABD	ACD	BCD	ABCD
A	20					60							15		5
B		20							60					15	5
C			10			15				35			20	10	10
D				10					15	35			10	20	10
AB					20						20	20			40
AC						30							55		15
AD							10					10	50		30
BC								10			10			50	30
BD									30					55	15
CD										40			20	20	20
ABC											10				90
ABD												10			90
ACD													40		60
BCD														40	60
ABCD															100

occurs when a component must be reworked which already had been finished earlier in the project. In such a case, the team which had finished that particular component is interrupted, reworks the component, and then resumes its current task. Recall that the scheduling model allows interrupting and re-allocating teams at the end of the phases.

4. Implementation

The simulation model is a discrete-time simulation written in the ModL language of the general-purpose, graphical simulation tool EXTEND [6]. We have chosen EXTEND because it is in widespread use, well-documented, and offers a free runtime system which allows to run pre-compiled models. ModL is a C-like language which allows to develop continuous and discrete models from scratch.

EXTEND models are subdivided into blocks. The scheduling strategy is implemented as a separate block in the simulation and hence can be easily replaced. Other blocks allow to enter or read from file the base probabilities and dependency degrees. The current *state* of the project is visible in another block. The state of the project includes the net development time spent on each component so far, the amount of rework left for each component, the project duration up to this point, and the current task assignment (that is, which team is working on which component).

The simulation determines which step the project will take next by *throwing dice.* There is a separate die for each currently active component. Suppose that team i is working on component k, the net development time for component k equals ζ time units, and there is no rework left for component k. The die for a component throws one of three events: complete, design problem, continue. The die behaves according to the base probabilities. For example, event complete for component k is thrown with probability

$$\frac{\mathrm{P}(\mathcal{D}_k^i(\zeta+1))}{1 - \sum_{t=1}^{\zeta}\{\mathrm{P}(\mathcal{D}_k^i(t)) + \mathrm{P}(\mathcal{E}_k^i(t))\}}.$$

After having thrown the die, the net progress for component k is incremented by one.

The simulation handles an active component which has some rework left differently. Instead of throwing a die, the rework is decremented for each simulation step. In particular, components under rework can not throw design problem events. As soon as a component's rework time has been counted down to zero, net progress can be achieved again. Once a component has been completed, only rework may occur for the component in the sequel.

If one or more of the components' dice have thrown a design problem, the simulation determines the set of those components which are affected by the resulting design change by throwing another die. Suppose that a design problem was detected simultaneously by each component listed in the set K. According to the dependency degrees, the additional die throws X as the set of affected components with probability $\alpha(K, X)$. Any component can be affected by a design change, including inactive components and components which have been completed earlier in the simulation.

After a design problem event, the amount of rework time required for each of the affected components is determined using yet another die which behaves according to the probabilities of rework time. The rework added to the affected component k equals ϱ with probability $\mathrm{P}(\mathcal{R}_k(\varrho))$.

The current project phase ends if throwing the components' dice results in at least one design problem or complete event. The simulation then asks the block containing the scheduling strategy for the task assignment to be used in the next phase.

5. Simulation Results

5.1. List policy performance

Even for the small sample project it is *not* obvious which list policy a manager should prefer because of the probabilistic nature of the development process. In particular, the amount of rework caused by design changes is subject to chance, which leads to additional uncertainty about the task completion times.

To find the best list policy for the sample project, we ran 1000 full project simulations for each of the 24 possible lists. For each simulation, we observed the project completion time. TABLE 4 gives a ranking of the list policies based on the mean project completion times observed in the simulations.

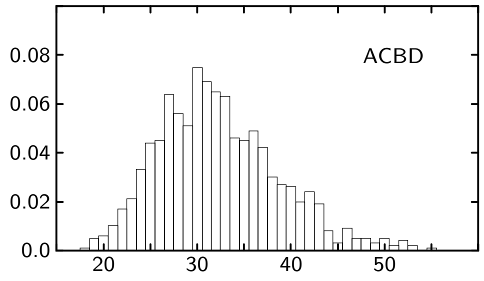
Table 4. Mean simulated project completion time for the sample project with different list policies.

policy	mean	σ	rank
ABCD	31.5	6.1	21
ABDC	28.4	5.4	12
ACBD	32.2	6.3	24
ACDB	27.4	4.9	3
ADBC	28.7	5.5	13
ADCB	28.4	4.9	11
BACD	30.1	6.0	15
BADC	28.2	5.4	9
BCAD	31.8	6.2	22
BCDA	27.4	5.0	4
BDAC	29.0	5.5	14
BDCA	27.8	5.0	7
CABD	31.3	7.5	20
CADB	30.3	5.8	16
CBAD	31.9	7.5	23
CBDA	30.4	6.0	17
CDAB	28.3	5.2	10
CDBA	28.1	5.1	8
DABC	30.8	7.0	19
DACB	27.1	5.0	2
DBAC	30.7	7.1	18
DBCA	27.6	5.0	6
DCAB	27.6	4.8	5
DCBA	26.9	4.5	1

There is a considerable performance gap between the best policies, which have a mean project completion time of 27 time units, and the worst policies, which have a mean of over 31 time units. In particular, the mean for the best policy DCBA is about 16 percent shorter than for the worst policy ACBD.

From the 1000 simulated project completion times for each list policy, we can compute a histogram for the project completion time. FIGURES 2 AND 3 show the histogram for the worst policy ACBD, respectively, the best policy DCBA.

Figure 2. Histogram of simulated project completion times for policy ACBD.



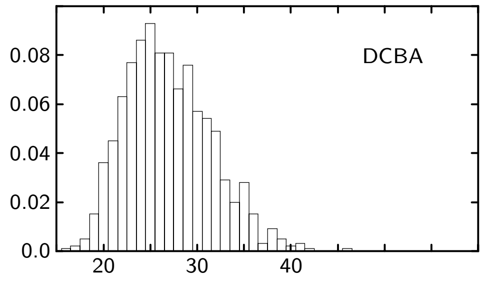


Figure 3. Histogram of simulated project completion times for policy DCBA.

The histograms make the difference in performance between the two list policies apparent. From the histograms, a manager can also compute the *risk* that a given deadline will be missed. For example, with policy DCBA the risk of not completing the project within 30 time units equals 22 percent; with policy ACBD, this risk is much higher, namely, 57 percent.

5.2. Statistical significance

To show that the differences in the mean simulated project completion times really are significant, we ran for each pair of list policies a two-sample Wilcoxon test with the simulated completion times as the samples. This amounts to 276 pairwise tests. We use the Wilcoxon test instead of the t-test because the project completion times are not normally distributed. The Wilcoxon tests show that the results of our simulations indeed are significant:

- Most importantly, the performance advantage of the best policy DCBA over the other list policies is highly significant at the 0.1 percent level (p-value of 0.001 or smaller) except for a few cases, see the upper part of Table 5. Only policy DACB has a performance similar to DCBA, which is reflected in the small diff of 0.2 for the means and a large p-value of 0.73. In the other cases, the difference is clearly significant.
- A difference in the mean project completion time of 0.6 or larger is statistically significant at the 5 percent level (p-value of 0.05) except for some cases, see the middle part of Table 5. The difference between policies BACD and DABC clearly is significant. The other cases are not important, since the mean project completion times are larger than 30 time units.
- A difference of 0.5 or 0.4 is statistically significant at the 15 percent level (p-value of 0.15) except for some cases, see the lower part of Table 5. The difference between ADBC and CDAB, respectively, BADC and BDCA, still has some significance. The other cases are not important, because the mean completion times are larger than 30 time units.
- We consider a difference of up to 0.3 too small to be meaningful, though in a number of these cases the difference is statistically significant.

Table 5. Special cases from Wilcoxon tests.

compared policies		diff	p-val
DCBA	ACDB	0.5	0.11
DCBA	BCDA	0.5	0.07
DCBA	DACB	0.2	0.73
DCBA	DBCA	0.7	0.007
DCBA	DCAB	0.7	0.003
BACD	DABC	0.7	0.06
CABD	CADB	1.0	0.20
CABD	CBDA	0.9	0.24
BACD	DBAC	0.6	0.29
CABD	DBAC	0.6	0.19
CABD	DABC	0.5	0.54
CADB	DABC	0.5	0.48
ABCD	CBAD	0.4	0.55
ADBC	CDAB	0.4	0.38
BADC	BDCA	0.4	0.19
CADB	DBAC	0.4	0.99
CBDA	DABC	0.4	0.53

5.3. Actual task assignments

To gain some understanding why a particular list policy shows the performance observed in the simulations, the task assignments which *actually occur* in the simulated projects are of central importance. Recall that the actual schedule in a simulation depends on the order in which the teams finish their tasks, which is subject to chance. Therefore, we observed for each simulation and component which team was allocated to that component.

To specify an assignment for the sample project, we use a 4-digit notation. The first digit is the number of the team which was allocated to component A, the second digit is the number of the team which was allocated to component B, and so on. For example, to specify that team One was allocated to components B and D, while team Two was allocated to components A and C, we use the notation 2121.

Table 6 shows for each list policy the actual task assignments and the relative frequency with which the assignments have occured among the 1000 simulations for that list policy. Only assignments with a frequency of more than 10 percent are listed. For each policy and assignment, Table 6 also shows the mean simulated project completion time corresponding to that assignment.

For example, list policy CDAB results in the assignment 1112 in 56 percent of the simulated projects, with a mean project completion time of 27.2 units. In 43 percent of the simulations, policy CDAB results in the assignment 1212, with a longer mean project completion time of 29.8 units. The performance of policy CDAB is a mixture of the peformance for the two assignments 1112 and 1212.

The performance difference between the assignments for a given policy (as listed in Table 6) is statistically significant (Wilcoxon test) at the 0.1 percent level, with the following exceptions: policy DCBA (diff = 0.1, not significant); policy DCAB (diff = 0.3, significant at the 10 percent level); and policy CABD (diff = 0.9, not significant due to the small sample size for 2112).

Table 6. Actual assignments, mean net component development times, and mean component rework times.

			project time		mean net develop time				mean rework time			
policy	assign	freq	mean	σ	A	B	C	D	A	B	C	D
ABCD	1212	0.91	31.9	6	5.9	9.2	10.4	16.7	0.8	0.9	4.3	4.8
ABDC	1221	0.92	28.3	5.5	5.9	9.2	13.5	12.1	0.8	0.9	4.2	4.8
ACBD	1121	0.71	31.2	6.2	5.8	6.5	14	12.2	0.9	0.9	4.9	5.2
	1122	0.29	34.7	6.1	6.4	7.1	12	17.1	0.6	1.1	2.7	4.3
ACDB	1221	0.94	27.4	4.9	6.1	9.1	13.2	11.9	0.8	0.9	3.9	4.7
ADBC	1112	0.95	28.5	5.4	6	6.6	10.3	16.7	0.7	0.8	4	4.7
ADCB	1112	0.50	27.4	4.8	5.7	6.7	9.8	17.5	0.5	1	3.4	5.6
	1212	0.50	29.4	4.9	6.4	9.2	10.7	15.8	1	0.8	4.6	3.9
BACD	2112	0.76	31	5.7	8.6	6.5	10.3	16.8	0.9	0.8	4.2	4.4
	2121	0.24	27.1	5.7	6.7	7.4	13.7	12.3	0.6	1.2	3.8	5
BADC	2112	0.22	29.5	5.3	6.8	7.3	10.5	16.8	0.6	1.3	4	5.5
	2121	0.78	27.9	5.4	8.6	6.4	13.6	12.3	0.9	0.8	4.1	4.3
BCAD	1121	0.69	30.8	6.1	5.8	6.5	13.9	12.1	0.8	0.8	4.5	5
	1122	0.31	34.1	6	6.6	7	12	17	0.6	0.9	2.8	3.9
BCDA	2121	0.95	27.4	5.1	8.2	6.7	13.3	12.2	0.9	1	4.1	4.9
BDAC	1112	0.95	28.9	5.5	6	6.5	10.2	16.9	0.8	0.9	4.4	4.9
BDCA	1112	0.49	27.2	4.9	6.2	6.4	9.7	17.7	0.7	0.9	3.5	5.3
	2112	0.51	28.4	5	8	6.9	10.7	15.9	0.9	0.8	4.4	3.6
CABD	2112	0.12	29.8	4.7	9.2	6.8	8.2	16.3	0.6	0.8	2.3	3.3
	2211	0.83	30.7	7.2	8.1	9.1	10.4	12.6	0.8	0.9	4.3	4.9
CADB	2112	0.88	30.9	5.5	8.1	6.7	10.4	16.7	0.8	1	4.6	5.3
	2211	0.12	25.6	5.2	9.3	9.2	8.4	12.9	0.6	0.9	2.5	3.5
CBAD	1212	0.23	32.9	6	6.5	10	8.9	16.7	0.6	1.1	3.2	4.6
	2211	0.72	31	7.4	8.3	8.9	10.5	12.7	0.8	0.9	4.3	4.9
CBDA	1212	0.76	31.4	5.6	6	8.8	10.7	16.8	0.9	0.9	4.7	5
	2211	0.24	27.3	6.1	8.6	10	8.9	12.8	0.6	1.1	3.1	4.5
CDAB	1112	0.56	27.2	4.9	5.9	6.6	9.8	17.5	0.6	1	3.6	5.2
	1212	0.43	29.8	5.2	6.5	9.2	10.7	16.1	1	0.8	4.5	4
CDBA	1112	0.51	27.4	4.8	6.3	6.6	9.6	17.7	0.7	0.9	3.6	5.4
	2112	0.47	28.8	5.4	8.1	6.9	10.7	16	0.8	0.9	4.5	3.9
DABC	2211	0.78	29.4	6.2	8.3	9.2	10.7	11.7	0.7	0.9	3.8	4
	2221	0.18	38	6.5	7.4	8.7	13.6	12.8	0.8	1.3	6.3	9
DACB	2121	0.95	27.1	4.9	8	6.8	13.3	12	0.7	0.9	4.1	4.7
DBAC	2211	0.76	29.2	6.3	8.3	9.2	10.6	11.7	0.8	0.8	3.8	3.8
	2221	0.19	37.7	6.3	7.4	8.6	13.6	12.8	0.9	1.3	6	8.6
DBCA	1221	0.96	27.8	4.9	6	9.1	13.3	12	0.8	0.9	4.2	4.8
DCAB	1221	0.56	27.6	4.5	6.2	9	13.7	11.4	0.8	0.8	4.1	3.9
	2121	0.35	27.3	5.1	8.3	7	12.4	12.6	0.7	1.1	3.7	6.1
DCBA	1221	0.36	26.8	4.5	6.3	9.1	12.4	12.6	0.7	1	3.5	5.5
	2121	0.58	26.9	4.6	8.3	6.9	13.8	11.4	0.7	0.8	4	3.7

5.4. Average schedules

For a given list policy, each actual assignment corresponds to a typical path of the project, or *scenario.* A project scenario can be visualized using an "average schedule," that is, a Gantt chart computed from the mean net development times and mean rework times for each component. These numbers are computed from the simulation traces for the policy and are listed in TABLE 6 for the sample project.

For example, when applying list policy CDAB, the sample project can proceed in two different ways. At the project start, component C is assigned to team One and component D is assigned to team Two. Since team One is faster than team Two and component C is smaller than component D, in both scenarios component C is completed faster than component D. Thus, component A (which is next on the list) gets assigned to team One.

The two scenarios for policy CDAB differ in the next scheduling action, as is shown by the average schedules in FIGURES 4 AND 5. The numbers below the bars are the mean development times for the components, including all rework. The shaded area of each bar is proportional to the rework spent on the component.

In FIGURE 4, the mean development time including rework for component C $(9.8 + 3.6 = 13.4)$ plus component A $(5.9 + 0.6 = 6.5)$ is shorter than for component D $(17.5 + 5.2 = 22.7)$. Therefore, component B also gets assigned to the fast team One.

Figure 4. Average schedule for policy CDAB with assignment 1112.

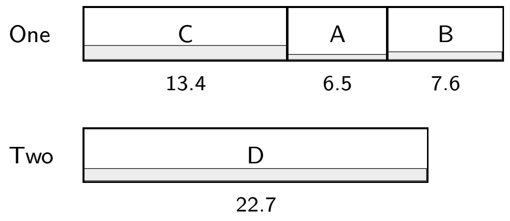


Figure 5. Average schedule for policy CDAB with assignment 1212.

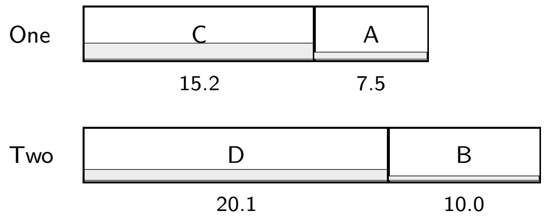


Figure 6. Average schedule for best policy DCBA with balanced assignment 2121.

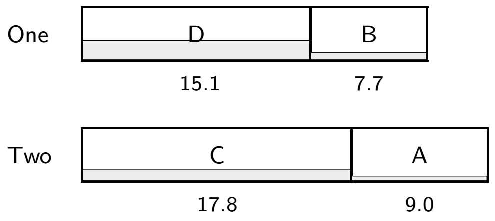
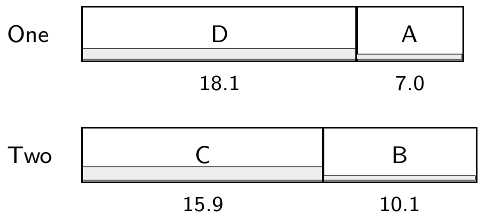


Figure 7. Average schedule for best policy DCBA with balanced assignment 1221.



In Figure 5, component D is completed earlier than component A. Thus, component B gets assigned to the slow team Two, and the project takes longer.

5.5. Good and bad policies

The best policy DCBA in many simulated projects leads to the assignment 2121, see Figure 6. With this assignment, the fast team works on the largest component and the slow team on the second largest component; furthermore, each team works on one of the remaining small components. Such an assignment is called *balanced,* because the size of the components is balanced by the productivity of the teams. The other balanced assignment for the sample project is 1221, see Figure 7. Balanced assignments are favorable, as can be seen from the average schedules for other list policies as well, such as ACDB and BCDA.

An assignment where each team works on one large and one small component, but where the slow team works on the largest component, in general is much less preferable, see for example the average schedules for policy ABCD with assignment 1212 and policy CADB with assignment 2112. There are some exceptional scenarios where the assignment 2112 shows a better performance; we shall discuss this case in the next subsection.

An alternative to a balanced assignment is revealed by policy CDAB. In about half of the projects, CDAB leads to the assignment 1112, see Figure 4. With this assignment, the slow team works on the largest component, but all the remaining components are assigned to the fast team. The assignment 1112 yields a good performance for the policies ADCB, BDCA, CDAB, and CDBA. These policies do not rank as high as the best policy DCBA, though, since in the other half of the projects they lead to the less favorable assignments 1212 or 2112.

The assignment 2221 is similar to the assignment 1112, but with the roles of the two teams switched. Hence, this assignment shows a poor performance, see for example policy DABC.

Policies which assign *both* large components to the fast team in general are a bad choice, see for example policy CABD with assignment 2211. Assigning both large components to the slow team is even worse, see for example policy ACBD with assignment 1122. There are some exceptional scenarios where the assignment 2211 shows a good performance; we shall discuss this case in the next subsection.

Finally, policies which assign the largest component and both small components to the same team lead to a long project completion time, see the policies ACBD and BCAD with the assignment 1121.

5.6. Context sensitivity

For some assignments, the mean project completion time depends on the *project context* in which that assignment arises. For example, in most cases the assignment 2211, which assigns both large components to the fast team, yields a long project completion time. Yet, there are two exceptions: if the assignment occurs for policy CADB or CBDA, the performance is as good as for the best balanced assignment. The reason is that for policies CADB and CBDA the assignment 2211 can only occur if team One finishes the large component C faster than team Two finishes its first small component (A or B). Such a project context is not very likely to occur, but points to a fast project completion.

Other assignments exhibit a similar sensitivity of their performance to the project context. The policies ADBC and BDAC almost always lead to the assignment 1112, but do not show as good a performance as, for example, ADCB does with this assignment. The difference is that for ADBC and BDAC team One works on the large component C *after* both small components. This does not hold for ADCB and the other policies which lead to assignment 1112. The assignment 1112 can occur for policies such as ADCB only if team One finishes component C and one small component more quickly than team Two finishes component D, which points to a fast project completion. For ADBC and BDAC, component C is always assigned to team One, independent of the project's progress.

6. Conclusions

We have presented a stochastic scheduling model for software projects and its implementation as a EXTEND simulation model. Using a small hypothetical project and list policies as an example, we have shown how to apply the simulation model to analyze the performance of scheduling strategies for software projects.

A stochastic model is more realistic for software projects than a deterministic model which assumes that the task durations are all known at the project start. In a stochastic setting, the duration of the project cannot be forecast exactly; we must rely on probability distributions and expected values instead. As a consequence, the best we can achieve is a policy which minimizes the *expected* project duration.

Besides a scheduling strategy, we must specify the base probabilities and the dependency degrees of the project as input to the simulations. The base probabilities for a component depend upon various human and technical factors, for example, the software process employed, the complexity of the component, and the skills of the team working on the component. For real projects, the base probabilities must be computed from empirical data collected during past projects and will reflect the specific development environment in a company; see [13] for an example. The dependency degrees reflect the strength of the coupling between the components and must be computed from the high-level design of the software.

The base probabilities for a component just model its *net* development time; they do not include rework caused by design changes. The amount of rework in the project is subject to chance and depends on the strength of the coupling between the components. The development time for some component is the sum of the net development time and all the rework time for that component. As a result, the expected completion time of a project cannot be computed from the base probabilities alone.

In a stochastic project setting, the final schedule of the project is subject to chance. Thus, we have used the *average schedule* for each task assignment which actually occurs for a given policy as a tool for analyzing the performance of the policy. The average schedule for some assignment gets computed from the simulation traces. Since an average schedule combines the mean values for the net development time and rework time of each component, it gives only an approximate picture of the corresponding project scenario.

The performance of a task assignment for the sample project often depends on the project context in which the assignment occurs. This result provides evidence that strategies which are more adaptive to the current state of the project than list policies might yield improved schedules. Recall that for a list policy, the scheduling actions take only into account which components are completed and which previously completed components must be reworked. No other information about the project's current state is used. In contrast, a *dynamic* strategy would base its scheduling decision at the end of a phase on the full data about the current state of the project, including such data as the current net development times of the components. In addition, the project input data would be utilized, such as the expected remaining net development times for the components, which can be computed from the base probabilities. For example, a dynamic strategy could allocate most of the staff to those tasks which are expected to be closest to completion.

Our stochastic scheduling model is not limited to list policies. The scheduling strategy is implemented as a separate block in our EXTEND simulation model. Therefore, the list policies used in this paper can be easily replaced by other, more dynamic strategies. The performance of the dynamic strategies then can be analyzed using the same simulation techniques as were used for list policies. This is work in progress.

7. Acknowledgments

This research has been supported by the Deutsche Forschungsgemeinschaft DFG (project title: OASE).

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 TOSEM 6:3 (1997) 283–328

[3] Collofello, Houston, e.a.: "A System Dynamics Simulator for Staffing Policies Decision Support", Proceedings of the Annual Hawaii International Conference on System Sciences 31 (1998) 103–111

[4] Derniame, Ali Kaba, Wastell: *Software Process: Principles, Methodology, and Technology.* Lecture Notes in Computer Science 1500, Springer 1999

[5] El Emam, Madhavji: *Elements of Software Process Assessment and Improvement.* IEEE Computer Society Press 1999

[6] EXTEND, `http://www.imaginethatinc.com/`

[7] Finkelstein, Kramer, Nuseibeh: *Software Process Modelling and Technology.* Research Studies Press 1994

[8] Madachy: "System Dynamics Modeling of an Inspection-Based Process", Proceedings of the International Conference on Software Engineering ICSE 18 (1996) 376–386

[9] Möhring: "Scheduling under Uncertainty: Optimizing Against a Randomizing Adversary", Proceedings of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, Springer LNCS 1913 (2000) 15–26

[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 [22] 309–332

[12] Padberg: "Towards Optimizing the Schedule of Software Projects with Respect to Development Time and Cost", Proceedings of the International Software Process Simulation Modeling Workshop PROSIM 2000

[13] Padberg: "Estimating the Impact of the Programming Language on the Development Time of a Software Project", Proceedings of the International Software Development and Management Conference ISDM / AP-SEPG (2000) 287–298

[14] Padberg: "Scheduling Software Projects to Minimize the Development Time and Cost with a Given Staff", Proceedings of the Asia-Pacific Software Engineering Conference APSEC 8 (2001) 187–194

[15] Padberg: "A Stochastic Scheduling Model for Software Projects", Dagstuhl Seminar on Scheduling in Computer and Manufacturing Systems, June 2002, Dagstuhl Report No. 343

[16] Padberg: "Using Process Simulation to Compare Scheduling Strategies for Software Projects", Proceedings of the Asia-Pacific Software Engineering Conference APSEC 9 (2002) 581–590

[17] Raffo, Kellner: "Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives", see [5] 297–341

[18] Ross: *Introduction to Stochastic Dynamic Programming.* Academic Press 1983

[19] Shepperd, Schofield, Kitchenham: "Effort Estimation Using Analogy", Proceedings of the International Conference on Software Engineering ICSE 18 (1996) 170–178

[20] Srinivasan, Fisher: "Machine Learning Approaches to Estimating Software Development Effort", IEEE Transactions on Software Engineering TSE 21:2 (1995) 126–137

[21] Tvedt, Collofello: "Evaluating the Effectiveness of Process Improvements on Software Development Cycle Time via System Dynamics Modeling", Proceedings of the International Computer Software and Applications Conference COMPSAC 19 (1995) 318–325

[22] Weglarz: *Project Scheduling. Recent Models, Algorithms, and Applications.* Kluwer, 1999

[23] Wittig, Finnie: "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort", Australian Journal of Information Systems 1 (1994) 87–94