Applying Process Simulation to Software Project Scheduling

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

1. Introduction

Good project scheduling is an essential yet extremely hard task in software management practice. The time needed to complete a development activity usually is known only roughly. Often, the completion of an activity is delayed due to unanticipated rework.

In this paper, we show how process simulation can be utilized in order to support software managers in finding good schedules for their projects. We present a novel, discrete simulation model for software projects which explicitly takes a scheduling policy 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 [4].

As a first application of our project simulation model, we systematically study the performance of various so-called list policies for a sample project. The simulations quickly show what impact each list policy has on the expected progress and completion time of the sample project. We also provide a detailed analysis of the task assignments which actually occur in the simulations. The analysis clearly identifies the reasons why the list policies perform as observed.

2. Scheduling Model

The simulation model is an implementation of the stochastic scheduling model for software projects which we have presented earlier [7, 9].

2.1. Project dynamics

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 scheduling 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. In particular, staff becomes available when some team completes its component. 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.

2.3. Strategies

The decision which team to allocate to which component at the end of a phase is made by the manager's scheduling *strategy.* The strategy specifies for each

possible *state* of the project which scheduling action to take. The state of the project includes the development time spent on each component so far, the project duration up to this point, the amount of rework left for each component, and the current task assignment.

There is a huge number of possible different strategies that can be applied to a project. The simulation model makes no assumptions about the strategy, except that the information used by the strategy when choosing an action must be contained in the project state and the model input data. The strategy is implemented as a separate block in the simulation model; thus, the strategy can be easily replaced.

2.4. 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.

2.5. 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 and the dependency degrees.

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 set of base probabilities which specify how likely it is that the team will need a prescribed amount of time to finish the component, report a high-level design problem, or finish reworking the component after a design change. The base probabilities must be computed from empirical data collected during previous projects.

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 degrees must be computed from the high-level design of the software.

3. Sample Project

As an example, we use the simulation model to study the performance of a well-known class of scheduling strategies, list policies, for a small sample project.

3.1. Architecture

The sample project consists of four components and two teams. 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. The base probabilities are chosen such that: team Two has a lower productivity than team One; components A and B have a similar complexity; components C and D require much more effort than components A and B. The dependency degrees are chosen to reflect that components C and D are strongly coupled. Please refer to [9] for details on the input data.

3.2. 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. 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.

4. Simulation Results

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. To find the best list policy for the sample project, we run 1000 full project simulations for each of the 24 possible list policies and compare the results.

4.1. List policy performance

For each simulation, we observe the project completion time as a measure for the performance of the pol-

icy. TABLE 1 gives the mean of the project completion times for each list policy.

Table 1. Mean simulated project completion time for the sample project with different list policies.

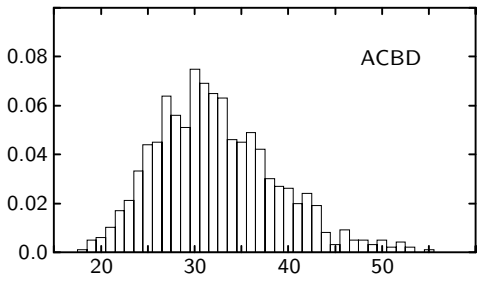
policy	mean	σ	policy	mean	σ
ABCD	31.5	6.1	CABD	31.3	7.5
ABDC	28.4	5.4	CADB	30.3	5.8
ACBD	32.2	6.3	CBAD	31.9	7.5
ACDB	27.4	4.9	CBDA	30.4	6.0
ADBC	28.7	5.5	CDAB	28.3	5.2
ADCB	28.4	4.9	CDBA	28.1	5.1
BACD	30.1	6.0	DABC	30.8	7.0
BADC	28.2	5.4	DACB	27.1	5.0
BCAD	31.8	6.2	DBAC	30.7	7.1
BCDA	27.4	5.0	DBCA	27.6	5.0
BDAC	29.0	5.5	DCAB	27.6	4.8
BDCA	27.8	5.0	DCBA	26.9	4.5

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.

For all interesting cases, a difference in the mean project completion time of 0.4 or larger is statistically significant (two-sample Wilcoxon test). Most importantly, the performance advantage of the best policy DCBA over the other list policies is highly significant.

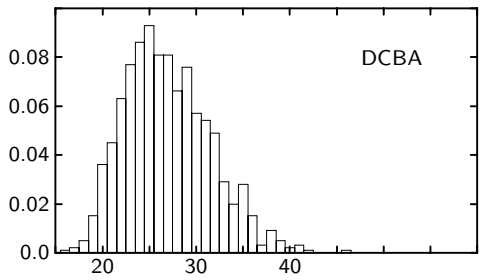
From the 1000 simulated project completion times for each list policy, we can compute a histogram for the project completion time. FIGURES 1 AND 2 show the histogram for the worst policy ACBD, respectively, the best policy DCBA.

Figure 1. Histogram of simulated project completion times for policy ACBD.



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.

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



4.2. 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 observe 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 2 shows for the list policies ACBD, CDAB, and DCBA 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 2 also shows the mean simulated project completion time corresponding to that assignment.

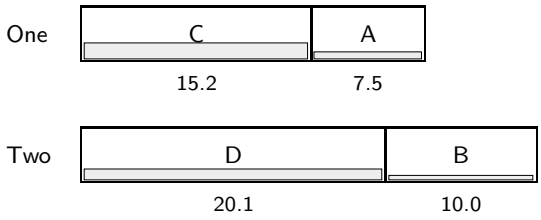
For example, list policy CDAB results in the assignment 1112 in 56 percent of the simulated projects, with a mean project completion time of 27.2 units. In 43 percent of the simulations, policy CDAB results in the assignment 1212, with a longer mean project completion time of 29.8 units. The performance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

Table 2. Actual assignments, mean net component development times, and mean component rework times for the sample project with selected list policies.

			project time		mean net develop time				mean rework time			
policy	assign	freq	mean	σ	A	B	C	D	A	B	C	D
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
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
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

Figure 3. Average schedule for policy CDAB with assignment 1112.

Figure 4. Average schedule for policy CDAB with assignment 1212.



4.3. 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 2 for the sample project and the list policies ACBD, CDAB, and DCBA.

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 3 AND 4. 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 3, 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. In FIGURE 4, component D is completed earlier than component A. Thus, component B gets assigned to the slow team Two, and the project takes longer.

4.4. Good and bad policies

The best policy DCBA in many simulated projects leads to the assignment 2121, see TABLE 2. 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. Balanced assignments are favorable, as can also be seen from the average schedules for other list policies (not listed in the table).

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. Also, policies which assign *both* large components to the fast team in general are a bad choice, as are policies which assign the largest component and both small components to the same team.

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 3. With this assignment, the slow team works on the largest component, but all the remaining components are assigned to the fast team. Policy CDAB does not rank as high as the best policy DCBA, though, since in the other half of the projects it leads to the less favorable assignment 1212.

5. Conclusions

In this paper, we have presented a stochastic scheduling model for software projects and its implementation as a EXTEND simulation model. Using a small project and the class of list policies as example, we have shown how to use simulation for analyzing the performance of scheduling strategies for software projects.

A stochastic model is more realistic for software projects than a deterministic model. In our model, the development time for some software component is the sum of the net development time and all the rework time for that component. The net development time and the amount of rework are subject to chance. The rework in a project also depends on the strength of the coupling between the components.

In a stochastic setting, the duration and final schedule 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 strategy which minimizes the *expected* project duration. We have used the concept of *average schedule* as a tool for visualizing and analyzing the performance of a scheduling strategy in a stochastic setting.

Process simulation is an established technique for evaluating the impact of process changes. A stochastic simulation model for part of the software process which is related to our work is presented in [10]. That 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 and depends on number of residual errors in the code, which decreases with each iteration through the loop. Although this model does not aim at scheduling, it is similar to our model by showing individual activities and allowing feedback in the process to have an impact on the stochastic activity durations.

Some system dynamics models also address the problem of project staffing [1, 2]. Yet, by modeling individual teams and components as well as explicit task assignments, our model is much more fine-grained than system dynamics models, which operate at the level of developer pools, task pools, and overall schedule.

The particular way in which our model describes feedback between activities is novel not only in software engineering, but also in operations research [8, 11]. Closest to the dynamics of software projects are *stochastic project networks* [5, 6]. A stochastic project network can model parallel execution of activities and repeated execution of activities. 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, different threads of execution are stochastically independent, as are different activities belonging to the same thread. These assumptions do not hold for software projects.

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 we have used for list policies. This is work in progress.

References

[1] Abdel-Hamid, Madnick: *Software Project Dynamics*. Prentice Hall, 1991

[2] 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

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

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

[5] Neumann: *Stochastic Project Networks*. Lecture Notes in Economics and Mathematical Systems 344, Springer 1990

[6] Neumann: "Scheduling of Projects with Stochastic Evolution Structure", see [11] 309–332

[7] 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

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

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

[10] Raffo, Kellner: "Modeling Software Processes Quantitatively and Evaluating the Performance of Process Alternatives", see [3] 297–341

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