A Decision Support System for Software Project Management

Software engineering projects are complex sociotechnical systems whose behavior is the result of many technical and human factors, from personnel skills to development tools and processes to business goals. Project managers must identify the right combination of these factors to obtain the desired results. In many fields, managers get support from sophisticated decision support systems, but these haven’t yet entered the software engineering mainstream. A hybrid modeling approach can quickly produce process models that can provide project managers accurate predictions, help them design the desired project trajectory, and validate process changes. The NASA Software Engineering Laboratory process provides a case study. Donzelli, P. Software, IEEE67-75July-Aug. 2006
Decision support systems combine individuals’ and computers’ capabilities to improve the quality of decisions.1 Usually adopted in manufacturing to design floor plans and optimize resource allocation and performance, DSSs are penetrating increasingly complex application areas, from insurance fraud detection to military system procurement to emergency planning.

Although researchers have suggested many approaches,3–8 DSSs haven’t yet entered the main stream of software engineering tools. The complexity of the software process and its sociotechnical nature are often mentioned as the main obstacles to their adoption. As DSSs are developed for other equally complex application areas, we need to identify approaches that can overcome these difficulties and enable project managers to exploit DSSs’ capabilities in their daily activities. A hybrid two-level modeling approach is a step in this direction.

**Applying DSSs to software engineering**

In emergency planning, decision makers must deal with technical and human factors such as unpredictable disaster characteristics (for example, an earthquake’s magnitude, duration, and epicenter), people’s behavior under stress, and potential domino effects resulting in additional instability (such as power outages). In this context, decision makers adopt DSSs to perform sophisticated what-if analyses, design the best recovery plans (for example, escape routes and resource allocation), and validate interventions (such as transferring emergency personnel among locations) before applying them. DSSs’ capabilities—accurate prediction in an unstable environment, support of trajectory design, and validation of interventions—would benefit software engineering project managers.

**Accurate prediction in an unstable environment**

Before starting a project, the manager could accurately estimate dynamic project behavior (for example, staffing, schedule, effort, and product quality). DSSs would also be useful in case of variable external conditions (such as changing requirements).
Shaping the project trajectory

The manager could shape the project’s behavior by choosing the right combination of development and quality-assurance activities and correctly allocating effort and resources to better meet business goals. For example, the manager could make sure to

- use most of the effort before a certain date to release staff to other projects,
- reach the testing phase at a time when necessary resources will be more available, or
- achieve a certain level of quality as soon as possible to have a product suitable to fill a market opportunity by a controlled release.

Validating interventions

Projects can derail because of unexpected events, such as a personnel strike, a new technology’s unpredicted low performance, or an outsourcer’s delay in delivering a component. Additionally, business concerns might demand a change in trajectory (for example, you might need to shorten the delivery time because a competing product is going to market sooner than expected). In any of these scenarios, project managers must act. Before acting, however, they must be sure that their actions will lead to the desired results. In software engineering, it’s well known that simple actions can produce counterintuitive feedback (for example, adding people to a late project could make it even later). So, for example, a project manager could check that adopting an extra review activity won’t result in a schedule overrun. A testing manager could verify staff availability before committing himself or herself to higher defect-reduction targets. A process designer could estimate the process overhead that introducing higher concurrency levels among process activities could generate.

Software process models and simulation

Process models are a promising means to understand, predict, manage, and improve the software process. Different types of models have different capabilities and degrees of complexity.

At the bottom level in terms of capability and complexity are analytical models—that is, mathematical models expressed by analytically computable equations. The software community widely adopts these models because they let project managers quickly estimate specific project attributes (such as delivery time or effort) or obtain snapshots of the process behavior over time given initial fixed conditions (such as requirements size). Examples include

- COCOMO, to estimate delivery time and effort,
- Rayleigh, to predict and shape the staffing profile, and
- reliability growth models, to estimate the testing required to achieve the desired product quality.

To obtain the required capabilities from an advanced DSS, we must turn to more complex mathematical models, which can’t be analytically solved but require simulation techniques. In this case, calculations indicated by the model’s equations are performed over and over to represent the passage of time. If calculations are performed along the time scale at a fixed interval (for example, every second), we have continuous simulation, as with system dynamics models; if they’re performed only when specific events happen (for example, termination of an activity), we have discrete-event simulation, as with queuing network models.

Although either technique can successfully model almost any process aspect, each has advantages and disadvantages. Discrete-event models highlight the process’s structure, capturing its components and the entities flowing among them. So, they allow efficient representation of a software process structure in terms of activities and exchanged artifacts and tend to be convenient for detailed process analysis. They’re CPU efficient because computation occurs only when relevant events happen, but they can’t represent continuously varying process variables (such as staff availability).

Continuous models, on the other hand, represent a process as a set of equations capturing relationships among dynamic variables of interest (programmer skill, staff availability, final defect density, and so on), although those variables aren’t individually traced through the process. Although these models can easily represent continuously changing variables, they might have difficulty representing process activities. They’re more suitable for strategic long-term analysis than for detailed analysis of a software project.
Because no single approach is well suited to representing all process aspects, the combination of continuous and discrete-event simulation—that is, hybrid simulation—is emerging as a promising approach.\textsuperscript{7,8} For example, Robert Martin and David Raffo present a model that represents software development as a series of discrete steps, executed in a continuously varying project environment.\textsuperscript{7}

**A hybrid modeling approach**

I propose exploiting the advantages of the three traditional modeling methods (analytical models and continuous and discrete-event simulation) by combining them into a hybrid two-level modeling approach.\textsuperscript{8}

At the higher abstraction level, the process is modeled as a discrete-event queuing network—that is, a set of service stations that process customers. A typical example of a queuing network is a set of supermarket checkout counters (service stations) with customers queuing up to pay. A customer arrives at a station to obtain a service, waits in line if the station is serving another customer, and leaves after being served. A queuing network provides a natural way to represent a software process structure, its activities, their interactions, and the exchanged artifacts. The process activities (design, coding, testing, and so on) can be represented as service stations. The circulating artifacts (design documents, code, fault reports, and so on) can be represented as customers moving from one service station to another (for example, a design document moving from the design to the testing activity).

At the lower abstraction level, analytical models and continuous simulation represent the dynamic behavior of the service stations introduced at the higher abstraction level.

I implemented the suggested approach using the Queuing Networks Analysis Package 2.\textsuperscript{15} Although I could have adopted any package, QNAP2 provides convenient language primitives to support the hybrid model’s development.

**The NASA Software Engineering Laboratory process**

As a case study, I used this hybrid approach to model the NASA SEL software process.\textsuperscript{16,17} The model focuses on the main process quality attributes (effort, delivery time, productivity, rework percentage, and product defect density) and numerous subattributes (final product size, process staffing profile, staffing profiles over single activities, defect patterns, and so on).

As figure 1 shows, the SEL software process consists of a series of phases: specification (SP), high-level design (HLD), low-level design (LLD), implementation (IMP), system test (ST), and acceptance test (AT). The final software product is the end result of a series of main artifacts: requirements, specification, high-level design, low-level design, code, system-tested code, and acceptance-tested code.

Although the phases are sequential, their respective activities can run concurrently, given the simultaneous execution of work tasks that generate the main artifacts and rework tasks to fix defects or introduce requirement modifications. So, in addition to the main artifacts, artifacts are also generated and dealt with by tasks aiming at fixing defects (LLD defects reports, code correction reports, and so on) and by tasks that introduce modifications due to requirements instability (such as SP changes and HLD increments).

**Modeling the SEL software process**

My hybrid approach translates the SEL software process into a two-level model. At the higher abstraction level, I model the process as a queuing network (see figure 2 on page 71). A set of service stations, each representing an internal task (for example, development of the main artifact, review, and defect correction), models process activity. In particular, I’ve used two different types of subnetworks to model the development activities (SP, HLD, LLD, and IMP) and the testing activities (ST and AT).

For example, figure 2 highlights the subnetworks that model the HLD and AT activities.

The subnetwork that models the HLD activity has these main service stations:

- The *work* station models the HLD’s development on the basis of the received SP.
- The *external rework* station simulates the activity necessary to modify the HLD on the basis of the received SP changes and increments and produces the corresponding HLD changes and increments.
- The *review* station simulates the review performed on the HLD and changes and increments.
- The *internal rework* station simulates the
activity necessary to fix the HLD on the basis of the received SP corrections reports and HLD defects reports. It then generates the corresponding HLD corrections reports.

Similarly, the subnetwork that models the AT activity has these main service stations:

- The work testing station simulates acceptance testing of the system-tested code.
- The external rework testing station simulates acceptance testing of the system-tested code changes and increments.

In both these networks, the remaining service stations coordinate artifact flow:

- The start station routes the input artifacts toward the appropriate service station
- The release and store stations handle the release of the output artifacts. For example, in the HLD activity, the HLD artifact will be released only after all the defects discovered during the review have been corrected.

At the lower abstraction level, each service station in figure 2 is modeled by a combination of analytical models and continuous simulation. In particular, depending on the specific task that the service station simulates, I've adopted a different set of analytical models, derived from SEL guidelines, to express the corresponding required resources (for example, time and personnel) and performances (such as defect injection during development, defect detection during review, and productivity).

For example, figure 3 shows some implementation details of the HLD work station. This station starts operating when it receives
Figure 2. The higher abstraction level: queuing network of the SEL software process.
the SP artifact (the input customer) and simulates the development of the HLD artifact (the output customer).

As schematized in figure 3, two attributes describe artifacts: size and effort. According to the SEL, the size of the SP and HLD artifacts are expressed in SP and HLD units.6,12 Effort (measured in person-weeks) represents the accumulated effort—that is, the effort that the staff has spent to develop the artifact since the process’s beginning. On the basis of the SP artifact’s attributes, the work station computes the corresponding HLD artifact attributes and simulates the development task’s dynamic behavior in terms of delivery time and required staff.

First, starting from the specification’s size (SP_size), a COCOMO-like estimator calculates the HLD’s size (HLD_size). Because such a quantity might have random deviations in real life, a Gaussian pseudorandom generator transforms the average size into the corresponding random value. Then, two other COCOMO-like estimators compute the delivery time (T) and the development effort (W).

On the basis of T and W, the staff necessary over time to produce the HLD artifact is represented using the Rayleigh model. Here, continuous simulation is adopted. The staff required each week for the development task is obtained by computing at a one-week interval (that is, the simulation interval) the Rayleigh equation:

$$E(t) = W \frac{t}{T^2} e^{-\frac{t^2}{2T^2}}$$

According to the Rayleigh model, the output artifact is released at time T—that is, when the staff level $E(t)$ peaks. So, to produce the HLD artifact, the work station uses only a percentage of the development effort computed through the COCOMO-like effort estimator (that is, the integral up to T, as in figure 3). This percentage ($0.39W$) is then added to the effort of the specification ($W_{SP}$) to obtain the effort attribute of the HLD ($W_{SP} + 0.39W$).

The COCOMO-like estimators used to model the work service stations in the different activities have been derived from SEL data.16,17 In particular, I combined the original SEL COCOMO models with the observed effort distribution across the various SEL process activities.

A decision support tool for the manager

To illustrate the simulation model’s capabilities as a DSS, I’ve used it to study how requirements instability affects a project. I applied the model to reproduce two possible software development scenarios:

- a stable-requirements scenario, wherein a requirements artifact of 1,500 function points is fed to the process model, and
- an unstable-requirements scenario, wherein the initial requirements artifact of 1,500 function points is followed by requirements increments and changes regularly fed into the process over the development time to reproduce a situation in which requirements grow by 20 percent and change by 15 percent.

Table 1 and figures 4 and 5 show the simulation results for the stable-requirements scenario. Table 1 shows the predicted average values for product size, development effort, delivery time, productivity, rework percentage, defects density,
and staff. It also shows the corresponding confidence intervals—that is, the intervals that the model predicts the results will be within in 95 percent of the cases.

Figure 4 shows the staff profile for the whole project, and figure 5 details the staff profile for the IMP activity. Both figures show the average value within its 95 percent confidence interval.

We can now investigate these results’ validity. A simulation model’s validity is usually verified by comparing simulation results quantitatively and qualitatively against real-world data and by demonstrating the model’s ability to reproduce empirically known facts (for example, requirements instability’s effects on a process).

To compare the simulation results against real data, table 1’s SEL data column shows the data obtained with the available SEL estimation models for a software product of 116 KLOC. This comparison’s results are comforting, because SEL data are within (in one case) or very close to the confidence interval the simulator provided. In addition, the staff behavior that figures 4 and 5 illustrate indicates the simulation results’ closeness to real-world SEL data. In both cases, staff profiles behave as expected for an SEL project: The average staff profile for the entire project (figure 4) strongly resembles the trapezoidal shape (compare with figure 6), while the staff profile for a single activity (figure 5) is very similar to a Rayleigh curve (the dotted line in figure 5).

To verify the model’s ability to reproduce empirically known facts, we can turn to the simulation results for the scenario with unstable requirements. Figure 7 illustrates the effects of requirements instability on the staffing profile and compares the profiles for stable and

Table I

<table>
<thead>
<tr>
<th>SW process attribute</th>
<th>Simulation results</th>
<th>SEL data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average values</td>
<td>Confidence interval (95%)</td>
</tr>
<tr>
<td>Final product size</td>
<td>116 KLOC</td>
<td>N/A</td>
</tr>
<tr>
<td>Effort (W)</td>
<td>500 person-weeks</td>
<td>440–560 person-weeks</td>
</tr>
<tr>
<td>Delivery time (T)</td>
<td>78 weeks</td>
<td>75–81 weeks</td>
</tr>
<tr>
<td>Productivity</td>
<td>5.8 LOC/person-hour</td>
<td>4–7.6 LOC/person-hour</td>
</tr>
<tr>
<td>Rework percentage</td>
<td>17%</td>
<td>14–21%</td>
</tr>
<tr>
<td>Defect density</td>
<td>0.9 defects/KLOC</td>
<td>0.6–1.2 defects/KLOC</td>
</tr>
<tr>
<td>Average number of staff</td>
<td>6.5</td>
<td>5.5–7.5</td>
</tr>
</tbody>
</table>

Figure 4. Stable requirements: staffing profile for the whole project.

Figure 5. Stable requirements: staffing profile for the implementation activity.
unstable requirements. The simulator confirms the empirical expectation that requirements instability leads to a substantial increase of effort and delivery time (38 and 60 percent, respectively). It also confirms the expected reduction of process productivity, increased rework percentage, and worsening of the final product quality (defect density). In fact, the rework percentage has more than doubled (+150 percent), productivity has clearly dropped (-21 percent), and the final product's defect density has increased (+66 percent).

In addition, simulation results allow for an interesting comparison with real SEL project behavior. The behavior of the simulator-generated staffing profile is qualitatively very similar to that of the staffing profile (Figure 6) that was measured for a project with similar characteristics (that is, a slightly larger product size, 160 KLOC rather than 140 KLOC, and high requirements instability).

Although the original SEL data show that managers had to replan twice during the project to fit its dynamics (that is, first and second replan to fit the actual data), the simulation model can estimate the process dynamics with only one application. Project managers can simulate different possible unstable scenarios (that is, different kinds of requirements behavior over time) to estimate the corresponding project trajectories. So, managers will not only have accurate predictions in unstable environments but also be able to manage the new requirements over time (that is, to decide when and how to feed them to the project) to design a trajectory that better fits the organizational needs.

This case study demonstrates that the hybrid modeling approach can provide both qualitative and quantitative suggestions on tuning the software process to improve its quality and better meet organizational needs. Combining three traditional modeling methods overcomes some of the limits of the individual approaches (for example, analytical models' limited estimation capabilities). It also provides some clear advantages.

First, it lets you bridge the gap between modelers and project managers. It allows exploiting the graphical clarity of queuing networks and project managers' familiarity with analytical models such as COCOMO and Rayleigh.

Also, it allows quick reuse of knowledge available in organizations. A queuing network, in fact, can be obtained as a direct replica of the process structure often described in organizational standards, and analytical models are commonly adopted within a software company to manage projects.

Finally, it results in flexible process models, easily adaptable to the organization's character-
istics and maturity level and updateable to follow its evolution (as advocated by software process improvement methods such as the Capability Maturity Model\textsuperscript{18} or the Quality Improvement Paradigm\textsuperscript{19}). You can easily change or update the queuing network and its embedded analytical models to address the specific context needs.

Acknowledgments
I completed part of this work as a PhD student in the Computer Science Department of the University of Rome Tor Vergata.

References

About the Author
Paolo Donzelli is the director of the Research and Information and Communication Technologies Division of the Department for Innovation and Technology of the Office of the Prime Minister of Italy. He’s also a visiting senior research scientist in the University of Maryland’s Computer Science Department. His research interests include software process improvement methods, requirements engineering, and dependability modeling and validation. He received his PhD from the University of Rome Tor Vergata. Contact him at the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; donzelli@cs.umd.edu.