Simulation-based Software Process Modeling and Evaluation

Ove Armbrust*, Thomas Berlage†, Thomas Hanne‡, Patrick Lang‡, Jürgen Münch*, Holger Neu*, Stefan Nickel‡, Ioana Rus⌂, Alex Sarishvili‡, Sascha van Stockum†, Andreas Wirsen‡

Corresponding author: Ove Armbrust <armbrust@iese.fraunhofer.de>

* Fraunhofer IESE
Sauerwiesen 6
67661 Kaiserslautern
Germany
{armbrust, muench, neu}@iese.fraunhofer.de

† Fraunhofer FIT
Schloss Birlinghoven
53754 Sankt Augustin
Germany
{berlage, stockum}@fit.fraunhofer.de

‡ Fraunhofer ITWM
Gottlieb-Daimler-Str., Geb. 49
67663 Kaiserslautern
Germany
{hanne, lang, nickel, sarishvili, wirsen}@itwm.fraunhofer.de

⌂ Fraunhofer USA Center for
Experimental Software Engineering
College Park MD 20742-3290
USA
irus@fc-md.umd.edu

Reference:
Ove Armbrust, Thomas Berlage, Thomas Hanne, Patrick Lang, Jürgen Münch, Holger Neu, Stefan Nickel, Ioana Rus, Alex Sarishvili, Sascha van Stockum, and Andreas Wirsen, „Simulation-based Software Process Modeling and Evaluation", In: Handbook of Software Engineering and Knowledge Engineering, Vol. 3: Recent Advances", (S. K. Chang, ed.), World Scientific Publishing Company, pp. 333-364, 2005.
DOI: 10.1142/9789812775245_0012
http://www.worldscientific.com/doi/abs/10.1142/9789812775245_0012

Abstract

Decision support for planning and improving software development projects is a crucial success factor. The special characteristics of software development aggregate these tasks in contrast to the planning of many other processes, such as production processes. Process simulation can be used to support decisions on process alternatives on the basis of existing knowledge. Thereby, new development knowledge can be gained faster and more cost-effectively.
This chapter gives a short introduction to experimental software engineering, describes simulation approaches within that area, and introduces a method for systematically developing discrete-event software process simulation models. Advanced simulation modeling techniques will point out key problems and possible solutions, including the use of visualization techniques for better simulation result interpretation.

Keywords: software process modeling, data analysis, simulation, human resources, decision support, visualization, scheduling, optimization

1 Introduction

Simulation is widely used in many sciences. For example, in molecular chemistry or biology, simulation helps to reduce costs by lowering the number of real experiments needed. In industrial contexts, simulation helps to lower costs, for example, in car construction, by replacing real crash tests with virtual ones.
Experimental software engineering, as a profession that heavily involves experiments with the (cost-intensive) involvement of human subjects, is obviously very interested in using simulation. Here, simulation promises to save cost and time. As a rather young profession, (experimental) software engineering has only recently started to discover and use the benefits of simulation. When implemented, real experiments already yield good results, for example, in choosing a reading technique for inspections in a certain context. Still, the required experiments cost a lot of money, which companies naturally dislike, despite the benefits. But as in other professions, simulation or virtual experimentation can also be of assistance in software engineering.
In this chapter, we will give a short introduction to experimental software engineering, then describe discrete and continuous simulation approaches within that area, and introduce a method for systematically developing discrete-event software process simulation models. Advanced simulation modeling techniques such as the integration of empirical knowledge, the use of knowledge-discovery techniques, modeling human resources, optimization and visualization as well as the topic of model reuse will point out key problems and possible solutions
The rest of the chapter is organizes as follows. Section 2 introduces the underlying discipline of experimental software engineering. Section 3 gives an introduction to simulation approaches. Section 4 describes a method for systematically creating discrete-event simulation models. Section 5.1 explains the integration of empirical knowledge into simulation models, Section 5.2 the use of knowledge discovery techniques for use in simulation modeling. Human resource modeling is described in Section 5.3, visualization issues in Section 5.4, model reuse in Section 5.5, and optimization in Section 5.5. Section 5.6 explains optimization issues. Some conclusions are drawn in Section 6.

2 Experimental Software Engineering

Many efforts have been made in the field of software engineering in order to contribute to one goal: to develop software in a controlled manner so that quality, costs, and time needed can be predicted with high precision. Other goals include the overall reduction of development time and costs, and a general increase in product quality. No single best

approach can be identified to achieve these goals, since software development is human-based and heavily depends on the context (such as organizational constraints, development methods, or personnel characteristics). Following Rombach et al. [1], three different approaches to study the discipline of software engineering can be identified:

- the mathematical or formal methods approach,
- the system building approach,
- and the empirical studies approach.

Regarding the empirical approach, the word "empirical", derived from the Greek "empeirikós" (from "émpeiros" = experienced, skillful), means "drawn from experience". This suggests that data from the real world is analyzed in order to understand the connections and interrelations expressed in the measured data. This is an important issue in the empirical approach: the data must be measured somehow. Measuring empirical data is not trivial, on the contrary: many wrong decisions are based on incorrectly measured or interpreted data [2].

The aim of the empirical approach consists of two parts: observing the real world correctly (with correct measurement and interpretation of the data measured), and coming to the right conclusions from the data measured. The latter concerns selection and usage of techniques, methods, and tools for software development as well as activities like resource allocation, staffing, and scheduling.

In order to improve software development performance, experimenting with changes to the current situation or alternatives is necessary. Generally, this is achieved by determining the current situation, then changing some parameters and evaluating the results. Evaluating a new technique across several projects, however, is difficult due to the different contexts. Additionally, whatever is proven by one experiment must not necessarily be true for other contexts [3].

Unfortunately, experimenting is a rather expensive way of gathering knowledge about a certain technique, method, or tool. The costs can be divided into two main parts:

First, the experiment itself must be conducted, that is, the experimental setup must be determined, the experiment needs to be prepared and carried out, the data must be analyzed, and the results evaluated. These are the obvious costs, which can be measured easily. Basili et al. [4] estimate the costs for conducting an experiment to be at least $500 per subject per day.

The second part consists of risks (of delaying the delivery of a product, or using an immature technology). If a new technique is tested in a controlled experiment or a case study, for example, before it is used during development, this ensures that no immature or unsuitable technique is used in real life. Since software development is human-based, human subjects are needed for the experiment. During the experiment, they cannot carry out their normal tasks, which may result in delays, especially in time-critical projects.

Competitors not carrying out experiments may be able to deliver earlier. These costs are not easily calculated. Still, in terms of product quality, delivery time, and person hours, using immature or unsuitable techniques is, in most cases, more expensive than experimenting.

3 Simulation and experimental software engineering

Simulation is increasingly being used in software process modeling and analysis. After concentrating on static process modeling for a long time, software engineers are beginning to realize the benefits of modeling dynamic behavior and simulating process performance. These benefits can be seen, for instance, in the areas of strategic management of software development, operational process improvement, or training situations [5]. Depending on the respective purpose, the simulation focus ranges from lifecycle modeling (for long-term strategic simulation) to small parts of development, for example, a code inspection (for training purposes).

Three main benefits of software process simulation can be identified: cost, time, and knowledge improvements. Cost improvements originate from the fact that conventional experiments are very costly. The people needed as experimental subjects are usually professional developers. This makes every experimentation hour expensive, since the subjects get paid while not immediately contributing to the company's earnings. Conducting a simulation instead of a real experiment saves these costs.

Time benefits can be expected from the fact that simulations can be run at (almost) any desired speed. While an experiment with a new project management technique may take months, the simulation may be sped up almost arbitrarily by simply having simulation time pass faster than real time. On the other hand, simulation time may be slowed down arbitrarily. This might be useful in a software engineering context when too much data is accumulated in too little time, and therefore cannot be analyzed properly. The time benefits result in shorter experimental cycles and accelerate learning about processes and technologies.

Knowledge benefits can be expected in two areas: new knowledge about the effects of technologies in different areas (e.g., performing replications in different contexts, such as having less experienced subjects) contributes to a better understanding of the software development process. Scenario-oriented training, for example, improves individual workforce capabilities. Using a simulation environment such as the one introduced in [6] enables the trainee to experience immediate feedback to his decisions: their consequences do not occur months after the decision, but minutes. This way, the complex feedback loops can be better understood and mistakes can be made without endangering everyday business [7].

There are several simulation techniques available. Which one to choose, depends on the model purpose. Two major approaches can be distinguished: the continuous approach and the discrete approach. A hybrid approach combines both, to overcome their specific disadvantages. In addition to that, there are several state- and rule-based approaches, as well as queuing models [8]. We will introduce the continuous approach and the discrete approach successively.

3.1 Continuous simulation modeling

In the early 1970s, the Club of Rome started an initiative to study the future of human activity on our planet [9]. A research group at the Massachusetts Institute of Technology (MIT) developed a societal model of the world. After a smaller-scale dynamic urban model, this was the first large continuous simulation model: the World Model [10].
Today, most continuous models are based on differential equations and/or iterations, which take several input variables for calculation and, in turn, supply output variables. The model itself consists of nodes connected through variables. The nodes may be instantaneous or non-instantaneous functions. Instantaneous functions present their output at the same time the input is available. Non-instantaneous functions take some time for their output to change after the input changes.
This approach of a network of functions allows simulating real processes continuously. An analogy would be the construction of mathematical functions (add, subtract, multiply, integrate) with analog components like resistors, spools, or condensers. Before computers became as powerful as they are today, this approach was the only way to solve that kind of equations in a reasonable time. Due to the continuous nature of the "solver", the result could be measured instantly.
Of course, simulating this continuous system on a computer is not possible due to the digital technology used. To cope with this, the state of the system is computed at very short intervals, thereby forming a sufficiently correct illusion of continuity. This iterative recalculating makes continuous models simulated on digital systems grow complex very quickly.
In the software engineering context, continuous simulation is used primarily for large-scale views of processes, like management of a complete development project, or strategic enterprise management. Dynamic modeling enables us to model feedback loops, which are very numerous and complex in software projects.

3.2 Discrete simulation modeling

The discrete approach shows parallels to the clocked operations car manufacturers use in their production, with cars moving through the factory. The basic assumption is that the modeled system changes its state only at discrete moments of time, as opposed to the continuous model. This way, every discrete state of the model is characterized by a vector containing all variables, and each step corresponds to a change in the vector.
To give an example, let us consider a production line at a car manufacturer. The production is described by a finite number of working steps: each work unit has to be completed in a certain fixed or stochastically described amount of time. When that time is over, the car-to-be is moved to the next position, where another work unit is applied, or waits in a buffer until the next position is available. This way, the car moves through the complete factory in discrete steps.
Simulating this behavior is easy with the discrete approach. Each time a move is completed, a snapshot is taken of all production units. In this snapshot, the state of all work units and products (cars) is recorded. At the next snapshot, all cars have moved to the next position. The effects of the variable values at time t are considered at time $t+1$. The real time that passes between two snapshots or simulation steps can be arbitrary, usually the next snapshot of all variables is calculated and then the simulation assigns the respective values. Since the time needed in the factory for completion of a production step is known (or determined from a stochastic distribution), the model describes reality appropriately. A finer time grid is certainly possible: instead of viewing every clock step as one simulation step, the arrival at a work position and the departure can be used, thereby capturing work and transport time independently.
The discrete approach is used in software engineering as well. One important area is experimental software engineering, e.g., concerning inspections. Here, a discrete simulation can be used to describe the product flow, which characterizes typical software development activities. Possible simulation steps might be the start and completion of activities and lags, together with special events like (late) design changes. This enables discrete models to represent queues.

4 Systematically creating discrete-event software process simulation models

This Section describes a method for developing discrete simulation models. For lack of space, only the most important steps are outlined with short descriptions. For a more comprehensive description of the method, please refer to [11] and [12]. The description in this chapter considers the development of a new simulation model without reusing or incorporating existing components. If reuse is considered (either incorporating existing

components or developing for reuse), then the method has to be changed to address possible reuse of elements (see Section 5.4).

4.1 Identification and specification of simulator requirements

During the requirements activity, the purpose and the usage of the model have to be defined. The questions that the model will have to answer are determined and so is the data that will be needed to answer these questions. Sub-activities of the requirements specification are:

4.1.1 Definition of the Goals, Questions, and the necessary Metrics

A goal/question/metrics (GQM) [13], [14] based approach for defining the goal and the needed measures seems appropriate. GQM can also be used to define and start an initial measurement program if needed.
The purpose, scope and level of detail for the model are described by the goal. The questions that the model should help to answer are formulated next. Afterwards, parameters (metrics) of the model (outputs) have to be defined. Once their values are known, the questions of interest can be answered in principle. Then those model input values have to be defined that are necessary for determining the output values. The input values should not be considered as final and non-changing after the requirements phase; during the analysis phase, they usually change.

4.1.2 Definition of usage scenarios

Define scenarios ("use cases") for using the model. For example, for answering the question: "How does the effectiveness of inspections affect the cost and schedule of the project?", a corresponding scenario would be: "All input parameters are kept constant and the parameter inspection effectiveness is given x different values between a minimum and a maximum value. The simulator is executed until a certain value for the number of defects per KLOC is achieved, and the values for cost and duration are examined for each of the cases." For traceability purposes, scenarios should be tracked to the questions they answer.

4.1.3 Test case development

Test cases can be developed in the requirements phase. They help to verify and validate the model and the resulting simulation.

4.1.4 Requirements validation

The customer (who can be a real customer, or plainly the stakeholder of the simulated process) has to be involved in this activity and agree with the content of the resulting model specification document. Changes can be made, but they have to be documented.

4.2 Process analysis and specification

The specification and analysis of the process that is to be modeled is one of the most important activities during the development of a simulation model.
We divide process analysis and specification into four sub-activities, as shown in Figure 4.3-1: analysis and creation of a static process model (group a), creation of the influence diagram for describing the relationships between parameters of the process (b), collection and analysis of empirical data for deriving the quantitative relationships (c), and quantification of the relationships (d). Figure 4.3-1 sketches the product flow of this activity, i.e., it describes which artifacts (document symbol) are used or created in each step (circle symbol).

4.3 Analysis and creation of a static process model

The software process to be modeled needs to be understood and documented. This requires that the representations (abstractions) of the process should be sufficiently intuitive for being understood by the customer and for constituting a communication vehicle between modeler and customer. These representations lead to a common definition and understanding of the object of modeling (i.e., the software process), and a refinement of the problem to be modeled (initially formulated in the requirements specification activity). The created process model describes the artifacts used, the processes or activities performed, and the roles and tools involved. It shows which activities transform which artifacts and how the information flows through the process.

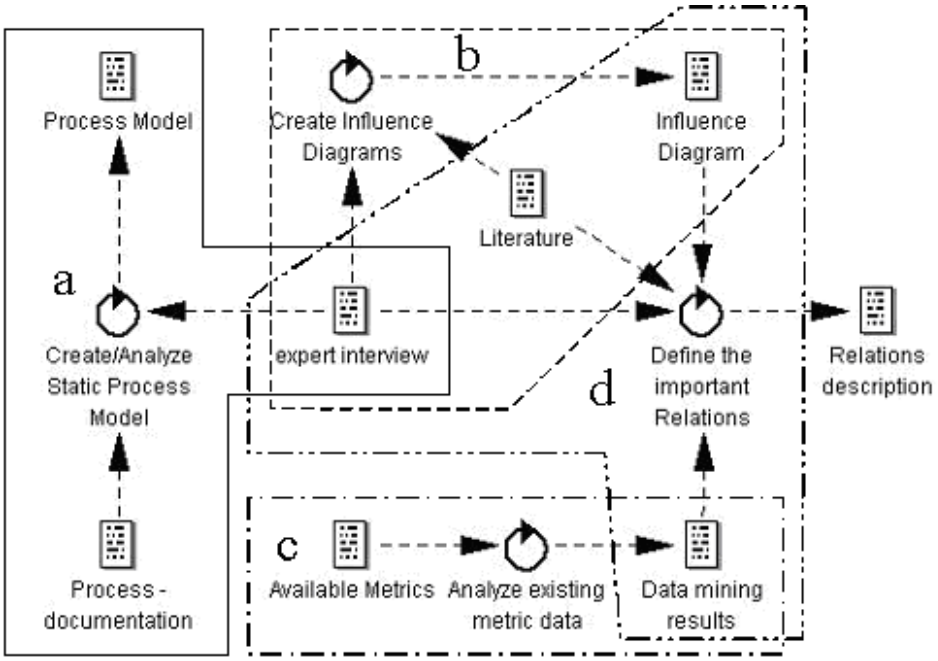


Figure 4.3-1: Process analysis and specification

4.4 Creation of the influence diagram for describing the relationships between parameters of the process

For documenting the relationships between process parameters, we use influence diagrams. In the influence diagram, we start capturing the factors that influence the process. Influence factors are typically factors that change the result or behavior of the process. If these factors rise, they change the related factors. This is displayed by + and – at the arrows that depict the relationship.

When we draw the influence diagram, we should have in mind the inputs and outputs identified in the requirements phase. These inputs, and especially the outputs, have to be captured in the influence diagrams. Figure 4.4-1 presents an excerpt of a static process model and a corresponding influence diagram.

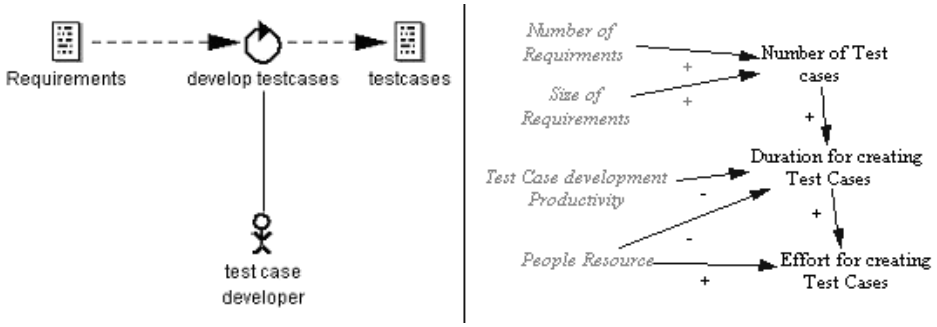


Figure 4.4-1: Static process model (left) and influence diagram (right)

4.5 Collection and analysis of empirical data for deriving the quantitative relationships

Depending on existing metrics, it is possible to use this metric data to calibrate the model. If a lack of data is identified, collecting data in the target organization during model development can fill this.

4.6 Quantification of the relationships

This is probably the hardest part of the analysis. Here, we have to quantify the relationships. Parameter types can be distinguished as follows:
Calibration parameters: for calibrating the model according to the organization, such as productivity values, learning, skills, and number of developers.
Project-specific input: for representing a specific project, such as number of test cases, modules, and size of the tasks.
Variable parameters: these are the parameters that are changed in order to record output variable behavior. In general, the variable parameters can be the same as the calibration parameters. The scenarios from the requirements or new scenarios determine the variable parameters during the model lifecycle.
Mathematical quantification is also done in this step. If historical metric data is available, it should be utilized, otherwise interviews with customers, experts, or bibliographic sources have to be used. The outputs of the process analysis and specification phase are:

1. models (static, influence diagrams, and relationships) of the software development process,
2. parameters that have to be simulated,
3. measures (metrics) needed to be received from the real process, and

4. a description of all the assumptions and decisions that are made during the analysis. The latter is useful for documenting the model for later maintenance and evolution.

Finally, several verification and validation steps must be performed here:

- The static model has to be validated against the requirements (to make sure it is within the scope, and also that it is complete for the goal stated in the requirements), and validated against the real process.
- The parameters in the influence diagram must be checked against the metrics for consistency, and the input and output variables of the requirements specification must also be checked.
- The relationships must be checked with respect to the influence diagram for completeness and consistency. All the factors of the influence diagram that influence the result have to be represented, for instance, by equations determining the relationships.

4.7 Model design

During this activity, the modeler develops the design of the model, which is independent of the implementation environment. The design is divided into different levels of detail, the high-level design and the detailed design.

4.7.1 High level design

Here, the surrounding infrastructure and the basic mechanisms describing how the input and output data is managed and represented are defined. Usually, the design comprises components like a database or a spreadsheet, a visualization component, and the simulation model itself, together with the information flows between them. The high level design is the latest point for deciding which type of simulation model should be created: system dynamics, discrete-event, or something else.

4.7.2 Detailed design

Here, the low level design of the simulation model is created. The designer has to decide on which activities have to be modeled (granularity, requirements, relationships), what the items are (granularity, requirements, relationships), and which attributes the items have. Additionally, the designer has to define the item flow in the model.

4.8 Implementation

During implementation, all the information and the design decisions are transferred into a simulation model. This activity in the development process heavily depends on the simulation tool or language used and is very similar to the implementation of a conventional software product.

4.9 Validation and verification

The model needs to be checked for its suitability regarding the purpose and the problems it should address, and to see whether it sufficiently reflects reality (here the customer has to be involved).
Throughout the simulator development process, verification and validation should be performed after each activity. Furthermore, traceability between different products created during simulator development should be maintained, to simplify model maintenance.

5 Advanced simulation modeling

In this Section, we present some typical problems with simulation modeling, and possible solutions. First, we introduce a possible way to increase model validity by integrating empirical results. As an example, acquiring influence factors by knowledge extraction techniques is explained more thoroughly. Human resource modeling helps with considering personal and social aspects, and visualization techniques assist in interpreting simulation results. Reuse and optimization techniques lower the effort during modeling and when performing the modeled process.

5.1 Increasing model validity: Integration of empirical studies

Since one of the goals of software process simulation modeling is to gain knowledge about the real world, the model should comply with two requirements. First, the amount of information included in the model should be as small as possible to reduce model complexity. Second, sufficient information to answer the questions posed should be included. The second requirement is a hard one, i.e., if it is not met, the modeling goals cannot be reached. For example, a model that is to be used for calculating project costs needs information on effort for project activities; otherwise, no reliable prediction for project costs can be expected. The first requirement directly influences model complexity. The more information about the real world is included, the more complex the model

becomes, with all known consequences for maintainability, usability, and understandability.

Many software process simulation models rely mainly on expert knowledge. Another way is the integration of empirical knowledge [15]. This can be done in three different ways.

1. Empirical knowledge is used for the development and calibration of simulation models.
2. Results from process simulations are used for planning, designing, and analyzing real experiments.
3. Process simulation and real experiments are performed in parallel (i.e., online simulations).

Concerning model development and calibration, empirical data can be integrated virtually anywhere in the model. In fact, the creation of the influence diagrams explained in Section 4.4 can be supported quite well by empirical data. It may not always be completely clear which factor has which effect, but there are mathematical solutions that can assist with these problems, e.g., sophisticated data mining methods (see Section 5.2 for an example). When an initial model exists, calibrating the model using more empirical data, e.g., from replications of the experiment/study used for creating the influence diagram, is also possible. Another use of empirical knowledge might be to extend the model to aspects of reality that have not been modeled yet.

The integration of empirical knowledge also works in a reciprocal manner: results from simulations can be used to scope real experiments. A simulation can predict where great changes in the result variables can be expected, and the real experiment can be designed to pay extra attention to these. This sort of decision support may also be used as a first stage for testing new technologies: only candidates that prevail in the simulation are examined more closely. This obviously requires a sufficiently reliable simulation model.

Online simulations combine real experiments, usually for a small part of a more comprehensive process, with simulations taking care of the rest of the process. This way, not only the results from the experiment can be evaluated, but also their effects in a larger context. This provides an inexpensive scale-up for experiments. Another application would be training. Using simulation technology, e.g., the SimSE environment described in [6], the trainee can experience immediate feedback to decisions, without the lengthy delays and dangers that would occur in a real project.

This is only a short overview on the use of empirical knowledge in simulations. More detailed information can be found in [15].

5.2 Acquiring influence factors: knowledge extraction in the software development process via additive nonlinear regression

Building a simulation model for the software development process requires the determination of input-output relationships for the different sub-processes. The basic information needed for building a simulation model is included in the qualitative process models, which provide a general understanding concerning the chronology of tasks, the flow, and the qualitative dependencies of objects. The simulation model can be built by following the control and flow diagram step by step. In this way, working units (WU) inside the simulation model with their related inputs and outputs, i.e. items, stuff, etc., are determined. Inside a working unit, the relationships between certain variables then have to be quantified by mathematical functions or logical rules. An influence diagram qualitatively describing the relationships in a working unit generally distinguishes between three types of variables:

- WU input variables, which do not depend on other variables;
- WU output variables, which do not affect other WU variables;
- Internal WU variables, which explain other WU variables and are also explained by them.

Based on the influence diagram, step by step one chooses each of the internal and output variables as the explained variable, and their corresponding predecessors as explaining variables. The related input-output relationships then have to be determined as a logical rule or a mathematical function. Possible methods for quantifying the qualitatively known input-output relationships are expert interviews, pragmatic models, stochastic analysis, and knowledge discovery techniques. The choice of the actual technique depends, of course, on the data or information available.

If sufficient measurement data is provided, an application of knowledge discovery techniques is possible. In a first step, these methods will be used for the quantification of the input-output relationships. Then, based on the identified mappings, new insight and knowledge for parts of the considered process can be extracted. Thus, the above mentioned knowledge discovery techniques can also be used for validating the simulation model. Based on simulation data, a simplified input-output mapping covering the complex dynamical dependencies of variables inside a WU could be determined. Analyzing this mapping, for example, by relevance measures, makes it possible to extract additional process knowledge.

Due to saturation effects, the influence of some of the input variables, like *skills of programmers or inspectors,* obviously is nonlinear. Therefore, it seems quite reasonable to use a nonlinear regression model. Here, we will consider an additive nonlinear regression model (AN). AN models are reasonable generalizations of classical linear models, since AN models conserve the interpretability property of linear models and, simultaneously,

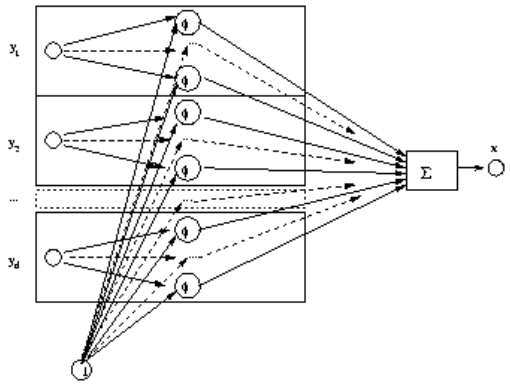
are able to reproduce certain nonlinearities in the data. It is also possible to interpret their partial derivatives. The importance of the partial derivatives lies in the relevance and sensitivity analysis.

The additive nonlinear regression function can be approximated by different methods. We believe that one appropriate technique are specially structured (block-wise) neural networks for the estimation of the AN model. In [16], [17], it was shown that fully connected neural networks are able to approximate arbitrary continuous functions with arbitrary accuracy. Furthermore, in [18], it was proven that neural networks are able to approximate the derivatives of regression functions. This result was assigned in [19] to block-wise neural networks as estimators for nonlinear additive, twice continuously differentiable regression functions and their derivatives. The network function consists of input and output weights for each unit in the hidden layer. These weights have to be estimated from given measurement data. One well-known method for performing this task is to minimize the mean squared error over a training set. The performance of the network is measured by the prediction mean squared error, which is estimated by cross validation [20].

Taking into account the special structure of the composite function of the AN model, which results from summing up *d* functions of mutually different real variables, a feed forward network with one hidden layer and with blockwise input to hidden layer connections seems to be convenient for its approximation. Figure 5.2-1 shows the topology and the mathematical description of a blocked neural network with *d* inputs and one output. In particular, each neuron in the hidden layer accepts only one variable as input apart from the constant bias node. The trained neural network will be implemented in the model at the considered working unit for the considered variable of the influence diagram, to calculate the output for a given input during the simulation runs.

Now it will be shown that relevance measures calculated on the identified network function can be used to verify the correctness of modeling assumptions and further provide a deeper understanding of the dependencies in the software development process. The problem, which often occurs especially in modeling a software development process is that measurement data is not available for all input variables. The granularity of the model determines the minimal amount of measurement data needed for quantification, since all input and output variables of the underlying qualitative model should be used in this step. In case of missing measurement data for one or more variables, one has to make

Blocked Neural Network Topology:



Network Function:

$$x_i = f(y_{1i}, ..., y_{di}, \Theta)$$

$$= \sum_{i=1}^{H(1)} v_i \phi(y_{1t} w_i + b_i) + ...$$

$$+ \sum_{i=H(d-1)+1}^{H(d)} v_i \phi(y_{dt} w_i + b_i)$$

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

H(j) - H(j-1), ∀ j=2,...,d: number of neurons in block *j*

H(d) : total number of neurons in the hidden layer

Θ : vector of all neural network parameters

w_j , j=1…d: weights from the input to the hidden layer

v_j , j=1…d: weights from the hidden layer to the output layer

b_j , j=1…d: biases

Figure 5.2-1: Topology and Network Function of a blocked neural network

further assumptions or skip these variables. Relevance measures in this case help to determine the impact of each input variable with respect to the output variable.

By considering the validation results and the corresponding relevance measure, one can easily verify whether the estimated functional dependencies describe the input-output relationship in a sufficient manner, if the impact of a skipped variable is too large. If a variable has only small relevance over its whole measurement range, it is redundant and thus can be skipped. Thus, based on measurement data, a relevance measure can also be used to validate the qualitative description of the dependencies given in the relevance diagrams, since the inputs for a node in the diagram are assumed to be non-redundant. Also, the size of the impact of every input variable for a given data set is available, and might give a manager new insight into the software development process under construction.

To define the relevance measures mathematically, we consider the following regression problem: $x_i=M(y_{1i},...,y_{di}) + \varepsilon_i$, $i \in [1,...,N]$, where ε_i is independent identical $N(0,\sigma^2)$ distributed noise, the function $M:\Upsilon^d \to \Upsilon$ fulfills certain mathematical conditions. We now approximate the true regression function M with a neural network approximator:

$$\hat{x}_i = f_{nn}(y_{1i},...,y_{di},\hat{\Theta}) + \varepsilon_i \quad \text{and} \quad \hat{\Theta} = \arg\min_{\Theta \in \Theta_H} \left(\sum_{i=1}^{N} \left(x_i - f_{nn}\left(y_{1i},...,y_{di},\Theta\right)\right)^2 \right)$$

where Θ_H is a compact subset of the parameter (weight) space. The most common measure of relevance is the average derivative (AD), since the average change in $\hat{x}$ for a very small perturbation $\delta y_j \to 0$ in the independent variables y_j is simply given by $AD(y_j) = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \hat{x}_i}{\partial y_{ji}}$, where $\hat{x}$ is the vector of estimated regression outputs. Additional important relevance measures are the coefficient of variation or the average elasticity.

In the case of a general nonlinear differentiable regression model, the computation of the relevance measures is also possible; however, in order to guarantee the interpretability of the results, further structural properties have to be fulfilled. For AN models, such as the blocked neural networks considered here, these properties hold, and therefore, the impacts of the single input variables can be estimated. In order to compute the relevance measures, the first partial derivatives of the trained network function with respect to each explaining variable have to be determined. For the sigmoid neuron activation function of the network, the partial derivatives (PD) are calculated via:

$$\frac{\partial f(y_1,...,y_d,\Theta)}{\partial y_j} = \sum_{i=H(j-1)+1}^{H(j)} v_i \left(1 - \tanh^2\left(b_i + w_i y_j\right)\right) w_i, \quad j = 1,...,d,$$

where *H(j)* is defined as shown in Figure 5.2-1. Obviously, the partial derivative explicitly only depends on the considered input, the influences of the other variables are comprised in the network parameters Θ. The PDs can already be used to analyze the impact of the input variables as shown in the following example:
the aim of the example was to identify the regression function between the input variables 'effort', 'inspected lines of code', and 'total lines of code', and the output variable 'major defects' on given measurement data. The left plot in Figure 5.2-2 shows the used blocked neural network cross validation performance. The solid line in this plot is the measured number of detected major defects, and the dashed line is the estimated number of the same variable. The estimated absolute mean error is ±0.8, i.e., one can expect that the neural network predicts in mean ±0.8 major defects per document incorrectly.

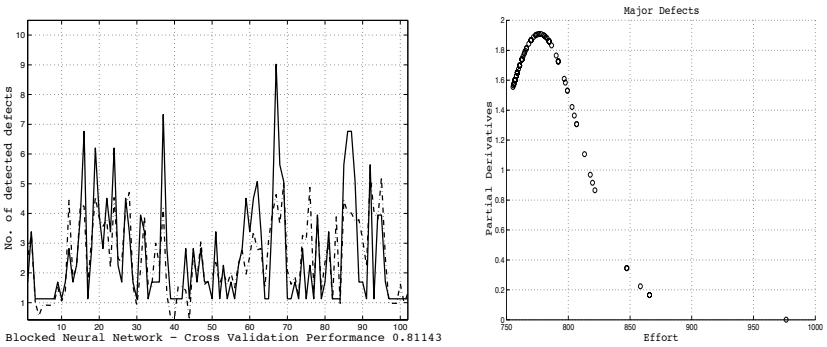


Figure 5.2-2: Performance Plot and PD Plot

The right plot in Figure 5.2-2 shows the PD plot for the variable 'effort'. One notices that for an actual effort in the range of 750 to 850 units, increasing the effort leads to a significant increase in the number of found defects. Obviously, the largest benefit for an increase in the working effort in terms of additional found major defects is obtained around 775 units. An increase in effort for documents with an actual value greater than 850 units will only lead to a slight increase in the number of found defects. Thus, based on the PD plot for the variable effort, and the known costs for each effort unit, a software manager can approximately determine the effort he would like to spend in the inspection.
The distribution of the mean relevances of the input variables 'effort', 'inspected lines of code', and 'total lines of code' to the output variable 'detected number of major defects' is respectively: AD=[34 17 6]. This means that on average, the variable effort has the largest impact on the considered output.
Thus, AN models, especially the presented blocked neural network, are suitable for the quantification of the qualitative models based on measurement data and for an analysis of the determined mathematical equations in order to get a deeper insight into the input-output relationship by relevance analysis.

5.3 Considering personal aspects: human resource modeling

5.3.1 Skills

When dealing with a detailed modeling of complex human-based processes, such as software development, there are various problems related to an adequate representation of persons (developers) involved in the processes. Until now, there are rather few established human-based models of software development processes. Occasionally, models appear to deal with human factors in an ad-hoc way. Possibly, this is because of a lack of empirical data available for validating models. On the other hand, empirically proven results from psychology and other fields may be employed for reasonable (but still simple) models, for considering the most important effects related to software developers.

As a most common influence on software developers, experiences/skills and their changes during the process (learning/training) are considered. This is mostly done in an aggregated fashion, e.g., in the simulation models by [21], [22], and [23]. Examples of modeling these effects on an individual basis can be found, e.g., in [24], [25], and [26]. This allows presenting a more detailed view on individual skills and learning, thus also on the increase of knowledge in software development processes.

A typical assumption of a detailed modeling of humans is that specific skills determine personal productivities (i.e., the speed of work, see [27]), and the quality of work. This quality may be measured, for instance, by the number of defects produced during coding, or the number of defects found during inspection and testing. Since productivity-oriented and quality-oriented skills do not need to be the same for a person, or evolve synchronously, these two types of skill values may be distinguished in a model. Moreover, skills may be distinguished according to the type of work (e.g., coding, testing) and the domain of the application.

For ease of use, skill values may be calibrated on a non-dimensional [0,1] interval. A skill value of about 0.5 is typical for an average developer, while a value near 1 characterizes an experienced or high performance developer. Multiplied with given values for maximum quality of work or maximum productivity (corresponding to skill values of 1), a person's actual defect (production, detection) rate and productivity can be determined.

5.3.2 Learning

One frequent approach to learning is that of a learning curve that describes the increase of productivities over time, or accumulates production (see, e.g., [28]). For the case of software developers' skill values, a suitable model for the learning mechanism is that of the

logistic growth model (see [29]). According to the pure logistic model, a minimum value of 0 and a maximum value of 1 is assumed with a continuous monotonic growth of the corresponding skill output variable (see Figure 5.3-1).
Assuming small changes of time, Δt, the corresponding changes of the skill value can be described by

$$\Delta skill := f \; \Delta t \; skill \; (1 - skill)$$

where f is a learning factor specific to the particular skill, which depends on the institutional environment and personal factors not explicitly considered in the simulation model. Δt is the time used for an activity that involves the particular skill, thus the time elapsed since the last update of that skill. If an activity involves several skills (e.g., phase-specific and domain-specific skills), all of them should be updated after finishing the activity. The factor f determines the steepness of the learning curve and thus the time needed to get from a low skill value to a high skill value. Therefore, it reflects the average learning capability of a software developing organization and depends, for instance, on the degree of re-usability of former artifacts.

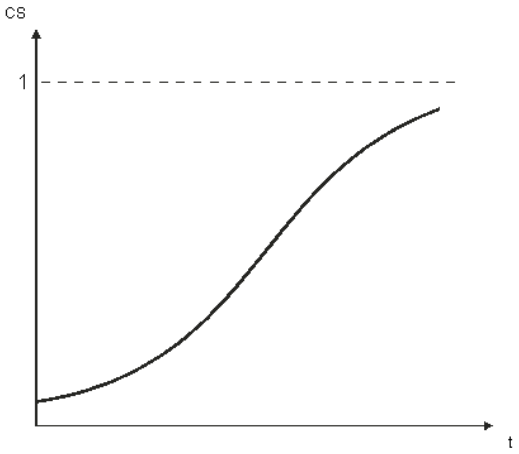


Figure 5.3-1: Logistic growth of skills

5.3.3 Time pressure, fatigue and boredom

As another main effect on personal productivities, we discuss a time pressure model for capturing some motivational effects related to fatigue and boredom. Considering a standard level of workload and time pressure, it is assumed that an increase of time pressure first leads to an increase in productivity. For instance, people work in a more concen-

trated way, social activities are reduced, or voluntary overtime is done. This, however, can usually only be expected for a limited period of time and to some specific extent. Excessive time pressure leads to stress and negative motivation of personnel, coupled with a decrease of work performance to below the standard level.

On the other hand, insufficient time pressure results in underperformance of the staff, who may use extra time for extending social and other activities. Boredom due to low working requirements may decrease motivation and thus, productivity.

These effects are expressed in Figure 5.3-2, which shows a time pressure factor, measured on a non-dimensional nonnegative scale, and depending on the deviation x of planned time t* to elapsed time t for the current activity, i.e., x=t-t*. If the activity is behind schedule (t>t*), then first positive and later negative effects take place. If the fulfillment of the plan is easy (t<t*), boredom shows some negative effects. Such a relationship can be expressed by superimposing two logistic functions. The resulting function can be calibrated with assumptions on the standard level (tp=1) for a deviation x=0, assumptions on the activation levels for extreme boredom, extreme backlog, and peak performance, the steepness of stepping inside these effects, and the mutual deferment of the effects. In Figure 5.3-2, for instance, the boredom level is approx. 0.75, the overload level is approx. 0.63, the peak level 1.13, the steepness toward boredom level is smaller than towards overload level, and the deferment between these transitions is about 12 on the x-axis (which may be measured in working days). Note that the basic characteristics of the above function correspond to general assumptions on the psychological relationships between activation and performance according to ([30]).

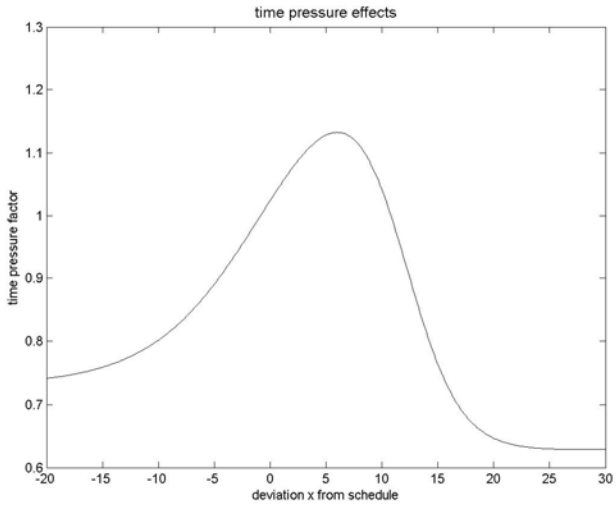


Figure 5.3-2: Plan deviation and time pressure

The above approaches are just a few examples on how human impacts can be integrated into detailed simulation models of software development processes. A realistic modeling in that area, however, requires significant efforts in getting empirical information about the specific situation to be modeled. Convincing simulation models of human factors in software development processes are uncommon up to now, possibly exactly because of this lack of empirical data. Nonetheless, appropriate planning of the processes requires such information, and promises significant savings and other improvements (see Section 5.4 for more information).

5.4 Interpreting simulation results: Visualization

5.4.1 Requirements

Because software engineering consists of human and organizational processes, models of these processes will always be simplifications, and the nature and detail of valid models may vary considerably. Therefore, one of the requirements in simulating these models is that the structure of the model must be made explicit to the user (white box modeling), otherwise the resulting data cannot be used to influence decisions in the real world.
A second requirement is that, because the models are still complex and have many parameters, a single view will not be sufficient to fully comprehend the simulation results.

Instead, an interactive simulation cockpit is needed that can provide different views and perspectives and support decision-making as a result.

The most important variables in software engineering are resource usage (costs), time and quality. Decisions need to be justified as optimizing this set of result parameters by influencing input parameters such as scheduling (e.g., allocation of tasks to programmers) or sampling strategies (number and type of inspection meetings).

Finally, the simulation cockpit needs to support what-if analyses, i.e., varying input parameters while observing the effects as directly as possible. For cognitive reasons, it is highly beneficial if parameter variations result in animated changes in the result variables, as these directly map to active control mechanisms of the human brain.

There are three levels of detail in the visualization of simulation results.

1. The *aggregate view*, summarizing the dependency of result variables on the input variables in terms of charts and diagrams. This view allows the user to quickly identify optimal points within the value range. However, as the simulation is often only qualitatively correct, the user does not understand the reason why the model behaves in this way.
2. The *resource view*, summarizing the way resources are spent and thus giving an impression of where resources are wasted, and where they can be spent more effectively. This view level is described in more detail below. However, this view does not show the structure of the model or the individual rules employed.
3. The *component view* that shows all details of the models, of the strategies and rules employed, and of the individual simulation runs. This level can be used to follow every detail, but is not able to convey action strategies to the user. This level is usually supported by the simulation tool (Figure 5.4-1).

The visualization is coupled to the simulation tool via a database interface that is able to read out an event-based description of simulation runs. In principle, this interface can be coupled directly to the simulation, or it can read a batch of results from a set of simulation runs. The former approach allows the user to vary all parameters of the model, but the user has to wait for the simulation to finish. The latter approach provides a much quicker interactive response to changing the input parameters and observing the results, but all parameter combinations of possible interest need to be calculated in advance, which is expensive in time and space.

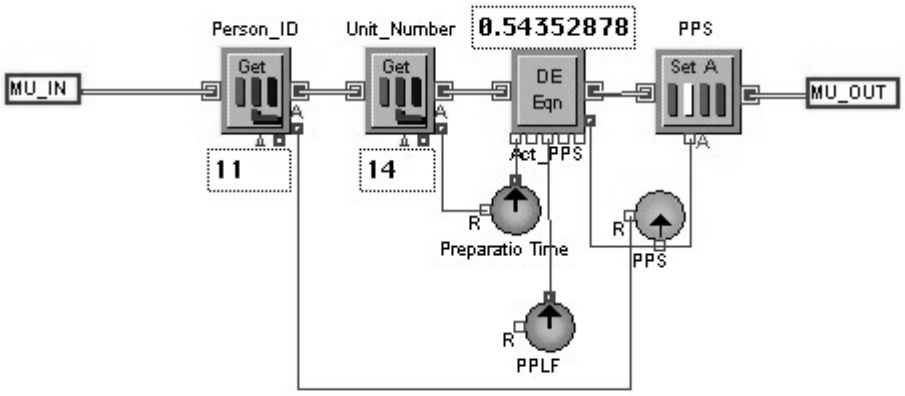


Figure 5.4-1. Detail view of the simulation model.

5.4.2 Resource view

The resource view (shown in Figure 5.4-2) is a presentation of the three result variables *time* (along the x axis), *relative cost* (along the y axis) and *quality* (shown in colors). The x-axis encodes the simulation time of actual work spent (e.g., hours or days). The y-axis shows the resources employed from two perspectives: the manpower partitioned into persons (bottom), and the software partitioned into modules (top). The height of each resource should be roughly proportional to the cost of the resource, which can be the skill (and thus the salary level) of a programmer, or the complexity (and thus the labor intensity) of a module.

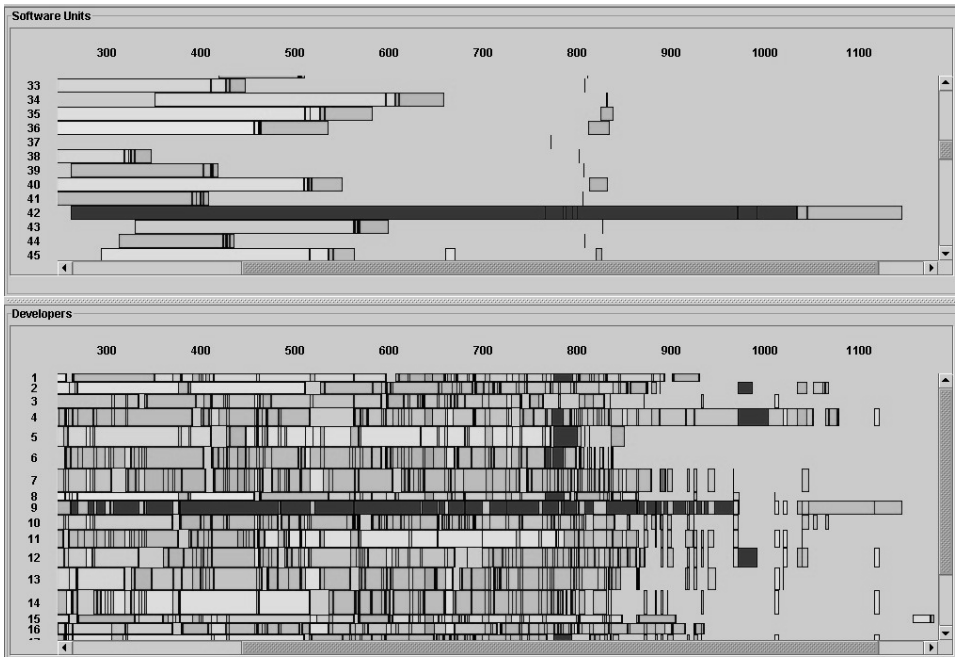


Figure 5.4-1: Resource view of a simulation result

Every rectangle in the resource view shows a resource-consuming activity (such as programming or testing) with its start and ending time coming from the simulation run. As the height of the rectangle is roughly proportional to the relative cost, the area of the rectangle is roughly proportional to the total cost. While the filled areas signal costs, the amount of empty areas visualizes the effective use of time towards a delivery date.

It has to be noted that every activity occurs twice in the resource view: once in the bottom area attached to the person conducting the activity, and once in the top area attached to the object worked on. This dual appearance of each activity can be followed by highlighting an activity in one area and seeing it selected in the other area.

As a third dimension, quality variables (such as number of errors remaining) are visualized through color-coding the rectangles. In this way, the user can quickly assess the quality of the results and where it has been achieved (or not achieved). Homogeneity can also be observed.

Cognitively, the resource view employs human abilities to perceive relative sizes of objects presented together, to compare lengths in the same dimension, to distinguish between about 16 shades of intensity, to distinguish between foreground an background

(colored vs. gray or white) and to evaluate the relative frequency or frequency distribution of line patterns.
As a result, the resource view shows the important quality criteria for software engineering activities: the resource consumption, the effective use of available time, and the quality of the end products. These variables are not simply summarized into a single number, but from a discrete simulation, particular causes can be spotted, such as critical modules, the effectiveness of scheduling, or time wasted on inspecting good quality modules.

5.4.3 Studying cause-effect relationships

A straightforward application for the simulation cockpit is to compare the results of two different simulation runs with different input parameters (e.g., with a different inspection or scheduling strategy). The two resource views can be shown side by side (in an appropriate scaling), and their relative effectiveness can be compared both globally (the total size of rectangles, their color density and their homogeneity) and locally (for outlier events such as large or colored or empty areas as well as for high-frequency, high-activity regions).
It is highly desirable to also study the effects of continuous input parameter manipulation on the resource view of the simulation. For that purpose, a representative set of input parameters can be pre-simulated, and each result can be stored as a data set of temporally ordered activities. The input values used can then be presented to the user as a slider or similar widget. While the user manipulates the widget and changes the input value, the corresponding resource view is shown. If the resource view display is fast enough, the result will be perceived as an animated transition.
Such an animated process allows the user to easily perceive not only the total cost, but also the nature of the change associated with the change of the input value. For example, the user would easily be able to spot reduced amount gaps, higher quality or quality being achieved earlier.

5.5 Minimizing effort within modeling: Reuse

5.5.1 Introduction

In the last years, many software process simulation models have been developed, using different simulation techniques and languages. The majority of these simulation models were developed from scratch, and development was ad-hoc. Reuse of products is unsystematic and rarely employed. Reuse in simulation modeling usually occurred at the

knowledge level, and was mostly accomplished by taking advantage of the modeler's own experience.
Reuse in software engineering resulted in reduction of development and maintenance time and effort, and in improved software quality. We expect that for process modeling and simulation, the advantages of reuse will be similar.

5.5.2 Reuse in process modeling simulation: What, When and How?

What, when, and how to reuse elements of a simulation model during model development is not obvious or simple. There are many questions to be answered. We identified four main questions that can lead to structured solutions.
1. What can be reused?
We identified different elements in the development process that can be subject to reuse:
a) Requirements, environment, and scenarios for using simulation.
Often organizations have the same or similar problems regarding duration, effort, and quality, because these are usually the three main factors influencing the success of software development.
b) Static process models or their components.
There are not so many different ways to organize certain base practices in software development, e.g., inspections, coding, or testing. For these components, predefined static descriptions can be created that could be transformed into the actual process description. A combination of these parts can lead to a process model that meets the scope of the problem under study.
c) Influence diagrams or their components.
Influence diagrams qualitatively describe the relationships between process parameters. They usually include the relations between measurable variables (like size) or statistical values (productivity of a team or person), and the values that the model should predict, like effort or quality (e.g., defects per 1000 lines of code).
d) Relations between process parameters.
The quantitative descriptions of relationships modeled in influence diagrams are represented as mathematical expressions that could be reused.
e) Model design (patterns) [31].
For specific problems during modeling, such as how to represent activities, item flow, or resources, a pattern can help a novice modeler. These patterns are not necessarily specific for solving technical problems for only one simulation technique, but they can also describe proposed solutions for higher-level static or dynamic model descriptions.
f) Executable model elements or components of executable models.

This type of information or models would correspond to a reuse of classes in software development. Models or components must be instantiated and parameterized with the input values for the specific problem.
Model elements in discrete event models could be compared to classes in software development. In fact, the simulation blocks of a discrete event model may be organized in libraries similar to class libraries. These simulation blocks can be combined to build more complex blocks for specific tasks. The blocks have input and output interfaces, and can form a hierarchy of sophisticated blocks.
Components are larger parts of simulation models, which are described with input / output interfaces and parameters. Depending on the definition, which has to be developed for the context of simulation models in general, discrete event model components can contain complete parts of a process (for example, an inspection). For components of models, we have to define the interfaces to enable reuse in other models. Also, a description is necessary to describe the properties and the interfaces of the component.
g) Knowledge about model development, deployment, operation, and evolution methodology.
Not only pieces of artifacts can be reused, but also the knowledge acquired by the modelers as well as the methods they applied and that proved to be successful. In software development, the knowledge about the process is described with process models. If we document the process of developing a simulation model, then the description and the experience stored in the documentation can be reused. In the next step, we should identify typical modeling patterns and provide information on how to implement these in different modeling approaches. By describing these, we can support the process of simulation modeling.
2. How to facilitate reuse?
How objects can be reused is really dependent on the type of the object. For example, executable simulation models or parts of them can be reused for other models that use the same modeling approach and the same language and environment. Obviously, complete simulation models can be reused if a similar software development process has to be modeled for simulation. If the software process that has to be modeled for simulation differs, the reuse of a complete simulation model is not appropriate.
For finding potential objects or parts of objects that we can reuse, we should follow a similar development method that produces the same artifacts during simulation model development. That makes it easier to identify reusable objects. Reuse can be done on different levels:

- by modularizing artifacts that are created during development, or
- by modularizing the model itself, or parts of it.

If the model itself will be based on modules, it is important to define standardized interfaces between the modules.

3. When to look for elements for reuse?
We look for elements for reuse all the time, even if we do not do it knowingly [32]. The farther we are in the development process, the more concrete reuse objects we are looking for. The objects created more specific to (and dependent of) the simulation language or technique can be reused later in the development process. In the early stages of the development of a software process simulation model, the reuse of knowledge is more important than the explicit usage of modules developed in earlier projects.
4. How to develop models for and with reuse?
This question is difficult to answer because often, simulation models are developed without using a well-defined method. At first, we have to adhere to a method for developing a simulation model with certain activities and artifacts that will be created. The first objects for reuse are artifacts and the knowledge captured in these artifacts. One possible method could start with a requirements definition for the model. The analysis has a high potential for reuse of knowledge and experience because here, the modeler has to understand the process and capture the important measures to fulfill the requirements. In the design and following implementation of the model, the objects for reuse become more and more concrete. During design, architectural patterns [33] can help to structure the model. The reuse of components can also be planned in the later design phase. More specific model elements or modeling patterns can support the implementation phase.

5.5.3 Potential benefits of reuse in simulation modeling

In [32], the authors describe the potentials and pitfalls for reuse in general in the software engineering domain. The benefits of reuse in software engineering are mainly a reduction of effort or gain in productivity for a particular project by reusing existing assets. With high quality assets, a gain in quality of the end product can be achieved, because more effort was spent in the development of the asset or the asset was more thoroughly debugged. Reusable assets also can reduce the development schedule and reduce the time-to-market.
The assets to be reused must first be developed and organized in libraries. Depending on the amount of information that is necessary to identify an asset for reuse, the likelihood of finding a matching asset decreases with an increasing amount of information. Also, the overhead necessary for developing, identifying, selecting, and adapting reusable assets and its costs are often not taken into account for a software reuse program. A poorly planned reuse program can cost more than it saves. The introduction of reuse also requires an organizational change.

In the context of simulation models, usually only isolated small teams or individuals are involved in the development of the model. Therefore, only the problem of finding a matching element for reuse can be considered to be an obstacle.

5.6 Optimization of software development models

5.6.1 Introduction

For simulations models, the problem frequently arises of how specific parameters of the model should be chosen (see, e.g., [34]). While the excerpt from reality to be represented specifies some of the parameters, there are others that can be chosen freely, and that influence the results and performance of the simulation model. Therefore, we are interested in finding optimal values for these parameters with respect to the model results. Mostly, such values are determined by systematic experiments or some trial and error approach. Only a few simulation tools are equipped with built-in optimization procedures for automatic determination of optimal parameters.

In the case of software development, even in detailed simulation models, one major requirement of modern project management is usually neglected: an efficient assignment of tasks to persons and the scheduling of tasks. For instance, in a model [35] based on the simulation tool Extend [36], the assignment of tasks to persons (e.g., items to be coded, items to be inspected) is done by a simple rule, namely in a first come-first serve (FCFS) fashion. This means that items are treated in a given arbitrary order and the developers being available next become their authors. The assignment of coding and other software development tasks to people is relevant, since the duration of the project, i.e., the makespan, depends on this assignment and, in general, the time required for performing a task depends on a person's productivity, which is, for instance, influenced by his or her experience.

While assignment and scheduling usually refer to non-explicit parameters of a simulation model, there are others that require explicit determination. For instance, the quality of the process may depend on the extent of quality assuring techniques. Typical questions are, for instance: Should inspections be applied? Which documents should be inspected? How many persons should inspect a document?

Because of a dissatisfactory FCFS task assignment and the missing possibility of optimizing other planning parameters within a pure simulation model, we may reformulate a software development model with an associated combined assignment and parameter optimization problem.

5.6.2 Multiobjective optimization

Usually, in practice, there are several objectives to be considered at the same time. For a software development process (see, e.g., [21]), the following three objectives are frequently considered as most important from a practical point of view: a) the quality of the product measured by the eventual overall number of defects, *td*, of the documents produced during a project, b) the duration, *du*, of the project (its makespan), and c) the costs or total effort, *tc*, of the project.

While the costs and number of defects can be determined once the model parameters (including persons assigned to specific tasks) are fixed, the calculation of the project duration requires the explicit generation of a schedule. In particular, dependencies on tasks have to be fixed. This, for instance, requires a commitment on which kinds of tasks may pre-empt (be pre-empted by) which other kinds of tasks. In the case of software development, we may assume that inspections tasks are done in an "interrupt fashion" such that waiting times for a synchronized start of the team do not occur. Interrupts for the inspection of other authors' documents are done "in between", where it is assumed that associated inspectors are immediately available when an item is completely coded. This assumption can be justified because inspection times are comparably small and, in practice, people usually have some alternative tasks for filling "waiting times".

The specific times of each task are calculated by constructing a schedule for all tasks to be done, i.e., a complete schedule for the software development project, which comprises several weeks or months. Based on this, the project duration, *du*, can simply be calculated by the maximum finishing time of the tasks.

The considered multiobjective optimization problem can then be formulated as

$$\text{"min"}\ (td(x), du(x), tc(x)) \qquad (1)$$

for the decision variables x, which comprise the assignment of developers to tasks (coding, inspections, testing, ...), and the specification of other parameters (e.g., size of inspection teams). This problem is restricted by various constraints, which refer to the assignment (e.g., a person may not work as an author and inspector of the same document), define the temporal logic (working times for tasks), and specify the quality of results (effects of activities on the number of defects).

"min" in Equation 1 means that each of the objectives should be minimized. Usually, the objectives can be considered to be conflicting, such that there exists no solution that optimizes all objectives at the same time. As a solution in the mathematical sense, generally the set of efficient solutions is considered. An efficient solution is an alternative for which there does not exist another one that is better in at least one objective without be-

ing weaker in any other objective, or formally: a multiobjective optimization problem is defined by

$$\text{“min”}\ f(x) \qquad (2)$$

with $x \in A$ (set of feasible solutions) and $f: \mathbb{R}^n \to \mathbb{R}^q$ being a vector-valued objective function. For x,y∈A, the Pareto relation “≤” is defined by

$$f(x) \leq f(y) := f_i(x) \leq f_i(y) \ \text{for all}\ i \in \{1,\ldots,q\} \ \text{and}$$

$$f_i(x) < f_i(y) \ \text{for at least one}\ i \in \{1,\ldots,q\}. \qquad (3)$$

The set of efficient (or Pareto-optimal) alternatives is then defined by:

$$E(A,f) := \{ x \in A: \text{there does not exist } y \in A: f(y) \leq f(x)\}. \qquad (4)$$

See [37] for more details on efficient sets.
Once the efficient set is determined, or a number of (almost) efficient solutions is calculated, then further methods may be applied to elicit preference information from the decision maker (the project manager), and for calculating some kind of compromise solution (see [38], [39], [40], and [41]). For instance, some interactive decision support may be applied for that purpose.

Numerical Results of Optimizing a Simulation Model

Evolutionary algorithms are one of the powerful techniques that can be used to optimize simulation models. These approaches are also especially useful for multiobjective optimization (see [42]). We have applied a multiobjective evolutionary algorithm for approximating the efficient set of a SD scheduling problem (see [41]). In Figure 5.6-1, we have shown the results of applying an EA approach to planning inspections and scheduling staff with the above properties to a given test problem. For the populations of some selected generations, the values for two objective functions are represented in the 2-dimensional space for each combination of two objectives.

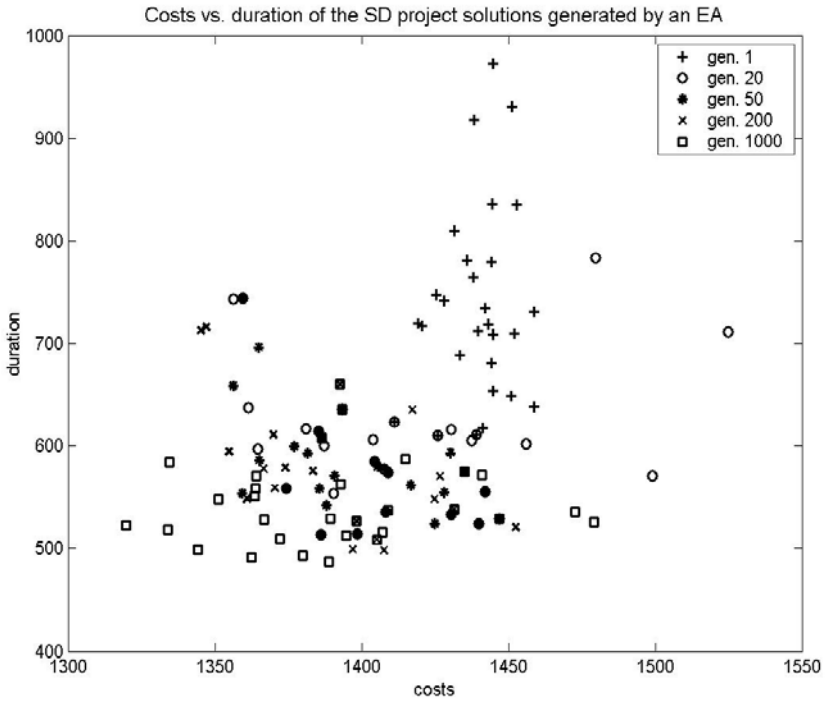


Figure 5.6-1: Costs and durations in solutions of several generations

Note that the populations move from the upper right to the lower left region, which indicates an improvement of both objectives during time. The preserved bandwidth of objective values during the evolution reflects the fact that the objectives are conflicting. Therefore, an approximation of the efficient set consists of a diverse set of solutions.
As can be seen, there is continuing progress towards minimization of the objectives and diversity of the solutions. For instance, considering the population of generation 1000 in Figure 5.6-1, there are several solutions that are better with respect to the duration and the costs, although there is no significant progress with respect to the best values in these objectives. Altogether, a more diverse population provides a better fundament for decision-making. Of course, it is always possible (if desired) to decrease this scope, e.g., by applying aspiration levels for some or all of the objectives. On the other hand, it is possible to get a denser representation of the efficient set by increasing the population size, which, however, increases to computational effort.

Let us note that the computational requirements of optimizing a simulation model may be rather high. In our example, the execution of the multiobjective evolutionary algorithm with a population size of 30 and an evolution of 1000 generations requires about 6 hours on an 850 MHz computer. In that case, about 90% of the running time is required for evaluating the fitness function, i.e., calculating project schedules. On the other hand, the benefits of optimizing the schedules are significant. Depending on the characteristics of the specific project, on average, between 5% and 30% of improvements can be obtained in each of the objectives compared with a simple FCFS solution.

6 Conclusions

This chapter presented the idea of discrete-event simulation in software process modeling. After an overview of experimental software engineering, the possibilities of integrating simulation into the experimental context were explored. We further presented a method for systematically creating discrete-event software process simulation models. Finally, we explained situations that typically occur during modeling and when using the simulation model, and presented ideas such as the integration of empirical knowledge, the use of knowledge-discovery techniques, modeling human resources, optimization and visualization as well as the topic of model reuse, to cope with these problems.
This short presentation does not claim to be exhaustive. It is intended to give an overview of simulation in software engineering, especially with respect to software process modeling. Determining the effects of processes for constructive purposes, as opposed to purely analytical intentions, is gaining importance within the research community. With respect to this, hybrid approaches look promising by combining the advantages of discrete-event and continuous simulation [43], [44], [45].

7 References

[1] H. D. Rombach, V. R. Basili, and R. W. Selby (Ed.), Experimental Software Engineering Issues: A Critical Assessment and Future Directions. International Workshop, Dagstuhl Castle, Germany, Lecture Notes in Computer Science, Springer Verlag, 1993.
[2] P. B. Ladkin, Report on the Accident to Airbus A320-211 Aircraft in Warsaw, http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html, 1994.
[3] M. Joseph, Software Engineering: Theory, Experiment, Practice or Performance, Research Report CS-RR-117, University of Warwick, UK, 1988.
[4] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz, The Empirical Investigation of Perspective-Based Reading. Empirical Software Engineering 2, 1996.
[5] M. I. Kellner, R. J. Madachay, and D. M. Raffo, Software Process Simulation Modeling: Why? What? How? Journal of Systems and Software 2-3, 1999.
[6] E. O. Navarro, SimSE -- An Educational Software Engineering Simulation Environment, http://www.ics.uci.edu/~emilyo/SimSE/main.html, 2002.
[7] J. Münch, and H. D. R. I. Rombach, Creating an Advanced Software Engineering Laboratory by Combining Empirical Studies with Process Simulation, Proceedings of the International Workshop on Software Process Simulation and Modeling (ProSim), Portland, Oregon, USA, 2003,.
[8] R. L. Glass (Ed.), The Journal of Systems and Software - Special Issue on Process Simulation Modeling, Elsevier Science Inc., 1999.
[9] D. H. Meadows, D. L. Meadows, J. Randers, and W. W. I. Behrens, The Limits to Growth, 1972.
[10] J. W. Forrester, The Beginning of System Dynamics, Banquet Talk at the International Meeting of the System Dynamics Society, Stuttgart, Germany, 1989.
[11] O. Armbrust, T. Berlage, T. Hanne, P. Lang, J. Münch, H. Neu, S. R. I. Nickel, A. Sarishvili, S. v. Stockum, and A. Wirsen, The SEV Method for the Simulation-based Evaluation and Improvement of Software Development Processes, Fraunhofer Institut for Experimental Software Engineering (IESE), Kaiserslautern, Germany, 2003.
[12] I. Rus, H. Neu, and J. Münch, A Systematic Methodology for Developing Discrete Event Simulation Models of Software Development Processes, 2003.
[13] V. R. Basili, and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering 6, 1984.
[14] L. C. Briand, C. Differding, and H. D. Rombach, Practical Guidelines for Measurement-based Process Improvement. Software Process: Improvement and Practice 1996.
[15] O. Armbrust, Using Empirical Knowledge for Software Process Simulation: A Practical Example, Diploma Thesis, University of Kaiserslautern, 2003.
[16] K. Funahashi, On the Approximate Realization of Continuous Mappings by Neural

Networks. Neural Networks 3, 1989.
[17] K. Hornik, M. Stinchcombe, and H. White, Multilayer Feedforward Networks are Universal Approximators. IEEE Neural Networks 5, 1989.
[18] K. Hornik, M. Stinchcombe, and H. White, Universal Approximation of an Unknown Mapping and its Derivatives Using Multilayer Feedforward Networks. IEEE Neural Networks 1990.
[19] A. Sarishvili, Neural Network Based Lag Selection for Multivariate Time Series, University of Kaiserslautern, Germany, 2002.
[20] B. Efron, and R. J. Tibshirani, An Introduction to the Bootstrap Chapman & Hall, New York, USA, 1993.
[21] T. Abdel-Hamid, and S. E. Madnick, Software Project Dynamics, Prentice Hall, Englewood Cliffs, NJ, 1991.
[22] J. D. Sterman, Business Dynamics - Systems Thinking and Modeling for a Complex World, McGraw-Hill, 2000.
[23] I. Rus, Modeling the Impact on Cost and Schedule of Software Quality Engineering Practices, Arizona State University, Tempe, Arizona, USA, 1998.
[24] N. Hanakawa, S. Morisaki, and K. Matsumoto, A Learning Curve Based Simulation Model for Software Development, Proceedings of the 20th International Conference on Software Engineering (ICSE), Kyoto, Japan, 1998, pp. 350-359.
[25] N. Hanakawa, K. Matsumoto, and K. Torii, A Knowledge-Based Software Process Simulation Model. Annals of Software Engineering 1-4, 2002.
[26] A. M. Christie, and M. J. Staley, Organization and Social Simulation of a Software Requirements Development Process. Software Process: Improvement and Practice 2000.
[27] L. F. Johnson, On Measuring Programmer Team Productivity, Proceedings of the Canadian Conference On Electrical And Computer Engineering (CCECE), Waterloo, Ontario, Canada, 1998,.
[28] H. A. Kanter, and T. J. Muscarello, Learning (Experience) Curve Theory: A Tool for the Systems Development and Software Professional. Technical Report, http://cobolreport.com/columnists/howard&tom/, 2000.
[29] M. I. Jordan, Why the Logistic Function? A Tutorial Discussion on Probabilities and Neural Networks, MIT Computational Cognitive Science Report 9503, 1995.
[30] A. Welford, On the Nature of Skill (D. Legge, Ed.), pp. 21-32, Harmondsworth, 1970.
[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1995.
[32] H. Mili, A. Mili, S. Yacoub, and E. Addy, Reuse-Based Software Engineering: Techniques, Organizations, and Measurement, John Wiley & Sons, 2001.
[33] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, John Wiley & Sons, 1996.
[34] A. Gosvi, Simulation-based Optimization - Parametric Optimization Techniques and Reinforcement Learning, Kluwer, 2003.
[35] T. Hanne, and S. Nickel, Scheduling in Software Development using Multi-objective Evolutionary Algorithms, Proceedings of the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA), Nottingham, UK, 2003, pp. 438-

465.
[36] D. Krahl, The Extend Simulation Environment, Proceedings of the Winter Simulation Conference (WSC), Arlington, Virginia, USA, 2001, pp. 214-225.
[37] T. Gal, On Efficient Sets in Vector Maximum Problems - A Brief Survey. European Journal of Operations Research 1986.
[38] M. Zeleny, Multiple Criteria Decision Making, McGraw-Hill, 1982.
[39] R. E. Steuer, Multiple Criteria Optimization: Theory, Computation, and Application, John Wiley & Sons, New York, 1986.
[40] P. Vincke, Multicriteria Decision-Aid, John Wiley & Sons, Chichester, UK, 1992.
[41] T. Hanne, Intelligent Strategies for Meta Multiple Criteria Decision Making, Kluwer Academic Publishers, 2001.
[42] T. Hanne, Global Multi-objective Optimization using Evolutionary Algorithms. Journal on Heuristics 3, 2000.
[43] P. Donzelli, and G. Iazeolla, Hybrid Simulation Modeling of the Software Process. Journal of Systems and Software 3, 2001.
[44] R. H. Martin, and D. A. Raffo, A Model of the Software Development Process Using both Continuous and Discrete Models. Software Process: Improvement and Practice 2-3, 2000.
[45] R. H. Martin, and D. A. Raffo, Application of a Hybrid Process Simulation Model to a Software Development Project. Journal of Systems and Software 3, 2001.