Chapter 5
Simulation Methods

Mark Müller and Dietmar Pfahl

Abstract This chapter aims to raise awareness about the usefulness and importance of simulation in support of software engineering. Simulation is applied in many critical engineering areas and enables one to address issues before they become problems. Simulation – in particular process simulation – is a state of the art technology to analyze process behaviour, risks and complex systems with their inherent uncertainties. Simulation provides insights into the designs of development processes and projects before significant time and cost has been invested, and can be of great benefit in support of training. The systematic combination of simulation methods with empirical research has the potential for becoming a powerful tool in applied software engineering research. The creation of virtual software engineering laboratories helps to allocate available resources of both industry and academia more effectively.

1. Simulation in the Context of Software Engineering

This chapter aims to raise awareness about the usefulness and importance of simulation in support of software engineering. Simulation is a standard technology in many engineering disciplines and has been successfully applied in manufacturing, economics, biology, and social science. Why can simulation enhance traditional software engineering, too? Simulation models are means to analyze the behaviour of complex processes. In the software process literature, according to our understanding, there is a general agreement that people who understand the static process (i.e., process activities, artefacts, resources, roles, and their relationships), and have data, still have difficulties to anticipate the actual process behaviour. This is due to the inherent (dynamic) complexity of software development processes. Software processes can contain iterations, such as rework loops associated with correction of defects. This can lead to delays which may range from minutes to years. As a consequence it is almost impossible for human (mental) analysis to predict the outcome.

Traditionally, process analysis in software engineering research uses static process descriptions like flow charts. This approach does not shed much light on

F. Shull et al. (eds.), *Guide to Advanced Empirical Software Engineering*.

the behaviour of a process over time. Therefore, the usual way to analyze process behaviour is to perform the actual process in a case study and observe the results. This is a very costly way to perform process analysis, because it involves the active participation of engineers. Furthermore, results from a particular case study cannot necessarily be generalized to other contexts. Another way of analyzing processes is to simulate them. Simulation models help to clarify assumptions – often referred to as mental models, on how a process works. They visualize and quantify the implicit mental models about the causes that govern the observed process behaviour and thus support understanding, analysis, prediction, and decision-support.

Simulation models are like virtual laboratories where hypotheses about observed problems can be tested, and corrective policies can be experimented with before they are implemented in the real system. Experience from applications in other fields than software engineering indicates that significant benefits can be drawn from introducing the use of simulation for management decision support. Furthermore, systematic experimentation with simulation models and the integration of simulation-based experiments with empirical research (i.e., case studies and controlled experiments) can support the building of a software development theory (Rus et al., 2002). Simulation-based virtual software engineering laboratories (Münch et al., 2003, 2005) can help focus experimentation in both industry and academia for this purpose, while saving effort by avoiding experiments in real-world settings that have little chances of generating significant new knowledge.

In practice, process simulation models are frequently used to support project planning and estimation. In a competitive world, accurate predictions of cost, quality and schedule provide a significant advantage. For example, if cost estimates are too high, bids are lost, if too low, organizations find themselves in a difficult financial situation. In this context, simulation is a risk management method. It offers not only estimates of cost, but also estimates of cost uncertainty. Simulation also allows for detailed analysis of process costs (Activity Based Costing).

Simulation is effective only if both the model, and the data used to drive the model, accurately reflect the real world. If quantitative output is expected, a simulation can only be executed if it is supplied with quantitative expert estimates or measurement data. Simulation may use industry data or results of quantitative experiments. In order to limit data collection effort, the simulation modeller has to focus on key variables, such as the percentage of design documents which pass or fail review. Thus, as a side effect, simulation modelling supports the focusing of measurement programs on relevant factors of an engineering process.

This chapter is structured as follows: Section 2 explains how simulation models are developed. Section 3 summarizes the variety of application areas and provides references to relevant publications. Sections 4 and 5 describe the simulation techniques and tools used in software engineering. Section 6 provides a simulation reference model which helps to design process simulation models. Section 7 covers practical aspects of simulation modelling. Finally, the chapter concludes with an outlook for trends in future simulation modelling research.

2. The Process of Simulation Modelling in Software Engineering

This chapter provides an overview of the design and implementation of simulation models. Additional information about process simulation paradigms and general introductions can also be found in (Banks et al., 2000; Cellier, 1991; Law and Kelton, 1999). Detailed descriptions of process simulation modelling methods specialized to instances of the event-driven and continuous simulation modelling paradigms can be found in (Rus et al., 2003) and (Pfahl and Ruhe, 2002), respectively.

Any process simulation modelling process consists of at least five steps (cf. Fig. 1):

1. Formulation of the Problem Statement (modelling goal)
2. Specification of the Reference Behaviour (based on observation or hypothetical)
3. Identification of Model Concepts (physical processes, information flows, decision rules)
4. Implementation of Executable Model (formal, executable representation)
5. Model Experimentation

The starting point of any simulation modelling project is the identification and explicit formulation of a *problem statement*. The problem statement defines the modelling goal and helps to focus the modelling activities. In particular, it determines

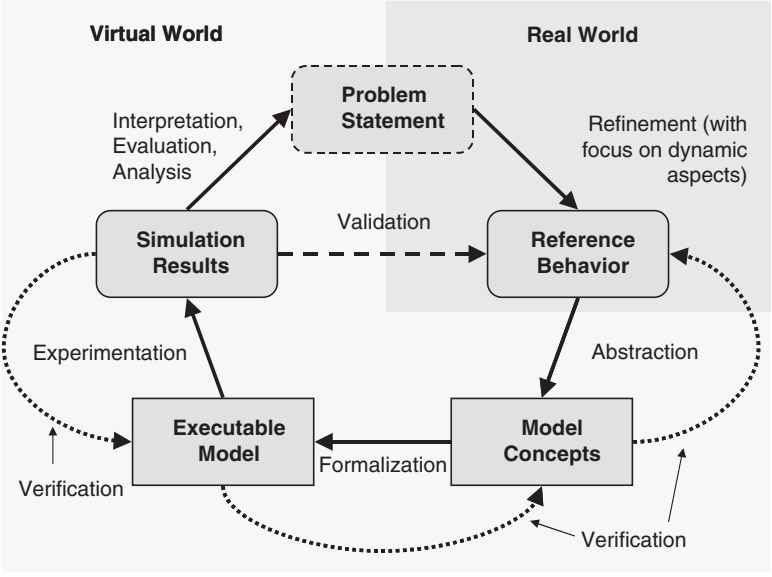

Fig. 1 Iterative process of simulation modelling

the model purpose and scope. For software process simulation models, Kellner et al. (1999) propose the following categories for model purpose and scope:

1. Purpose:
   (a) strategic management
   (b) planning, control and operational management
   (c) process improvement and technology adoption
   (d) understanding
   (e) training and learning
2. Scope:
   (a) a portion of the life cycle (e.g. design phase, code inspection, some or all of testing, requirements management)
   (b) a development project (e.g. single product development life cycle)
   (c) multiple, concurrent projects (e.g., across a department or division)
   (d) long-term product evolution (e.g. multiple, successive releases of a single product)
   (e) long-term organization (e.g., strategic organizational considerations spanning successive releases of multiple products over a substantial time period)

In order to make the problem statement suitable for simulation-based problem-solving, it is helpful to specify the *reference behaviour*. Reference behaviour captures the dynamic (i.e., time-dependent) variation of key attributes of real-world entities. The reference behaviour can be both observed problematic behaviour (e.g., of quality, effort, or cost), which are to be analyzed and improved, and/or a desired behaviour that is to be achieved. The importance of the reference behaviour for the modelling process is twofold. Firstly, it helps identify important model (output) parameters and thus further focuses the subsequent modelling steps. Secondly, it is a crucial input to model validation because it allows for comparing simulation results with observed (or desired) behaviour.

The next step is the definition of *model concepts*, which entail:

1. Existing process, quality, and resource models
2. Implicit or explicit decision rules
3. Typical observed behaviour patterns
4. Organizational information flows
5. Policies

Typically, model concepts can be in the form of quantitative or qualitative models, which are abstractions of behaviours observed in reality. They capture implicit and tacit expert knowledge and are formalized as rules. Usually, in this step, domain experts play a crucial role not only because they often have knowledge that cannot be found in documents or data bases alone, but also because they can help distinguish relevant real-world information from what is irrelevant for the problem under study.

After the definition of model concepts the model is implemented in the simulation tool. Consistent with the modelling technique and tool chosen, all the information, knowledge and experience represented by the model concepts has to be transformed into a computer executable language. The result is an *executable model*. Technical

simulation modelling expertise is crucial in the transformation of model concepts into the formal model representation which eventually will be executed on a computer.

The last step is model calibration and experimentation with the executable model, producing *simulation results*. Simulation experiments are performed to understand the system's behaviour. Experimentation goes hand in hand with model calibration. Model calibration refers to the adjustment of simulation model parameters until the model output corresponds to real word data. Model calibration can be done based on expert estimates or through parameter fitting based on historic data. The calibration step is important in order to ensure that the model accurately reflects real-world behaviour and is required to build confidence in simulation results. After a model is calibrated, simulation experiments are performed to understand observed behaviour, to evaluate planning alternatives, or to explore improvement opportunities. At this stage, iteration is likely in model execution and modification as variables and model structures are changed and the simulation model results are compared against each other. Thus, experimentation not only provides simulation results, but also validates the simulation model. Guidance on how to design simulation experiments in general can be found in (Banks et al., 2000) and (Law and Kelton, 1999), and specifically for software processes in (Wakeland et al., 2003).

Like software development projects, simulation modelling involves verification and validation activities. In short, verification can be seen as an activity that ensures that the model fits its intended purpose, while validation can be seen as the activity that ensures that the model appropriately reflects the real-world behaviour. Verification and validation are continuing activities throughout the modelling and simulation life cycle. They help

1. To produce simulation models that represent system behaviour closely enough to be used as a substitute for the actual system when conducting experiments
2. To increase the credibility of simulation models to a level that makes them acceptable for managers and other decision makers

Verification activities check the internal correctness or appropriateness of a simulation model, i.e. they ensure that the model was constructed in the right way. In particular, verification checks whether the transformation steps defined by the simulation modelling process have been conducted correctly. For example, verification ensures that the identified model concepts have properly been implemented in the executable model. For verification activities, expert knowledge on the simulation modelling technique is a major requirement. To some extent, verification is supported by simulation modelling tools. For example, the consistency of units in model equations can be automatically checked by a tool.

Validation activities check the external correctness or appropriateness of a simulation model, i.e. they try to find out whether the right model (with regards to its purpose or application) was constructed. In particular, validation checks whether the model represents the structural and behavioural properties of the real system correctly (appropriately). For example, simulation results can be used to check the robustness or sensitivity of model behaviour for extreme values of input data. Even though

validation can be partly supported by simulation modelling tools, expert knowledge about the real world system is needed to interpret the range of results obtained.

The simulation literature offers several proposals for verification and validation of simulation models (Balci, 2003; Banks et al., 2000; Barlas, 1989; Forrester and Senge, 1980; Law and Kelton, 1999; Sargent, 2003). For example, Balci (2003) proposes more than 30 different verification and validation techniques, classified into informal, static, dynamic, and formal. However, full verification and validation of simulation models whilst desirable, are often practically impossible due to cost and time restrictions (Pidd, 2004). Typically, only a subset of the available techniques and methods for model verification and validation are used.

3. Applications of Simulation in Software Engineering

Simulation models have been applied in many technical fields and are increasingly used for problems in business management and software engineering management. This section summarizes applications of simulation and some of the benefits that can be obtained.

Abdel-Hamid and Madnick (1991) were among the first to apply simulation modelling in software project management. They focused on project cost estimation and the effects of project planning on product quality and project performance. During the last decade many new process simulation applications in software engineering have been published, focusing on other specific topics within software project and process management [e.g., Christie (1999a); Kellner et al. (1999); Waeselynck and Pfahl (1994)]. Table 1 lists some significant publications in various application areas.

4. Simulation Techniques

The way in which a simulation model works depends on the modelling technique chosen. Generally, four important distinctions between types of simulation techniques can be made.

4.1. Deterministic Versus Stochastic Simulation

Simulation models that contain probabilistic components are called *stochastic*,[1] those that do not are termed *deterministic*. In the case of a deterministic simulation model, for a fixed set of input parameter values the resulting output parameter values

[1] The word "stochastic" is used here in a very broad sense of its meaning, i.e., referring to any type of source of randomness, including, for example, mutation or cross-over generation in genetic algorithms.

Table 1 Simulation applications in software engineering

Application area in software engineering	Selected publications
Project management	Lee and Miller (2004), Lin et al. (1997), Padberg (2006), Pfahl and Lebsanft (2000)
Risk management	Houston et al. (2001), Neu et al. (2002), Pfahl (2005)
Product and requirements engineering	Christie and Staley (2000), Ferreira et al. (2003), Höst et al. (2001), Lerch et al. (1997), Pfahl et al. (2006), Stallinger and Grünbacher (2001)
Process engineering	Bandinelli et al. (1995), Birkhölzer et al. (2004), Christie (1999b), Kuppuswami et al. (2003), Mišic et al. (2004), Powell et al. (1999), Raffo et al. (1999), Tvedt and Collofello (1995)
Strategic planning	Andersson et al. (2002), Pfahl et al. (2006), Williford and Chang (1999)
Quality assurance and management	Aranda et al. (1993), Briand and Pfahl (2000), Briand et al. (2004), Madachy (1996), Müller (2007), Raffo and Kellner (2000), Raffo et al. (2004), Rus (2002), Rus et al. (1999)
Software maintenance and evolution	Cartwright and Shepperd (1999), Smith et al. (2005), Wernick and Hall (2004)
Global software development	Roehling et al. (2000), Setamanit et al. (2006)
Software acquisition management and COTS	Choi and Scacchi (2001), Häberlein (2003), Häberlein and Gantner (2002), Ruiz et al. (2004), Scacchi and Boehm (1998)
Product-lines	Chen et al. (2005)
Training and education	Dantas et al. (2004), Drappa and Ludewig (1999), Madachy and Tarbet (2000), Oh Navarro and van der Hoek (2004), Pfahl et al. (2001)

will always be the same for simulation runs. In the case of a stochastic simulation model, the output parameter values may vary depending on stochastic variation of the values of input parameters or intermediate (internal) model variables. Since the variation of input and intermediate variables is generated by random sampling from given statistical distributions, it is important to repeat stochastic simulation runs for a sufficient number of times in order to be able to observe the statistical distribution of output parameter values. This number depends on limitations to computing power and how much confidence in simulation results is required.

4.2. Static Versus Dynamic Simulation

Static simulation models capture the variation of model parameters at one single point in time, while dynamic simulation models capture the behaviour of model parameters over a specified period of time.

Static simulation in software engineering is often used as a reference to stochastic Monte Carlo simulation which does not investigate behaviour over time. Related examples can be found in (Briand and Pfahl, 2000; Houston, 2003; McCabe, 2003).

4.3. Continuous Versus Event-Driven Simulation

Dynamic simulation models can be either continuous or event-driven. The difference between continuous and event-driven simulation models is the way in which the internal state of the model is calculated.

Continuous simulation models update the values of the model variables representing the model state at equidistant time steps based on a fixed set of well-defined model equations. Essentially, the model equations in continuous simulation models establish a set of time-dependent linear differential equations of first or higher order. Since such mathematical systems usually cannot be solved analytically, the differential equations are transformed into difference equations and solved via numerical integration. The most popular representative of continuous simulation is System Dynamics (SD) (Coyle, 1996). SD was originally invented by Jay Forrester in the late 1950s (Forrester, 1961) and has its roots in cybernetics and servomechanisms (Richardson, 1991). Since the end of the 1980s, when Abdel-Hamid and Madnick published the first SD model for software project management support, more than 100 other SD models in the application domain of software engineering have been published (Pfahl et al., 2006). Thus, SD can be considered the most frequently used dynamic simulation technique in this domain.

Event-driven simulation models update the values of the model variables as new events occur. There exist several types of event-driven simulation techniques. The most frequently used is discrete-event (DE) simulation. DE simulation models are typically represented by a network of activities (sometimes called stations) and items that flow through this network. The set of activities and items represent the model's state. The model's state changes at the occurrence of new events, triggered by combinations of items' attribute values and activities' processing rules. Events are typically generated when an item moves from one activity to another. As this can happen at any point in time, the time between changes in the model state can vary in DE simulations. There exist several other – but less popular – types of event-driven simulation, namely Petri-net based simulation (Bandinelli et al., 1995; Fernström, 1993; Gruhn and Saalmann, 1992; Mizuno et al., 1997), rule-based simulation (Drappa et al., 1995; Mi and Scacchi, 1990), state-based simulation (Humphrey and Kellner, 1989; Kellner and Hansen, 1989), or agent-based simulation (Huang and Madey, 2005; Madey et al., 2002).

4.4. Quantitative Versus Qualitative Simulation

Quantitative simulation requires that the values of model parameters are specified as real or integer numbers. Hence, a major prerequisite of quantitative simulation is either the availability of empirical data of sufficient quality and quantity or the availability of experts that are willing to make quantitative estimates of model parameters. Often, the quantitative modelling approach is costly and time-consuming and might not be appropriate for simulations that aim at delivering simple trend

analyses. Qualitative simulation is a useful approach if the goal is to understand general behaviour patterns of dynamic systems, or when conclusions must be drawn from insufficient data.

QUAF (Qualitative Analysis of Causal Feedback) is a qualitative simulation technique for continuous process systems (Rose and Kramer, 1991). The method requires no numerical information beyond the signs and relative values of certain groups of model parameters. QSIM (Qualitative SIMulation) is another well-established qualitative technique for continuous simulation (Kuipers, 1986). Instead of quantifying the parameters of the differential equations underlying the continuous simulation model, it is only required to specify the polarity (i.e., positive or negative) of model functions, indicating whether they represent an increase or decrease of a quantity over time.

In the case of event-driven simulation, for example, Petri-net based and rule-based simulation can be conducted purely qualitatively, if events (e.g., the activation of transitions in Petri-nets, or the execution of a rule in rule-based systems) are triggered exclusively based on the evaluation of non-quantitative conditions.

4.5. Hybrid Simulation

Dynamic simulation models that combine continuous with event-driven or deterministic with stochastic elements are called hybrid simulation models. One benefit of hybrid approaches is the possibility to combine the advantages of stochastic, continuous and event-driven models. In the case of hybrid models that combine continuous and event-driven simulation, however, the drawback is increased model complexity. An example of a hybrid simulation model that combines continuous with event-driven simulation can be found in (Martin and Raffo, 2001).

5. Simulation Tools

Today, many software tools are available to support the various simulation techniques described above. Compared to the first tools available in the 1960s, 1970s, and 1980s, most of today's more popular tools have a user-friendly interface and are inexpensive, making them practical to use for a large variety of decision making situations. Today, most tools

1. Allow for rapid model development through using, for example
   (a) Drag and drop of iconic building blocks
   (b) Graphical element linking
   (c) Syntactic constraints on how elements are linked
2. Are very reliable
3. Require little training
4. Are easy to understand

Because of these features, simulation tools allow modellers to develop large detailed models rapidly. Modern tools have followed the evolution of software languages and software development environments. Now they focus on model design and a proper visualization rather than on programming the simulation logic.

The simulation tools in today's market place are robust and reasonably inexpensive. Most tools cost in the range of $1,000–10,000, and free versions are available for experimentation and evaluation. They run on standard PC hardware, and are therefore affordable even for small organizations with tight budgets.

The number of simulation tools is large, in particular if one counts the ever-growing number of simulation environment research prototypes developed at universities all over the world. In principle, a simulation model based on any of the above mentioned simulation techniques can also be implemented in an ordinary programming languages (e.g., Java®), or by using general purpose simulation languages (e.g., MATLAB®). However, several commercial simulation tools use the most important simulation techniques and are suited to support software engineering problems. Table 2 characterizes three popular examples of simulation tools supporting SD, DE, and Monte Carlo simulation, respectively.

The choice of a simulation tool environment depends on several factors. Since the prices are comparatively low, the most important factor is the appropriateness of the simulation technique that is supported. In a professional simulation environment, in conjunction to the simulation modelling tool, other tools are often used. Professional simulation studies typically involve information systems or data bases which store the input, calibration, and output data, a statistical distribution fitter to analyze the calibration data, and an optimizer. High-end tools such as the more expensive versions of VENSIM® and EXTEND® already include the distribution fitters and optimizers.

Table 2 Examples of commercial simulation tools used in software engineering

Tool name	Main focus	Characterization	Interesting features
VENSIM® (Vensim, 2006)	Support of SD simulation	Dynamic, continuous, deterministic and stochastic, quantitative	Optimization function, calibration support, graphical modelling language (using standard SD symbols), animation, can emulate event-driven simulation to some extend by introducing if-then-else-conditions
EXTEND® (Extend, 2006)	Support of DE and SD simulation	Dynamic, event-driven and continuous, deterministic and stochastic, quantitative	Optimization support, graphical modelling language, strong modularization capability; statistical fitting (StatFit®), library source code available
@RISK® (@Risk, 2007)	Monte Carlo simulation	Static, deterministic, stochastic, quantitative	Can easily be integrated with standard spreadsheet tools (i.e., Microsoft's EXCEL®), provides functionality for distribution fitting (BestFit®)

Next follows a brief introduction into the SD modelling tool VENSIM®, which will be used in the presentation of a process simulation example in Sect. 6 below.

5.1. Essentials of System Dynamics Models

SD models are represented by a set of difference equations, which is resolved by numerical integration. Model variables, which represent the model state are called *levels* and have the following form:

$$\text{Level}\left(t+dt\right) = \text{Level}\left(t\right) + \text{Integral}\left[\text{Rate\_in}\left(t\right) - \text{Rate\_out}\left(t\right)\right] dt \qquad (1)$$

The value of a level at a certain point in time[2] depends on its value at the previous discrete point in time plus the integral of the inflows minus the outflows. The initialization of the level happens at the start time of a simulation. In the world of difference equations this would correspond to the starting conditions. In the example given by (1) there is only one inflow, represented by the *rate* variable Rate_in(t) and one outflow, represented by Rate_out(t). Level variables can be considered as containers or reservoirs that accumulate some tangible (e.g., a pile of papers) or intangible (e.g., number of defects in a documents or motivation level of developers) entities, represented by some countable attribute.

In the physical world, the quantities of the accumulated commodities in a reservoir can be regulated through inflow and outflow pipes, each pipe having a valve. In SD models rate variables play the role of valves. Like levels, rates are represented by equations. Rates can depend on levels, e.g., if information feedback concerning the quantity in a level affects the rate of flow elsewhere in the model, on constants, or on auxiliary variables, which are used as abbreviations for intermediate calculations to break up more complex computations. (2) gives an example of a rate variable that represents the development rate (inflow) of a design document (level variable DesignDocSize). If DesignDocSize(t) is less than the estimated expected size of the design document (constant TargetSize), then the daily amount of design documentation added to DesignDocSize equals the product of the number of active designers (Workforce allocated at time t) and the average productivity per person (constant AveragePr oductivityPerPerson). If the design document is complete, i.e., DesignDocSize ≥ TargetSize, then there is nothing to do and the rate variable DesignDevelopment Rate equals 0. Thus no more is added to DesignDocSize unless or until some other activity in the model reduces DesignDocSize or increases TargetSize.

[2] "dt" denotes a time step from one discrete point in time to the next.

$$\begin{aligned}&\text{DesignDevelopmentRate}(t) = \\ &\text{IF THEN ELSE} \\ &\quad (\text{DesignDocSize}(t) < \text{TargetSize}, \\ &\quad \text{Workforce}(t) * \text{AverageProductivityPerPerson}, \ 0)\end{aligned} \tag{2}$$

5.2. A System Dynamics Tool: VENSIM®

The VENSIM tool offers a development workbench supporting both textual and graphical model representations. The symbols that are used for the basis model variables and constants follow a de-facto-standard for SD modelling. Level variables are represented as boxes, while rates are represented as valves on pipes (double lines) connected with these boxes. Constants and auxiliary variables are simply represented by their names. Flows of information are represented by single-line arrows.

Figure 2 shows a screen shot of the VENSIM® modelling workbench with a loaded view (sub-model) of a SD model representing the design phase of a software development project. The flow through the pipes attached to level variables (e.g., *design to do size* and *design doc size* in Fig. 3) is regulated by rate variables, represented by valve symbols (e.g., *development activity* in Fig. 3). Auxiliary

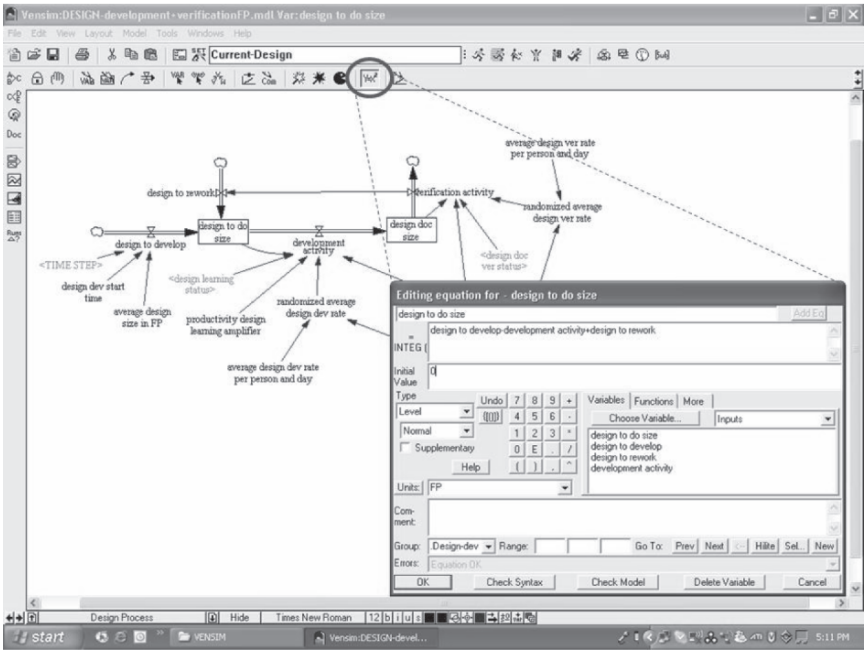

Fig. 2 VENSIM workbench with activated equation editor

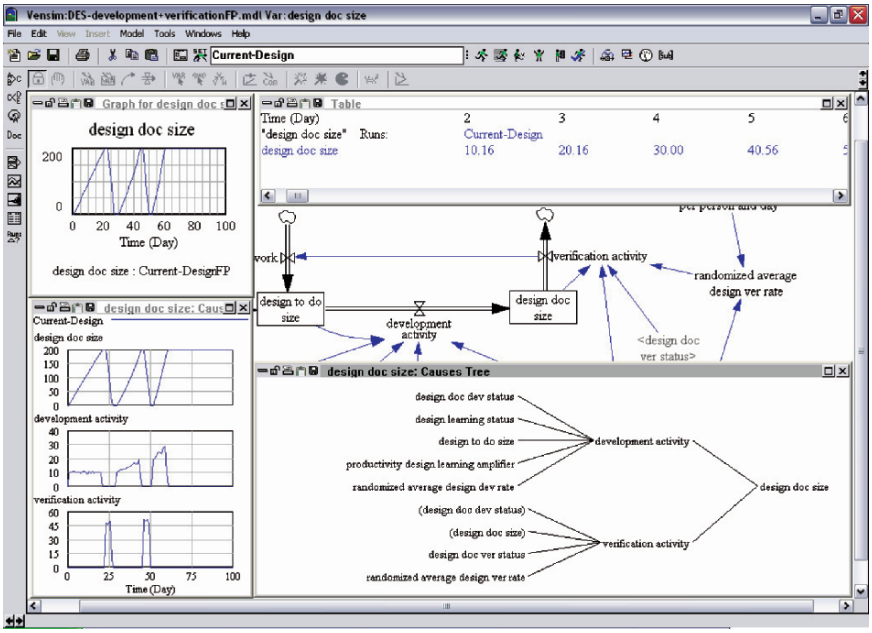

Fig. 3 VENSIM workbench with activated analysis and output tools

variables and constants are represented simply by their names. Values of level, rate, or auxiliary variables are calculated by evaluating functions of the form $y = f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are other variables and constants. The variables and constants involved in such a function are illustrated by a connecting arc (or pipe).

The definition of a function is done through a text-based equation editor. The equation editor window automatically pops up if the details of an equation have not yet been fully defined and the workbench button [$y = x^2$] is pressed (see Fig. 2). The equation editor not only provides an input window for specifying the exact function but also provides fields for specifying the variable unit and an explanatory comment. The equation editor automatically performs simple syntax and consistency checks. There exists also an equivalent textual representation of the entire model (not shown in Fig. 2). The textual representation of model equations has the advantage that string insertion, deletion, and renaming can easily be performed for the complete model.

The list of buttons directly above the graphical modelling panel offers specialized functionality for adding, deleting, removing, renaming, hiding, and showing of model variables. The column of buttons on the left hand side of the modelling panel provides specialized functionality for model analysis and simulation output presentation in the form of graphs or tables (cf. Fig. 3). For example, the window in the lower right corner of the screen shot presented in Fig. 3 shows two levels of causal dependencies between variables. Values shown in parentheses indicate feedback loops. From the open window within the modelling panel one sees that:

$$\text{design doc size} = f\left(\text{development activity, verification activity}\right) \quad (3)$$

while

$$\begin{aligned}\text{development activity} = f\,(&\text{design doc dev status, design learning status,}\\ &\text{design to do size, productivity design learning amplifier,}\\ &\text{randomized average design dev rate})\end{aligned} \quad (4)$$

Graphs showing the reverse dependencies, i.e., variable or constant uses, can also be automatically generated (not shown in Fig. 3). Other windows in Fig. 3 show the output of one simulation run (here: Current-Design) in the form of tables and graphs (lower and upper windows in the left half of the graphical modelling panel), as well as information about the model structure.

6. A Reference Simulation Model for Software Development Processes

This section shows a simulation model example and introduces the concept of a simulation reference process. The model is implemented as a stochastic SD model using the VENSIM® tool. Based on the example, a comparison between SD simulation and DE simulation will be made, and the advantages and disadvantages of each technique discussed.

6.1. A Generic Software Development Process

The following example presents a generic – in the sense of re-usable and adaptable – implementation of a standard process typically occurring in any constructive software development phase.

The left-hand side of Fig. 4 shows a typical development and verification work-flow of any type of software-related artefacts. The work-flow presentation uses the following symbols: boxes (for artefacts), ovals (for activities), hexagons (for resources), and arcs (representing *uses*, *produces*, and *consumes* relationships). An artefact may be, for example, a requirements, design, test, or code document. The actual artefact to be developed and verified is positioned in the centre of the work-flow. Before the development of this artefact can start, some input information must be available. For example, a design documents needs to know which requirements have been specified in a previous project stage. The development activity transforms an available artefact input into a new or modified artefact, e.g., a set of requirements into a design document. This artefact is then checked in a verification activity. The result of the verification activity, e.g., an inspection, is a list of defects in the newly created or modified artefact, which in turn is the basis for rework of

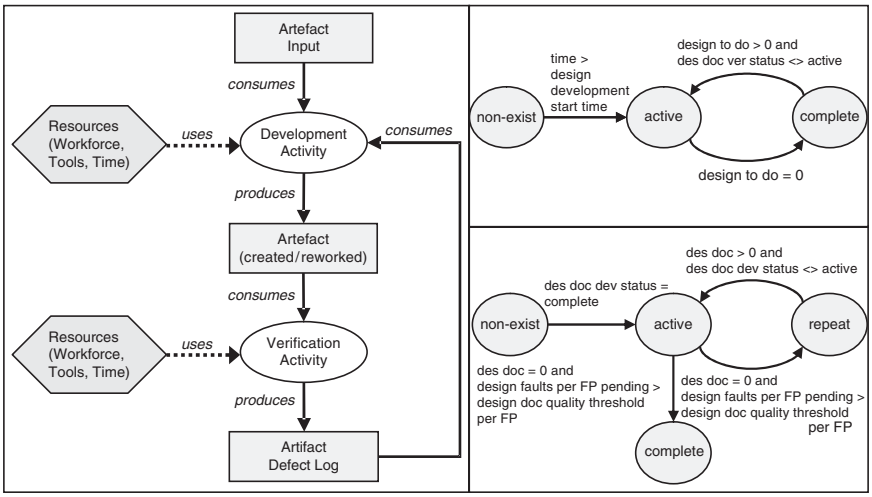

Fig. 4 Generic artefact development/verification process

the artefact. The rework loop is indicated in Fig. 4 by the *consumes*-relationship between the artefact defect log and the development activity. No distinction is made between initial work and rework performed on previous output. Activities use resources, e.g., personnel (implying some cost), tools (also incurring some cost, and supporting certain techniques), techniques (implying a need to quantify productivity), and time.

For larger simulation models, covering more than one stage of the software development process, instances of the generic work-flow shown in Fig. 4 can be combined sequentially by connecting work-flows that create predecessor artefacts with work-flows that create successor artefacts, and concurrently to represent work-flows conducted in parallel that produce separate instances of artefacts of the same type.

The right-hand side of Fig. 4 shows the control of the work-flow, expressed in terms of states that the artefact can assume in relation to its development (upper diagram) and verification activities (lower diagram), and the transitions between states, including the conditions for activating a transition. For example, a development activity related to the artefact "requirements" can either have not yet been started ("non-exist"), be active ("active"), or it can be completed ("complete"). The transition from "non-exist" into "active" is triggered as soon as the elapsed time t is greater than the defined starting time of the related development activity. A transition from "active" to "complete" is triggered, if all of the artefact inputs have been used up in producing the output document (e.g., a design or code document). If rework needs to be done in order to correct defects detected during verification, then a transition from "complete" back to "active" is triggered. The state-transition diagram associated with the verification activity is similar to that of the development activity. The only difference is its fourth state, "repeat." This state signals that a repetition of the verification activity is needed after rework of the defects found in

the previous verification round has been completed. The decision as to whether the verification step must be repeated depends on the number of defects found per size unit of the artefact. For example, if requirements size is measured in Function Points (FPs), then a threshold value can be defined in terms of defects per FP. If the number of detected defects per FP is larger then the defined threshold value, then verification has to be repeated, otherwise the document is considered (finally) complete after rework.

6.2. Conceptualization of the Generic Software Development Process

While the work-flow on the left-hand side of Fig. 4 is static, the control-flow presented on the right-hand side contains some behavioural information. Both static and behavioural information contained in the generic software development (and verification) process are the basis for the creation of a related simulation model, e. g., using the System Dynamics (SD). As will be shown below, the process shown in Fig. 4 is actually a re-usable pattern that captures the most important aspects of the work-flow, including activities and artefacts, as well as resources that will be used. It also captures some behavioural aspects by specifying the possible states of an activity (or the resulting artefact) and the feasible state transitions. However, for the development of an SD simulation model more information is needed. First, measurement data are needed for model calibration. Second, additional information about managerial decision rules and control policies are needed in order to understand the causal relationships that govern the process behaviour.

Table 3 lists attributes that often characterize the entities of the generic artefact development/verification process (second column), and gives typical examples (third column). The transformation of these attributes into SD model parameters follows a regular pattern (cf. fourth column). The attribute “efficiency” of the entity “activity” always maps to a rate variable. Attributes of artefacts and resources usually map to level variables. However, there are situations where an attribute value of an artefact or resource is considered constant. In particular, this is the case when – for the purpose of the modelling task – it is of no interest to model the variation of an attribute value. An example is the number of designers involved in a design task which may be controlled by processes outside the scope of the activities to be modelled, e.g. senior management policy. The fifth column of Table 3 indicates how the values of model parameters are determined. Level and rate variables are calculated by their defining functions. Constants are either defined by the model user (INPUT) or, in the case that they are used to calibrate the model, based on expert estimates (EST) or derived from available empirical data (EMP). Calibration constants are either deterministic (e.g., by taking the mean) or stochastic (e.g., by triangulation of expert estimates or by statistically fitting the distribution of empirical data).

Table 3 Mapping of generic process attributes to SD model parameters

Process description			System dynamics	
Entity	Attribute	Example	Parameter type	Quantification
Artefact	Size	Design/specification document: – Function points (FP) – Pages	Level Constant	CALC (from flow rates) INPUT or EST or EMP
		Code document: – Lines of code (LOC)	Level Constant	CALC (from flow rates) INPUT or EST or EMP
		Test plan: – Number of test cases	Level Constant	CALC (from flow rates) INPUT or EST or EMP
	Quality	Spec./design/code/test plan: – Defects injected, detected, corrected	Level Constant	CALC (from flow rates) INPUT or EST or EMP
	State	Spec./design/code/test plan: – State values	Level	CALC (flow rates emulate state-transition logic)
Activity Efficiency		Spec./design/code/test plan: – Development (and rework) volume per time unit – Verification (and validation) volume per time unit – Defect injection, detection, correction ($\rightarrow$ rework) per time unit	Rate	CALC (based on attribute values of used Resources)
Resource	Size	Workforce: – Number of architects, designers, programmers, testers, etc.	Level Constant	CALC (from flow rates) INPUT or EST or EMP
	Quality	Workforce: – Training – Experience	Level Constant	CALC (from flow rates) INPUT or EST or EMP
	Productivity	Development, verification, or validation technique: – Number of pages, FP, LOC, test cases developed, inspected, or tested per person and time unit	Constant	INPUT or EST or EMP

(continued)

Table 3 (continued)

Process description			System dynamics	
Entity	Attribute	Example	Parameter type	Quantification
	Effectiveness	Development, verification, or validation technique: – Number of defects injected per document size unit – Number of defects detected per document size unit	Constant	INPUT or EST or EMP
	Cost	Workforce:	Level	CALC (from flow rates)
		– Variable cost (e.g., hourly rate)	Constant	INPUT or EST or EMP
		Development, verification, or validation tools:	Level	CALC (from flow rates)
		– Fixed costs (e.g., purchase price)	Constant	INPUT or EST or EMP
		– Variable costs (e.g., leasing cost, storage cost, energy cost)		

CALC calculated by simulation tool; *INPUT* input by model user; *EST* estimated by experts (modelled either deterministic or stochastic); *EMP* derived from empirical data (modelled either deterministic or stochastic)

Figure 5 shows the network of individual cause-effect relationships (so-called base mechanisms) of a SD model of the generic process. The most creative – and difficult – part during simulation model creation is the identification of cause-effect relationships that essentially generate the dynamic behaviour of the system, i.e., the variation of level variables over time. The control flows represented by the state-transition diagrams in Fig. 4 are not sufficient to explain the model behaviour, because they do not specify how relations between model variables change in response to value changes of the entities' attributes. One possible network of base mechanisms that (qualitatively) provides exactly this information is shown in Fig. 5. A base mechanism is represented as a directed graph connecting two nodes (model parameters), e.g., A $\rightarrow$ (+) B or A $\rightarrow$ (–) B. The arc that connects the nodes A and B can have a positive or a negative polarity, represented by "+" or "–" respectively. A positive polarity implies that B increases (or decreases), if A increases (or decreases). A negative polarity implies that B increases, if A decreases and vice versa. Using this encoding, the causal diagram in Fig. 5 can be read as follows:

1. If the workforce (e.g., the number of designers) is increased, then both development (or rework) and verification rate increase.

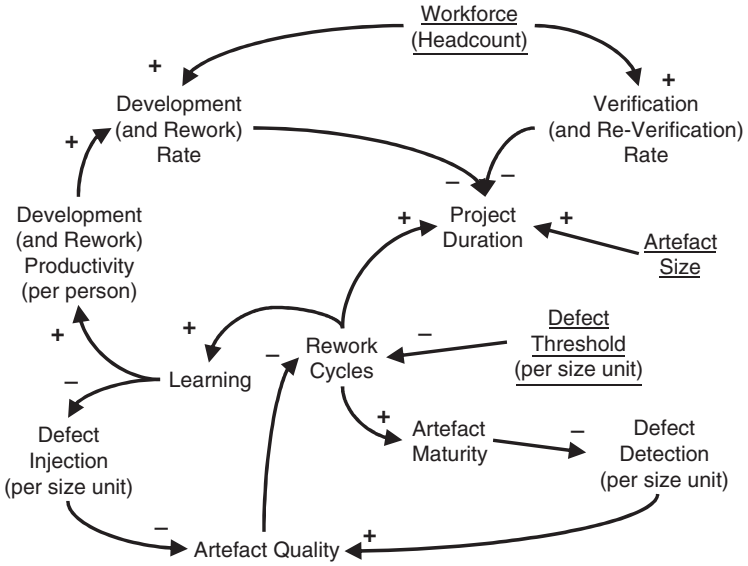

Fig. 5 Base mechanisms and causal network

2. If development and verification rates increase, then project duration decreases (because the artefact is developed faster).
3. If the artefact size is increased, then project duration increases (because a larger artifact has to be developed at a given rate).
4. If the defect threshold is increased (i.e., more defects per size unit have to be found before a re-verification is triggered), then possibly fewer rework cycles (incl. re-verification) have to be performed.
5. If fewer rework cycles (incl. re-verification) are performed, then project duration decreases.
6. If more re-work cycles are performed, then there is more learning and increased product maturity.
7. If there is more learning, then development productivity increases and defect injection (per size unit) decreases.
8. If defect injection (per size unit) decreases, then artefact quality increases.
9. If artefact maturity increases, then defect detection (per size unit) decreases.
10. If defect detection (per size unit) decreases, then artefact quality decreases.

Figure 5 contains three underlined nodes (workforce, artefact size, and defect threshold). These nodes represent either calibration or input parameters of the simulation model. The parameter "Defect Threshold" specifies the number of defects needed to trigger a rework cycle. It determines whether a verification step needs to be repeated (cf. in Fig. 4 the state-transition diagram associated with artefact verification). The importance of the parameter "Defect Threshold" resides in the fact that it not only plays a crucial role in the decision to repeat the verification

step, but also because it triggers workforce learning and product maturation. A repetition of the verification (and, as a consequence, the rework) step has multiple effects. First of all, it increases project duration. On the other hand, it speeds up the development (more precisely: rework) rate due to learning. Similarly, due to learning, it reduces the defect injection (per size unit) during rework. Finally, it also decreases the defect detection rate during the subsequent verification step due to product maturation, because most of the defects have already been detected, and there are only a few defects still contained in the artefact which are harder to detect. The last two effects mentioned have a damping effect on the number of rework (and re-verification) cycles, since they both make it more probable that the number of defects detected during re-verification are below the value of model parameter "Defect Threshold." This is an example of negative feedback.

It should be pointed out that the causal network in Fig. 5 is only a subset of the base mechanisms that typically drive the behaviour of a software project. For example, normally one would expect an influence on development rate from defect detection (per size unit). This, and possibly other base mechanisms, have been omitted to keep the example simple and compact. For the same reason, base mechanisms related to project effort consumption have been omitted.

6.3. Implementation of the Generic Process Using a System Dynamics Tool

With the help of the causal network – in addition to the information already contained in Table 3 – the full set of simulation model parameters are determined, and their type and role (from the perspective of the model user) can be defined. In the following, an example SD simulation model implementation for the generic code document development/verification process is presented.

Table 4 lists the complete set of model variables (second column), together with their type (third column) and usage (fourth column). Column one helps to trace back model parameters to the generic process map (cf. Fig. 4 with "artefact" replaced by "code document"). Using the mapping scheme presented in Table 3, the following mappings apply:

1. Size, quality, and state attributes of artefacts (Artefact Input, Artefact, Artefact Defect Log Size) are mapped to level variables
2. Efficiency attributes of activities (Development Activity and Verification Activity) are mapped to rate variables
3. Size, quality, productivity, and effectiveness attributes of resources (for Development and Verification) are mapped to level variables or constants

The list of attributes in Table 4 is very detailed. For example, the quality attribute information related to the code document distinguishes between the number of defects injected, the number of defects detected, the number of defects undetected (equals the difference between injected and detected defects), the number of defects

Table 4 Mapping of static process representation to SD model variables

Process map element	SD model parameter	Type	Usage
Artefact input [Size]	code to do size	Level	Output
initialization	*code dev start time*	*Constant*	*Input(E)*
initialization	*average code size in KLOC*	*Constant*	*Input(E)*
initialization	*code to develop*	*Rate*	*Internal*
Artefact [Size]	code doc size	Level	Output
Artefact [State Devel.]	code doc dev status	Level	Internal
Artefact [State Verif.]	code doc ver status	Level	Internal
initialization	*code doc quality limit per KLOC*	*Constant*	*Input(P)*
Artefact [Quality 1]	code faults generated	Level	Output
Artefact [Quality 2]	code faults detected1 (in one verification round)	Level	Output
re-initialization	*detected code faults flush*	*Rate*	*Internal*
Artefact [Quality 3]	code faults pending	Level	Output
Artefact [Quality 4]	code faults corrected1 (in one rework round)	Level	Output
re-initialization	*corrected code faults flush*	*Rate*	*Internal*
Artefact [Quality 5]	code faults undetected	Level	Output
Artefact Defect Log [Size 1]	code faults detected (total)	Level	Output
Artefact Defect Log [Size 2]	code faults corrected (total)	Level	Output
Devel. Activity [Effic. 1]	development activity	Rate	Internal
calibration	productivity code learning amplifier	Constant	Input (C)
Devel. Activity [Effic. 2]	code fault generation	Rate	Internal
calibration	quality code learning amplifier	Constant	Input (C)
Devel. Activity [Effic. 3]	code fault correction	Rate	Internal
Verif. Activity [Effic. 1]	verification activity (= code to rework)	Rate	Internal
Verif. Activity [Effic. 2]	code fault detection	Rate	Internal
Artefact State Trans. (Dev.)	cdd status change	Rate	Internal
Artefact State Trans. (Ver.)	cdv status change	Rate	Internal
Resources (Devel.) [Size]	Workforce	Constant	Input (E)
Resources (Devel.) [Qual.]	code learning status	Level	Output
Resources (Devel.) [Prod. 1]	average code dev rate per person and day	Constant	Input (C)
Resources (Devel.) [Prod. 2]	average code fault injection per KLOC	Constant	Input (C)
Resources (Verif.) [Size]	Workforce	Constant	Input (E)
Resources (Verif.) [Prod.]	average code ver rate per person and day	Constant	Input (C)
Resources (Verif.) [Effect.]	code ver effectiveness	Constant	Input (C)
Res. State Trans. (Qual.)	cl status change	Rate	Internal

Devel. development; *Effic*. efficiency; *Prod*. productivity; *Qual*. quality; *Res*. resources; *Trans*. transition; *Verif*. verification; *C* calibration; *E* exploration; *P* policy

corrected, and the number of defects pending (equals the difference between detected and not yet corrected defects). Additional distinctions could be made, e.g., between different defect types or severity classes. For the sake of the simplicity of the presentation, these additional distinctions have not been included in the example presented here.

Model parameters that are of purely technical nature are printed in italics. For example, in order to set up a simulation run, certain initializations have to be made, or for the realistic calculation of model attributes, coefficients in the related model equations have to be calibrated.

Typically, level variables play the role of output parameters, since they represent the state of the modelled system. Constants play the role of input parameters. Depending on their purpose, three types of input parameters can be distinguished: policy (P), exploration (E), and calibration parameters (C).

Policy parameters like, for example, the variable *code doc quality limit per KLOC* represent process specific threshold values which are evaluated in managerial decision rules. In the example, the threshold for the number of detected defects per KLOC in a verification step determines whether a re-verification has to be performed.

Calibration parameters like, for example, the variable *productivity code learning amplifier* help to quantify the effects imposed by one or more model variables on another model variable realistic.

Finally, exploration parameters like the variables *average code size in KLOC* or *workforce* represent those model parameters whose effect on the overall behaviour of the system is subject to analysis. In the example, the process completion (i.e., the time when code development is complete) as well as code quality in terms of the density of undetected defects after verification (*code faults undetected/ average code size in KLOC*) are model outputs that depend on other model variables including the size of the artefact to be developed (*average code size in KLOC*) and available resources (*workforce*).

Figures 7–9 show the graphical representations (views) of the complete SD model implementation for the code development and verification process:

1. Figure 6 captures the workflow in terms of size
2. Figure 7 captures the code development and verification states as well as the workforce learning state
3. Figure 8 captures the workflow (or defect co-flow) in terms of quality

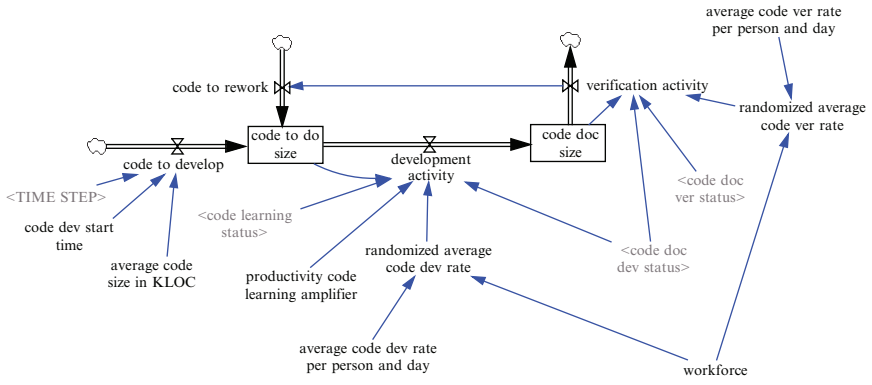

Fig. 6 Implementation of code development and verification work flow (view 1)

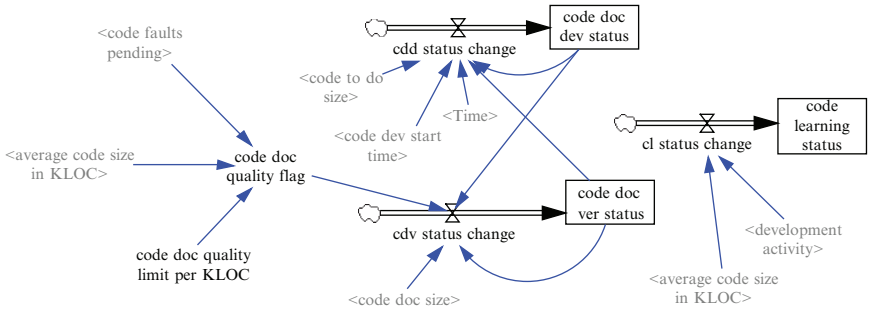

Fig. 7 Implementation of state attributes (view 2)

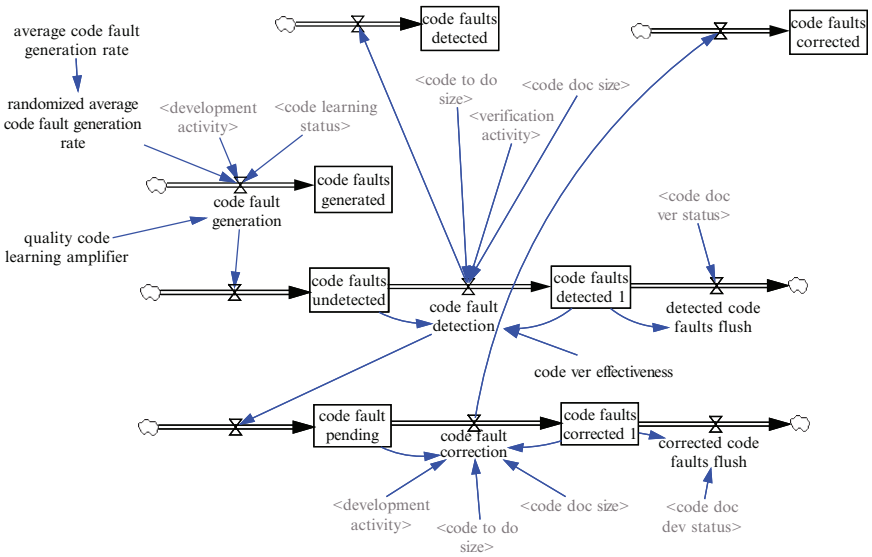

Fig. 8 Implementation of defect injection, detection, and correction co-flow (view 3)

The level variables, represented by boxes, are calculated with the help of inflow and outflow rates as defined by (1) introduced in Sect. 4. For example, level variable *code doc size* increases as a result of development activity and decreases as a result of verification activity (because verified code needs to undergo rework). The rate variables are calculated similarly to (2) introduced in Sect. 4.

In Fig. 6, the inflow rate *code to develop* initializes the level *code to do size*, which otherwise would be equal to zero, and thus no development or verification work has to be performed. At simulation time *code dev start time*, the value of *average code size in KLOC* flows into *code to do size*. In the example implementation, *code dev start time* and *average code size in KLOC* are model inputs. These two model parameters also define an interface to predecessor development and verification processes. For example, if a predecessor process produces a design

document, then the completion time and the size of this document can be used to calculate *code dev start time* and *average code size in KLOC*.

Figure 7 shows the part of the model which calculates the states related to code document development and verification as well as resource quality (learning). For example, using the encoding 0, 1, and 2, for the states "non-exist," "active," and "complete," respectively, the rate variable *cdd status change* is calculated as shown in (5) below.

```
cdd status change =
IF THEN ELSE
(code doc dev status = 0    /* state = "non-exist"
        :AND:Time >= code dev start time,
1,                          /* transition "non-exist" → "active"
IF THEN ELSE
(code doc dev status = 1    /* state = "active"
        :AND:code to do size <= 0,
1,                          /* transition "active" → "complete"
IF THEN ELSE
(code doc dev status = 2    /* state = "complete"
        :AND:code to do size > 0:AND:code doc ver status <> 1,
-1,                         /* transition "complete" → "active"
0)))                        /* do nothing
```
(5)

The first transition, from "non-exist" to "active," executes as soon as development has started, i.e., as soon as the simulation time is greater or equal to the defined development start time. The second transition, from "active" to "complete," executes as soon as there is no code waiting for implementation any more. The third transition, from "complete" back to "active," executes as soon as there is some code waiting for development and code verification is no longer active.

Figure 8 shows the defect co-flow, i.e., the injection (generation), detection, and correction of code faults. Fault generation and correction occur in parallel with code development and rework, while fault detection occurs in parallel with code verification (and re-verification). For example, the rate variable *code fault generation* is directly correlated with the rate variable *development activity*. The actual calculation of code fault generation is shown in (6) below.

```
code fault generation = development activity*
randomized average code fault injection per KLOC*
(1/MAX(1, code learning status^quality code learning amplifier))
```
(6)

From (6) it can be seen that there is only defect injection when *development activity* > 0. The actual number of faults generated per time step depends on the number of KLOC developed per time step and the *randomized average code fault injection*

per KLOC, which – in this example – is calculated by multiplying the *average code fault injection per KLOC* with a random number sampled from the triangular distribution triang(0.9, 1, 1.1,), where 1 represents the most probable value, and 0.9 and 1.1 the minimal and maximal values, respectively. The last factor in (6) models the learning effect. As soon as *code learning status* adjusted for the learning amplifier becomes greater than 1, the learning factor is less than 1 and thus the number of injected code faults decreases.

At the start of a simulation run, all model constants are initialized with a default value which can be modified by the user. Figure 9 shows a graphical user interface to the model, built using a Vensim utility, in the form of an input panel with slide bars, default initialization, and admissible value range. For example, variable *code ver effectiveness* is to be initialized with 0.75 (representing a defect detection effectiveness of the code verification technique of 75%), and maximum and minimum values of 0 and 1.

As soon as the simulation has started, the values of all model variables are calculated by Vensim® at each time step, which represents, for example, one work day. When the simulation run is complete the calculated values can be displayed either in tabular form or as graphs showing the time line on the *x*-axis and the variable value on the *y*-axis. Figures 10 and 11 below show example output graphs of the example model.

The upper part of Fig. 10 shows the simulation output for the level variables *code to do size* and *code doc size*. At simulation start (Time = 0), the amount of code work to do, in this case 200 KLOC, flows instantaneously into *code to do size*. This then decreases at a constant rate, caused by the development activity which transforms *code to do size* into *code doc size* (cf. Fig. 6). Consequently, the value of *code doc size* is exactly complementary to the value of *code to do size*, the sum of both always adding up to 200 KLOC. The lower part of Fig. 10 shows the behaviour of the state variables controlling the behaviour of code development, code verification, and learning, respectively. For example, one can see that *code doc dev status* equals 1 (“active”) while code is developed. As soon as there is nothing more

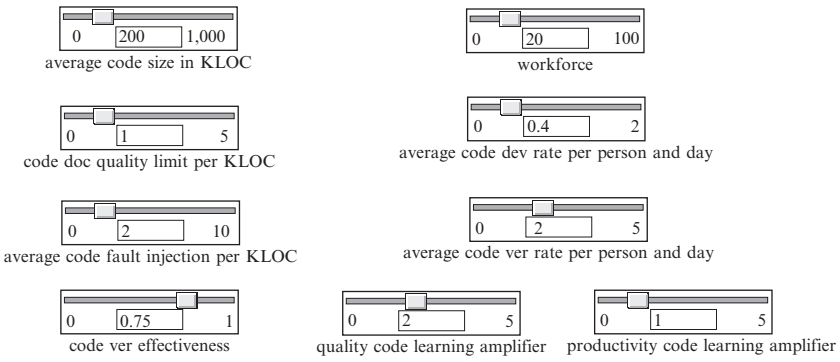

Fig. 9 Simulation input panel

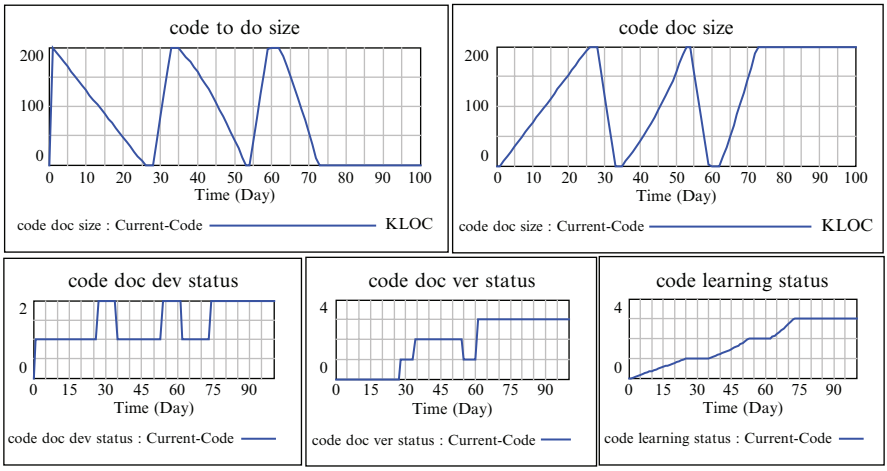

Fig. 10 Simulation output related to model views 1 and 2

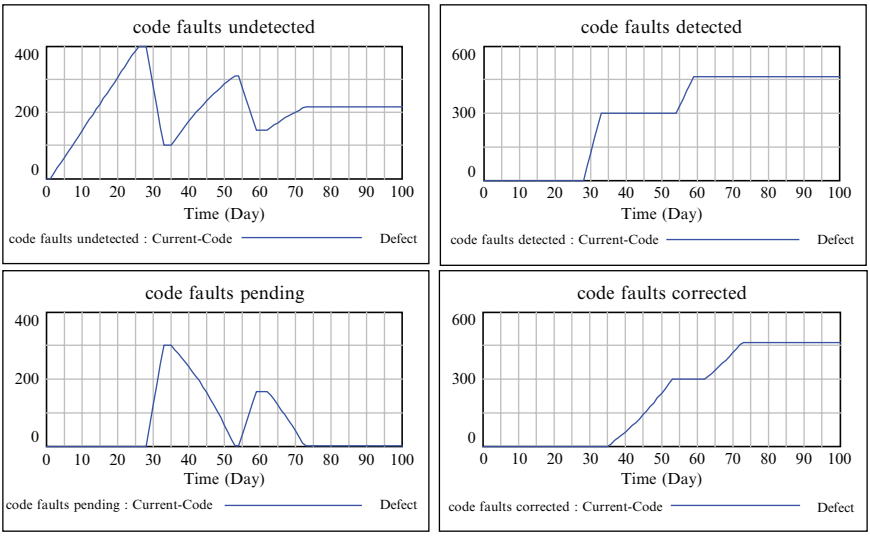

Fig. 11 Simulation output related to model view 3

to develop, i.e., *code to do size* = 0, it switches to 2 (“complete”). At that moment, *code doc ver status* switches from 0 (“non-exist”) to 1 (“active”). After some time during which verification is done, depending on how many defects are found, *code doc ver status* switches either to 2 (“repeat”) or 3 (“complete”). In Fig. 10, one can see that after the first verification round it is signalled that a second verification round needs to be performed (“repeat”).

Figure 11 shows a selection of diagrams related to code fault generation, detection, and correction. The model variable *code faults undetected* represents the difference between the numbers of injected and detected faults, while *code faults*

pending represents the difference between detected and corrected code faults. One can see that fault detection occurs when verification is active, and fault correction occurs when development (rework) is active.

6.4. Extension and Reuse of the Reference Simulation Model

The SD model developed in the previous section can be extended and reused in several ways. For example, as mentioned earlier, it is possible to make the model more realistic by adding a causal relationship between the number of errors detected and the fraction (size) of the artefact to be reworked.

The more interesting aspect of reusability is illustrated by Fig. 12. The figure shows the V-model software development process on the right hand side. Simulation models representing the *Design* and *Coding* phases are presented as boxes. For example, the Boxes labelled views 1C to 3C represent the SD model views presented in Figs. 7–9. In Figs. 7–9, the code documents developed and verified in the coding phase are represented by one single level variable. There is no differentiation between code sub-systems or modules. To facilitate a more detailed representation of reality, i.e., explicit modelling of individual subsystems (or even modules), the SD tool VENSIM® offers the possibility of "subscripting," i.e., the possibility of replacing a monolithic entity by an array of entities of the same type. A subscript works like the index of an array. With the help of this mechanism, potentially all variables used in the model views 1C to 3C can be duplicated. For example, if five code sub-systems shall be modelled, they would be represented by level variables *code doc size [1]* to *code doc size [5]*, or, if 100 code modules are to be modelled, the index of *code doc size* would run from 1 to 100, each index representing the *levels* and *rates* associated with each module.

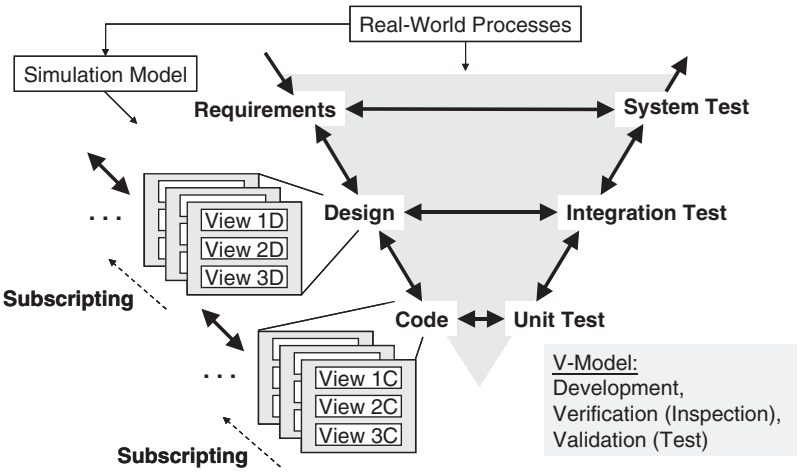

Fig. 12 Reuse-based construction of a simulation model representing a V-Model process

Finally, it is possible to represent the design and requirements specification phases of the V-Model process by simply duplicating the code related views 1C to 3C. This can be done by copying a complete view and replacing the sub-strings "code" by strings "design" in all variable names. Of course, the resulting Views 1D to 3D (and 1R to 3R) have to be re-calibrated based on suitable data or expert estimates. The connection between subsequent views requires only a few information links between variables, e.g., between model variables *design doc size* (which plays in the design phase the role that *code doc size* plays in the coding phase) and *average code size in KLOC*. These connections can be considered similar to "glue code" used to connect reusable software components.

Figure 13 shows several simulation output diagrams for a code development and verification process in which five sub-systems are developed concurrently. The size of each subsystem varies between 35 and 45 KLOC, accumulating to a total of 200 KLOC. One can see the individual traces for each subsystem. The development of one subsystem starts at Time = 0 (begin of coding phase), the others are more or less delayed due to variation in completion of required design documents. Similar graphs are generated for the design and requirements specification phases.

Figure 14 shows for each variable displayed in Fig. 13 the aggregated values of the individual code sub-systems. If compared to the monolithic simulation (i.e., without subscripting) presented in Figs. 11 and 12, one can see that the overall behaviour is similar but that some temporal displacement occurs due to late start of coding of some of the subsystems.

With some additional minor modifications, it is possible to model five subsystems in the design phase and, say, 100 modules in the coding phase. This enhancement requires a mapping of sub-system subscripts (used in the design views 1D to 3D) to module subscripts (used in the code views 1C to 3C). With this modification, the quality views for design (3D) and coding (3C) generate the simulation results shown in Fig. 15 (simulation time $T = 0$ at start of design phase). The Design phase lasts from simulation time $T = 1$ until $T = 140$ days, while the Coding phase starts at time $T = 96$ and ends at time $T = 174$ days. For each phase, the simulated values of injected, detected, pending, and undetected faults are shown.

6.5. Comparison Between System Dynamics and Discrete-Event Simulation

The simulation application example outlined in Sects. 6.2 and 6.3 demonstrated how SD captures complex software process behaviour with a small set of core modelling constructs (i.e., level and rate variables, and constants). This is possible by creation of generic model patterns that are reusable in several ways, either by replicating model variables via subscripting, or by duplicating complete submodels (i.e., model views) by simple text replacement (e.g., replacing the string "code" by the string "design").

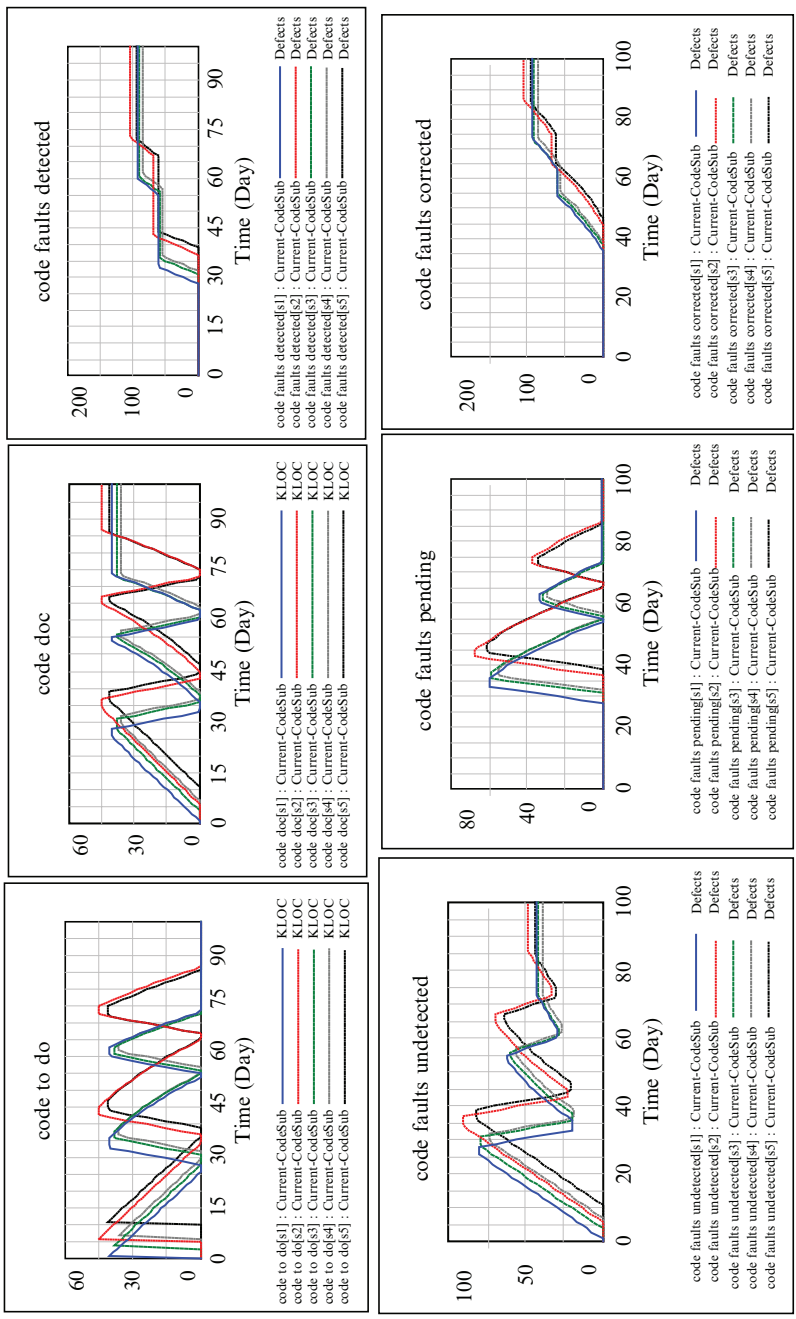

Fig. 13 Simulation outputs for concurrently coding five sub-systems

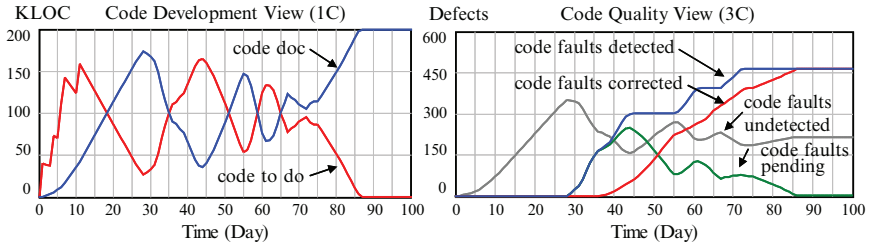

Fig. 14 Aggregated simulation outputs for concurrent coding of five sub-systems

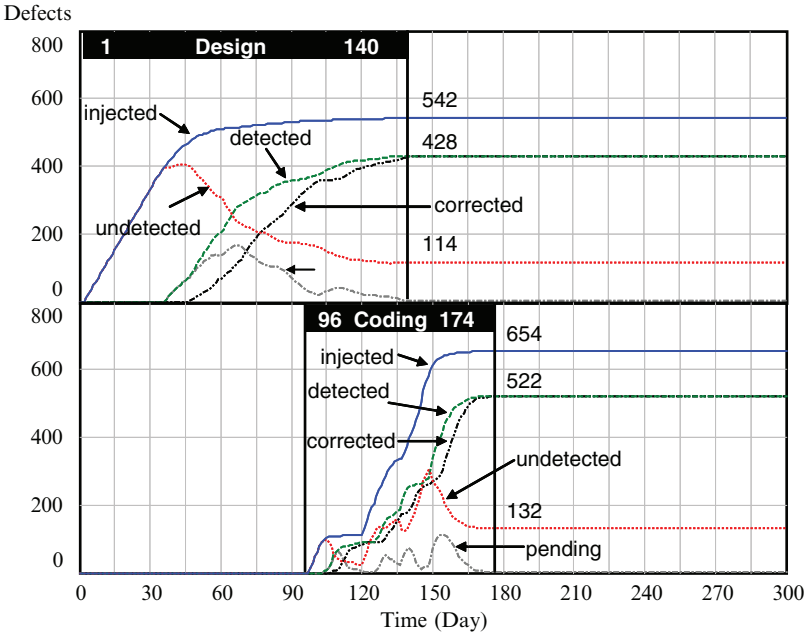

Fig. 15 Aggregated simulation outputs for concurrent sub-system design and module coding

Event-driven simulation techniques take a complementary perspective when modelling the generic artefact development and verification process introduced in Sect. 6.1. For example, instead of modelling the artefact as one monolithic document, e.g., of size 200 KLOC in the case of the code document, event-driven simulation models individual code units as single items which are routed through a sequence of processing stations, e.g., a station for development and a station for verification. These items have several attributes, e.g., size, state, number of defects (injected, detected, corrected), etc. The list of attributes can be extended or refined, e.g., by introducing attributes to distinguish defect types and severity classes. The attribute information determines, for example, the processing time in the development and verification stations, and the routing of an item after leaving a station.

What distinguishes DE simulation from SD simulation is the degree of model detail, the model representation, and the logic underlying the computation of model states. DE simulation modelling is very flexible and easily adaptable when it becomes necessary to add or change attributes of entities. Moreover, in DE simulation it is possible to model the behaviour of distinct real-world entities (e.g., artefacts, resources) of the same type individually, while SD typically models the average behaviour of a large number of entities of the same type. The possibility of subscripting mitigates this limitation of SD only to some extent.

One disadvantage of DE simulation comes as a downside of its ability to capture many details. DE simulation tools like, for example EXTEND®, offer a large number of different modelling constructs, often specifically tailored to manufacturing processes. Although these blocks are reusable in several contexts, more training is needed for the modeller to become familiar with the variety of options and they have to be adapted to capture software development processes. While DE simulation is capable to model production processes in greater detail, SD simulation models can capture not only the "mechanical" aspects of software development processes (which mainly consist of writing and checking different types of documents), but also the cause-effect mechanisms underlying the process behaviour. This includes the flow of information, which is important in software engineering, in contrast to material flows. Typically, information about these cause-effect relationships are part of the (mostly implicit) mental models of managers or decision makers, and contain intangible concepts like learning (cf. variable *code learning state* in the example above), motivation, stress, communication, decision policies, etc.

7. Practical Aspects

As a cautionary note it is well to remember that simulation has limitations and is not a "silver bullet." The predictive power of simulation strongly depends on the degree of model validity. While many scientific and engineering fields base their models on established physical laws, organizational models contain human aspects and intangible processes. This leads to two problems: It is difficult to gather data from human actors and it is very costly and sometimes not feasible to reproduce simulated scenarios in reality for the purpose of model validation.

Simulation is a simplification of the real world, and is thus inherently an approximation. As indicated in (Robertson, 1997) it is impossible to prove a priori the correctness of a simulation model that aims at generating previously unobserved and potentially unexpected behaviour. Thus, model verification and validation must be concerned with creating enough confidence in a model for its results to be accepted. This is done by trying to prove that the model is incorrect. The more tests that are performed in which it cannot be proved that the model is incorrect, the more increases confidence in the model.

Finally, one should not forget that simulation is neither a means in itself (it needs to be followed by action) nor does it generate new ideas. It is still the software manager's and simulation modeler's task to be creative in generating new scenarios for simulation, and in applying the simulation results to improve real-world processes. Simulation does not automatically produce new facts such as knowledge-based expert systems do (e.g., through inference).

8. The Future of Simulation in Software Engineering

The application of simulation techniques, in particular process simulation techniques, offers several interesting perspectives for improving management and learning in software organizations.

Business simulator-type environments (micro-worlds) can confront managers with realistic situations that they may encounter in practice. Simulation allows the rapid exploration of micro-worlds, without the risks associated with real-world interventions and provides visual feedback of the effects of managers' decisions through animation. Simulation increases the effectiveness of the learning process, because trainees quickly gain hands-on experience. The potential of simulation models for the training of managers in other domains than software engineering has long been recognized (Lane, 1995). Simulation-based learning environments also have the potential to play an important role in software management training and education of software engineers, in particular if they are offered as web-based (possibly distributed multi-user) applications.

Analyzing a completed project is a common means for organizations to learn from past experience, and to improve their software development process (Birk et al., 2002). Process simulation can facilitate post-mortem analysis. Models facilitate the replaying of past projects, diagnose management errors that arose, and investigate policies that would have supplied better results. To avoid having a software organization reproduce – and amplify – its past errors, it is possible to identify optimal values for measures of past project performance by simulation, and record these values for future estimation, instead of using actual project outcomes that reflect inefficient policies (Abdel-Hamid, 1993).

To further increase the usage (and usability) of simulation techniques in software engineering, the time and effort needed for model building must further be reduced. One step in this direction is to provide adaptable software process simulation frameworks. Similar to the process simulation reference model described above, these frameworks can be used like a construction kit with reusable model components. Supporting tools and methodological guidance must accompany reuse-based simulation modelling. Furthermore, simulation tools should be connected to popular project planning and tracking tools to decrease the effort of model parameterization and to increase their acceptance by software practitioners. As more and more companies improve their development process maturity, it is also expected that process simulation will gain more attention in industry.

References

Abdel-Hamid TK (1993) Adapting, Correcting and Perfecting Software Estimates: a Maintenance Metaphor. IEEE Computer 20–29.

Abdel-Hamid TK, Madnick SE (1991) Software Projects Dynamics – an Integrated Approach, Prentice-Hall, Englewood Cliffs, NJ.

Andersson C, Karlsson L, Nedstam J, Höst M, Nilsson BI (2002) Understanding Software Processes through System Dynamics Simulation: A Case Study, In: *Proceedings of 9th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp 41–48.

Aranda RR, Fiddaman T, Oliva R (1993) Quality Microworlds: Modeling the Impact of Quality Initiatives Over the Software Product Life Cycle. American Programmer 52–61.

Balci O (2003) Verification, Validation, and Certification of Modelling and Simulation Applications. In: *Proceedings of the 2003 Winter Simulation Conference*, pp 150–158.

Bandinelli S, Fuggetta A, Lavazza L, Loi M, Picco GP (1995) Modeling and Improving an Industrial Software Process. IEEE Transactions on Software Engineering 21(5): 440–453.

Banks J, Carson JS, Nelson BL (2000) Discrete-Event System Simulation, 3rd edn, MOUS Test Preparation Guides Series, Prentice-Hall, New York.

Barlas Y (1989) Multiple Tests for Validation of System Dynamics Type of Simulation Models. European Journal of Operational Research 42: 59–87.

Birk A, Dingsøyr T, Stålhane T (2002) Postmortem: Never Leave a Project without It. IEEE Software 19(3): 43–45.

Birkhölzer T, Dantas L, Dickmann C, Vaupel J (2004) Interactive Simulation of Software Producing Organization's Operations based on Concepts of CMMI and Balanced Scorecards. In: *Proceedings 5th International Workshop on Software Process Simulation Modeling (ProSim)*, Edinburgh, Scotland, pp 123–132.

Briand LC, Pfahl D (2000) Using Simulation for Assessing the Real Impact of Test-Coverage on Defect-Coverage. IEEE Transactions on Reliability 49(1): 60–70.

Briand LC, Labiche Y, Wang Y (2004) Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart. In: *Proceedings of International Conference on Software Engineering (ICSE)*, pp 86–95.

Cartwright M, Shepperd M (1999) On Building Dynamic Models of Maintenance Behavior. In: Kusters R, Cowderoy A, Heemstra F, van Veenendaal E. (eds.) *Project Control for Software Quality*, Shaker Publishing, Maastricht.

Cellier FE (1991) Continuous System Modeling, Springer Press, New York.

Chen Y, Gannod GC, Collofello JS (2005) A Software Product Line Process Simulator. In: *Proceedings of 6th International Workshop on Software Process Simulation and Modeling (ProSim)*, pp 102–109.

Choi SJ, Scacchi W (2001) Modeling and Simulating Software Acquisition Process Architectures. Journal of Systems and Software 59(3): 343–354.

Christie AM (1999a) Simulation: An Enabling Technology in Software Engineering. CROSSTALK – The Journal of Defense Software Engineering 12(4): 25–30.

Christie AM (1999b) Simulation in Support of CMM-Based Process Improvement. Journal of Systems and Software 46(2/3): 107–112.

Christie AM, Staley MJ (2000) Organizational and Social Simulation of a Requirements Development Process. Software Process Improvement and Practice 5: 103–110.

Coyle RG (1996) System Dynamics Modelling – A Practical Approach, Chapman & Hall, London.

Dantas A, de Oliveira Barros M, Lima Werner CM (2004) A Simulation-Based Game for Project Management Experiential Learning. In: *Proceedings of 16th International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pp 19–24.

Drappa A, Ludewig J (1999) Quantitative Modeling for the Interactive Simulation of Software Projects. Journal of Systems and Software 46(2/3): 113–122.

Drappa A, Deininger M, Ludewig J (1995) Modeling and Simulation of Software Projects. In: *Proceedings of 20th Annual Software Engineering Workshop*, Greenbelt, MD, USA, pp 269–275.

Extend (2006) http://www.imaginethatinc.com/ (accessed on March 22, 2006).
Fernström C (1993) PROCESS WEAVER: Adding Process Support to UNIX. In: *Proceedings of 2nd International Conference on the Software Process (ICSP)*, pp 12–26.
Ferreira S, Collofello J, Shunk D, Mackulak G, Wolfe P (2003) Utilization of Process Modeling and Simulation in Understanding the Effects of Requirements Volatility in Software Development. In: *Proceedings 4th Software Process Simulation Modeling Workshop (ProSim)*, Portland, USA.
Forrester JW (1961) Industrial Dynamics. Productivity Press, Cambridge.
Forrester JW, Senge P (1980) Tests for Building Confidence in System Dynamics Models. In: Forrester JW et al. (eds.) *System Dynamics*, North-Holland, New York.
Gruhn V, Saalmann A (1992) Software Process Validation Based on FUNSOFT Nets. In *Proceedings of 2nd European Workshop on Software Process Technology (EWSPT)*, pp 223–226.
Häberlein T (2003) A Framework for System Dynamic Models of Software Acquisition Projects. In: *Proceedings 4th Software Process Simulation Modeling Workshop (ProSim)*, Portland, USA.
Häberlein T, Gantner T (2002) Process-Oriented Interactive Simulation of Software Acquisition Projects. In: *Proceedings of First EurAsian Conference on Information and Communication Technology (EurAsia-ICT)*, LNCS 2510, Shiraz, Iran, pp 806–815.
Höst M, Regnell B, Dag J, Nedstam J, Nyberg C (2001) Exploring Bootlenecks in Market-Driven Requirements Management Processes with Discrete Event Simulation. Journal of Systems and Software 59(3): 323–332.
Houston DX (2003) A Case Study in Software Enhancements as Six Sigma Process Improvements: Simulating Productivity Savings. In: *Proceedings of 4th Software Process Simulation Modeling Workshop (ProSim)*, Portland, USA.
Houston DX, Mackulak GT, Collofello JS (2001) Stochastic Simulation of Risk Factor Potential Effects for Software Development Risk Management. Journal of Systems and Software 59(3): 247–257.
Huang Y, Madey GR (2005) Autonomic Web-Based Simulation. In: *Proceedings of Annual Simulation Symposium 2005*, pp 160–167.
Humphrey WS, Kellner MI (1989) Software Process Modeling: Principles of Entity Process Models. In: *Proceedings of 11th International Conference on Software Engineering (ICSE)*, Pittsburg, PA, USA, pp 331–342.
Kellner MI, Hansen GA (1989) Software Process Modeling: A Case Study. In: *Proceedings of 22nd Annual Hawaii International Conference on System Sciences, Vol. II – Software Track*, pp 175–188.
Kellner MI, Madachy RJ, Raffo DM (1999) Software Process Simulation Modeling: Why? What? How?. Journal of Systems and Software 46(2/3): 91–105.
Kuipers B (1986) Qualitative Simulation. Artificial Intelligence 29(3): 289–338.
Kuppuswami S, Vivekanandan K, Rodrigues P (2003) A System Dynamics Simulation Model to Find the Effects of XP on Cost of Change Curve. In: *Proceedings of 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP)*, LNCS 2675, pp 54–62.
Lane DC (1995) On a Resurgence of Management Simulation Games. Journal of the Operational Research Society 46: 604–625.
Law A, Kelton WD (1999) Simulation Modeling and Analysis, 3rd edn, McGraw-Hill, New York.
Lee B, Miller J (2004) Multi-Project Management in Software Engineering Using Simulation Modeling. Software Quality Journal 12: 59–82.
Lerch FJ, Ballou DJ, Harter DE (1997) Using Simulation-Based Experiments for Software Requirements Engineering. Annals of Software Engineering 3: 345–366.
Lin CY, Abdel-Hamid TK, Sherif J (1997) Software-Engineering Process Simulation Model (SEPS). Journal of Systems and Software 38(3): 263–277.
Madachy RJ (1996) System Dynamics Modeling of an Inspection-Based Process. In: *Proceedings 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, IEEE Computer Society Press, pp 376–386.
Madachy RJ, Tarbet D (2000) Case Studies in Software Process Modeling with System Dynamics. Software Process Improvement and Practice 5: 133–146.

Madey G, Freeh V, Tynan R (2002) Agent-Based Modeling of Open Source using Swarm. In: *Proceedings of Americas Conference on Information Systems (AMCIS)*, Dallas, TX, USA, pp 1472–1475.

Martin R, Raffo D (2001) Application of a Hybrid Process Simulation Model to a Software Development Project. The Journal of Systems and Software 59: 237–246.

McCabe B (2003) Monte Carlo Simulation for Schedule Risks. In: *Proceedings of the 2003 Winter Simulation Conference*, pp 1561–1565.

Mi P, Scacchi W (1990) A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. IEEE Trans. Knowledge Data Engineering 2(3): 283–294.

Mišic VB, Gevaert H, Rennie M (2004) Extreme Dynamics: Towards a System Dynamics Model of the Extreme Programming Software Development Process. In: *Proceedings 5th International Workshop on Software Process Simulation Modeling (ProSim)*, Edinburgh, Scotland, pp 237–242.

Mizuno O, Kusumoto S, Kikuno Y, Takagi Y, Sakamoto K (1997) Estimating the Number of Faults Using Simulator Based on Generalized Stochastic Petri-Net Model, In: *Proceedings of the Asian Test Symposium (ATS)*, pp 269–274.

Müller M (2007) Analyzing Software Quality Assurance Strategies through Simulation, Fraunhofer IRB, Stuttgart, pp 262.

Münch J, Rombach HD, Rus I (2003) Creating an Advanced Software Engineering Laboratory by Combining Empirical Studies with Process Simulation. In: *Proceedings 4th Process Simulation Modeling Workshop (ProSim)*, Portland, USA.

Münch J, Pfahl D, Rus I (2005) Virtual Software Engineering Laboratories in Support of Trade-off Analyses. Software Quality Journal 13(4): 407–428.

Neu H, Hanne T, Münch J, Nickel S, Wirsen A (2002) Simulation-Based Risk Reduction for Planning Inspections. In: Oivo M, Komi-Sirviö S (eds.) *Proceedings 4th International Conference on Product Focused Software Process Improvement (PROFES)*, LNCS 2559, Springer Press, Berlin, pp 78–93.

Oh Navarro E, van der Hoek A (2004) SIMSE: An Interactive Simulation Game for Software Engineering Education. In: *Proceedings 7th IASTED International Conference on Computers and Advanced Technology in Education (CATE)*, pp 12–17.

Padberg F (2006) A Study on Optimal Scheduling for Software Projects. Software Process Improvement and Practice 11(1): 77–91.

Pfahl D (2005) ProSim/RA – Software Process Simulation in Support of Risk Assessment. In: Biffl S, Aurum A, Boehm B, Erdogmus H, Grünbacher P (eds.) *Value-based Software Engineering*, Springer Press, Berlin, pp 263–286.

Pfahl D, Lebsanft K (2000) Knowledge Acquisition and Process Guidance for Building System Dynamics Simulation Models: An Experience Report from Software Industry. International Journal of Software Engineering and Knowledge Engineering 10(4): 487–510.

Pfahl D, Ruhe G (2002) IMMoS – A Methodology for Integrated Measurement, Modeling, and Simulation. Software Process Improvement and Practice 7: 189–210.

Pfahl D, Klemm M, Ruhe G (2001) A CBT Module with Integrated Simulation Component for Software Project Management Education and Training. Journal of Systems and Software 59(3): 283–298.

Pfahl D, Ruhe G, Lebsanft K, Stupperich M (2006) Software Process Simulation with System Dynamics – A Tool for Learning and Decision Support. In: Acuña ST, Sánchez-Segura MI (eds.) *New Trends in Software Process Modelling, Series on Software Engineering and Knowledge Engineering*, Vol. 18, World Scientific, Singapore, pp 57–90.

Pidd M (2004) Computer Simulation in Management Science, 5th edn, Wiley, New York, pp 328.

Powell A, Mander K, Brown D (1999) Strategies for Lifecycle Concurrency and Iteration: A System Dynamics Approach. Journal of Systems and Software 46(2/3): 151–162.

Raffo DM, Kellner MI (2000) Analyzing the Unit Test Process Using Software Process Simulation Models: A Case Study. In: *Proceedings 3rd Software Process Simulation Modeling Workshop (ProSim)*, London, UK.

Raffo DM, Vandeville JV, Martin RH (1999) Software Process Simulation to Achieve Higher CMM Levels. Journal of Systems and Software 46(2/3): 163–172.

Raffo DM, Nayak U, Setamanit S, Sullivan P, Wakeland W (2004) Using Software Process Simulation to Assess the Impact of IV&V Activities. In: *Proceedings 5th International Workshop on Software Process Simulation Modeling (ProSim)*, Edinburgh, Scotland, pp 197–205.

Richardson GP (1991) Feedback Thought in Social Science and Systems Theory, University of Pennsylvania Press, Philadelphia, PA, USA.

Robertson S (1997) Simulation Model Verification and Validation: Increase the Users' Confidence. In: *Proceedings of the 1997 Winter Simulation Conference*, pp 53–59.

Roehling ST, Collofello JS, Hermann BG, Smith-Daniels DE (2000) System Dynamics Modeling Applied to Software Outsourcing Decision Support. Software Process Improvement and Practice 5: 169–182.

Rose P, Kramer M (1991) Qualitative Analysis of Causal Feedback. In: *Proceedings of 9th National Conference on Artificial Intelligence (AAAI)*, pp 817–823.

Ruiz M, Ramos I, Toro M (2004) Using Dynamic Modeling and Simulation to Improve the COTS Software Process. In: *Proceedings 5th International Conference on Product Focused Software Process Improvement (PROFES)*, Kyoto, Japan, pp 568–581.

Rus I (2002) Combining Process Simulation and Orthogonal Defect Classification for Improving Software Dependability. In: *Proceedings 13th International Symposium on Software Reliability Engineering (ISSRE)*, Annapolis.

Rus I, Collofello C, Lakey P (1999) Software Process Simulation for Reliability Management. Journal of Systems and Software 46(2/3): 173–182.

Rus I, Biffl S, Hallig M (2002) Systematically Combining Process Simulation and Empirical Data in Support of Decision Analysis in Software Development. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Ischia, Italy, pp 827–833.

Rus I, Neu H, Münch J (2003) A Systematic Methodology for Developing Discrete Event Simulation Models of Software Development Processes. In: *Proceedings 4th International Workshop on Software Process Simulation and Modeling (ProSim)*, Portland, Oregon, USA.

@Risk (2007) @Risk Simulation Software: http://www.palisade-europe.com/ (accessed on June 26, 2007).

Sargent R (2003) Verification and Validation of Simulation Models. In: *Proceedings of 2003 Winter Simulation Conference*, pp 37–48.

Scacchi W, Boehm B (1998) Virtual Systems Acquisition: Approach and Transitions, Acquisition Review Quarterly 5(2): 185–216.

Setamanit S, Wakeland W, Raffo DM (2006) Exploring the Impact of Task Allocation Strategies for Global Software Development Using Simulation. In: Wang Q, Pfahl D, Raffo DM, Wernick P (eds.) *Software Process Change – SPW/ProSim 2006, Shanghai, China, May 2006, Proceedings* (LNCS 3966), Springer, Berlin, Heidelberg, pp 274–285.

Smith N, Capiluppi A, Ramil JF (2005) A Study of Open Source Software Evolution Data Using Qualitative Simulation. Software Process: Improvement and Practice 10(3): 287–300.

Stallinger F, Grünbacher P (2001) System Dynamics Modeling and Simulation of Collaborative Requirements Engineering. Journal of Systems and Software 59: 311–321.

Tvedt JD, Collofello JS (1995) Evaluating the Effectiveness of Process Improvements on Development Cycle Time via System Dynamics Modeling. In: *Proceedings Computer Science and Application Conference (COMPSAC)*, pp 318–325.

Vensim (2006) http://www.vensim.com/ (accessed on March 22, 2006).

Waeselynck H, Pfahl D (1994) System Dynamics Applied to the Modeling of Software Projects. Software Concepts and Tools 15(4): 162–176.

Wakeland W, Martin RH, Raffo D (2003) Using Design of Experiments, Sensitivity Analysis, and Hybrid Simulation to Evaluate Changes to a Software Development Process: A Case Study. In: *Proceedings of 4th Process Simulation Modelling Workshop (ProSim)*, Portland, USA.

Wernick P, Hall T (2004) A Policy Investigation Model for Long-Term Software Evolution Processes. In: *Proceedings of 5th International Workshop on Software Process Simulation Modeling (ProSim)*, Edinburgh, Scotland, pp 149–158.

Williford J, Chang A (1999) Modeling the FedEx IT Division: A System Dynamics Approach to Strategic IT Planning. Journal of Systems and Software 46(2/3): 203–211.