Chapter 7
Simulation-Based Software Engineering

Oryal Tanir

Abstract Software Engineering is the application of methodical principles to the planning, design, development, testing, implementation, and maintenance of software-based systems. Each phase of the Software Design Life Cycle (SDLC) addresses a different set of problems, commencing from an abstract need with the eventual goal of producing a stable working solution. To accomplish this, many different tools and techniques may be employed, from project management planning estimators to automated code testers. However, a specific tool-simulation—has found its way into almost every phase of the SDLC. As a general-purpose technique, it can be invaluable for assessing complex multifaceted solution spaces early on during the planning and design phases in a cost-effective and timely manner without the need for physically deploying possible design alternatives. A major and often overlooked element in the design of a new complex solution is the impact on the business and information technology processes. Simulation can help assess these impacts along with any process redesign that may be required. This chapter addressed these and other applications of simulation in Software Engineering.

Keywords Domain driven design · Software engineering · Synthesis · Governance models · Holistic view · Process simulation

7.1 Introduction

Software engineering, in comparison to other engineering disciplines, is relatively young. However, the field has matured rapidly as the demands and complexities in the market have grown. Simulation, in contrast, has a longer history; however, its use in software engineering was initially limited to ad hoc models. Over time, the synergy between the two disciplines has improved significantly to the point where

O. Tanir (✉)
Office of the CIO, McGill University, Montreal, Canada
e-mail: oryaltanir@gmail.com

S. Mittal et al. (eds.), *Guide to Simulation-Based Disciplines*,
Simulation Foundations, Methods and Applications,
DOI 10.1007/978-3-319-61264-5_7

simulation is applicable in a systematic fashion and can benefit the software engineering processes in many ways.

This section will present a brief overview of software engineering and the rational for applying simulation within the disciple. A holistic view of software engineering and application of simulation will describe the key uses of simulation. A model-driven design approach will present another major context for simulation in software engineering. The conclusions will summarize the major concepts and provide insight into some of the future directions of simulation and software engineering.

7.2 Software Engineering

Software engineering is a broad discipline, which spans and borrows practices from multiple areas such as Computer Science, Engineering, Management and Behavioral Sciences. Many definitions help qualify what software engineering is, however for the purposes of this chapter we will use the definition:

"Software engineering is that part of systems engineering that deals with the systematic development, evaluation, and maintenance of software" (Endres and Rombach 2003).

Intuitively we are dealing with a complex function, which (1) takes as its input processes, technology and resources, and (2) either optimizes or outputs a "good-enough" product that is (3) based upon market constraints (time, cost, quality). Hence, there are many layers and concerns related to software engineering—and much research has helped evolve the practice.

Some of the major concepts that are presented in this chapter and are dominant within a software engineering activity are briefly defined below.

7.2.1 The Software Design Life cycle

A key concern for any company is the choice of the software design life cycle (SDLC) model they will utilize. The SDLC describes how the software is planned, designed, implemented, and maintained. Numerous SDLCs exists such as waterfall, iterative design, spiral, agile and others. Based on the particular SDLC that is under consideration, the use of simulation will vary as it adheres to specific parts of processes that are part of the SDLC.

7.2.2 Governance Frameworks

Many software engineering activities rely upon established frameworks to ensure proper governance and oversight of activities. For example, ITIL provides a set of practices for Information Technology Service Management (ITSM)—which is broadly applicable for any software engineering organization. Other frameworks exist for particular industry domains.

7.2.3 Roles

Software engineering is not a purely technology-oriented discipline (see the holistic section later). It involves many different roles to product the final product and this aspect needs careful consideration.

7.2.4 Project Management

Project management practices work in parallel to software engineering ones during the course of a project in alignment to the SDLC. Hence some of the concerns for software design are attributable to project management ones.

7.3 Rationalization of Simulation and Software Engineering

Given, the numerous and different type of challenges that can be encountered in software engineering, it is not surprising that a tool such as simulation can be leveraged to address many of these problems. However, organizations may often find themselves in the position of justifying the use of simulation (which can be a costly affair if the right skills and resources are not available) within a project. This section reviews some of the fundamental decision points to adopt simulation within the software engineering domain. Many of these arguments will be seen later in the chapter as addressed through the application of simulation.

7.3.1 The Cost of Software Defects

One argument frequently seen in industry against the use of simulation within a software project is cost. There can be a prevailing view that the use of simulation will significantly add to the cost of a project and possibly hamper the delivery time. Hence, in this scenario a project manager who would like to avoid cost overruns and complete the project in a timely manner views simulation as a roadblock. In the context of large software projects, this argument is flawed. Many studies have shown the cost effectiveness of tools such as simulation, which can uncover design flaws early on in a project. In particular, Barry Boehm conducted extensive studies on multi-industry software projects to understand when they failed (Boehm 1981). Many other studies confirmed the fundamental finding of the research, which has since been termed as "Boehm's law":

> "Errors are most frequent during the requirements and design activities and are the more expensive the later they are removed."

Although intuitive, the consequences of this statement are illustrated in Fig. 7.1. The figure's basis is data gathered from typical software development projects and depicts Boehm's law in action.

The graph, constructed from empirical data, shows that the cost of fixing errors later in the software cycle (i.e., in the maintenance phase) as compared to early on (i.e., in specification) is several magnitudes costlier. The magnitude of the cost can vary dependent upon the type of software, available skill-sets, organizational maturity and many other factors. Different studies have shown that the cost impact can vary from a linear to exponential one (Hait 2003; Boehm and Papaccio 1988). In addition, another problem arises when defects are uncovered later in the project phase: the may not have enough left in its budget to fix the unforeseen defects (Saultz 1997).

The important idea here is that it is prudent and cost-effective to resolve design issues as early on as possible. Software code test tools help address issues related to coding, but tools such as simulation, which utilize concepts that are more abstract, can address design level issues early on.

The arguments presented above are not always sufficient to justify the use of simulation for a given project. There are cases where simpler and more cost-effective approaches are economically or practically more feasible. For example, in some cases, mathematical models or heuristics (both of which can be leveraged through commonly used tools such as spreadsheets), may be applicable. Mathematical models usually require some basic assumptions or constraints to be in

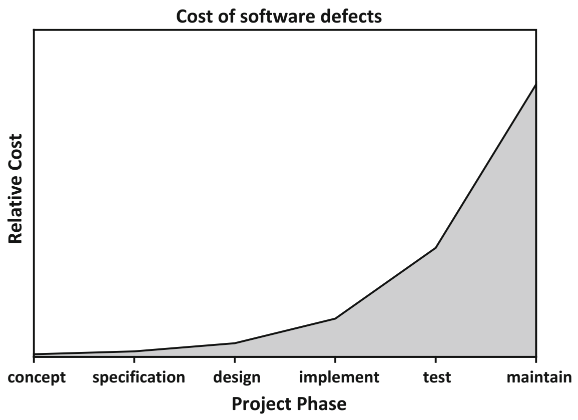

Fig. 7.1 Illustration of Boehm's law

place before they are applicable to the problem. Such models can be applied to solve issues around contained, less complex and small software systems in a timelier manner than simulation. There is however, another reason one may not use simulation: the organization may not be able to obtain the necessary specialized resources to create, simulate, and analyze the results. There may be many reasons for this. For example, the organization may not have the funds or budget to support the work or the skill-set may not be available in the regional market. These are all factors that can influence the use of simulation in a software engineering project.

7.3.2 Business Impacts

A frequently overlooked aspect in the engineering of software is the impact on the business or end user. Software is designed against a set of requirements, however most of the time the impact on the existing processes is under emphasized or overlooked which incurs additional costs to the project due to the underestimation of the cost of change. Areas particularly vulnerable are

- processes (undocumented modification or replacement of existing processes),
- operations (miscalculation of the type of resources needed to manage the new software),
- training (miscalculation of effort needed to use software),
- and supply chain management (inadequate understanding of the full end-end integration of the software with the business).

7.3.3 Project Planning

In many cases, projects are not undertaken in isolation. The project may be part of a program composed of many other projects and understanding the dependencies between projects and the technologies that are impacted can become an exasperating problem. In addition, the timely scheduling of resources for different phases of projects is crucial for a successful and cost-effective endeavor. Simulation can be utilized in this case to evaluate different scenarios and roadmaps.

7.3.4 Time to Market

The application of simulation can also help understand the impact of time to market of the software product. In many industries, a certain window of opportunity for a software release exists, after which expected returns from the market begin to diminish. Simulation can provide indispensable insight to identify these opportunities and the various scenarios, which are both favorable and unfavorable to a software release.

7.4 Holistic View

At its core, software engineering is the business of creating a software product. This is not a simple or straightforward task for a typical project. It involves collaboration and cooperation between many different stakeholders. To accomplish the task, frameworks have been introduced to ensure the best practices and required functions are engaged at the right times. For example, ITIL provides a set of practices for ITSM, and TOGAF a framework for architecture practices (TOGAF 2005).

However, software engineering concerns are beyond just the technology of the final product. There are distinctive issues related to the business, technology, people, and governance as shown in Fig. 7.2, which play a role in a successful software practice. In each case, the application of simulation can help to ensure a successful design. This section will examine the role simulation plays within these broad categories.

7.4.1 Business Concerns

Business concerns encompass financial- and process-related issues. Within the scope of finance there are business case (expected value of the software) and budgetary (total cost of the software) issues. In contrast, process engineering has broader implications for simulation.

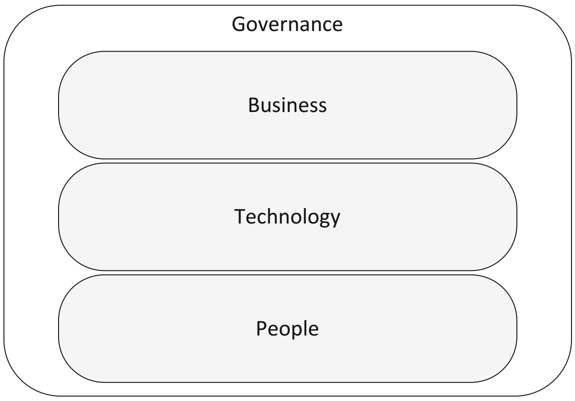

Fig. 7.2 Software engineering concerns

7.4.1.1 Financial Models

Within software engineering, financial models justify the direction of the design and viability of the project. Hence, they are typically applicable in the early planning stages of a software's life cycle. A frequently used tool is a Monte Carlo based risk simulator. Users can gain insight on the impact of design decisions and leverage the quantitative analysis for decision-making to reduce the overall risk.

For example, Monte Carlo simulation can generate statistical outcomes for specific design choices for a project and identify the respective cost implications. Then, the decision makers can make more educated project choices with an understanding of the risk that they will incur.

Another application is the use of such simulators to justify the business case for a software project. In this example, possible outcomes are the expected returns on the investment (revenue, customer satisfaction measures, and market share) to support the decision of undertaking the design commitment.

The applicability of financial simulations is mostly early in the design life cycle where access to precise data may be problematic; hence, their usefulness is dependent upon the accuracy of the available information for the models. If the margin of error of the input is high, the effectiveness of the simulated results will be questionable. However, if the model is maintainable and augmentable with new information, it can also be more useful in latter stages of a project to answer similar financial concerns.

7.4.1.2 Process Simulation

Process modeling and the subsequent activity of process simulation is a mature and common practice in software engineering with a substantial history of research and successful applications (Zhang et al. 2011; Bai et al. 2011). This is due to the fact that a complex software engineering endeavor will typically affect a multitude of business process(es). For example, new software may require a different level of interaction than its predecessor, or if a manual process will undergo some form of automation, the executable sequence of tasks may require modification (to adapt to the automated workflow). It is often difficult to make good design decisions around the process that accompanies a software product. It is also difficult to persuade the stakeholders of the benefits of a process change. These are areas where technicians successfully leverage simulation.

Process simulation is a generic enough practice that any general-purpose simulation language can be utilized. However, there is standard notation available too. For example, the Business Process Modeling Notation (BPMN) provides a standard representation for processes (White 2008). Many tools that support modeling using BPMN also have some form of simulation support. The advantage of using a standard enables the models to be more widely consumable and understandable by the business stakeholders. Hence, it can be an effective communication tool when persuading them of the benefits of a process design. They are intuitive and require knowledge of BPMN—which is not difficult to acquire in the business analysis market. However, BPMN tools use simulation in a limited fashion—mostly supporting visualization of flow in a process—that limits the potential benefits that are usually available in simulators that are more sophisticated. The typical strengths of general-purpose simulation environments such as powerful animation (visualization), statistical analysis and support of experimentation, and a rich set of random distribution support is rarely available in a BPMN tool. The drawback of such simulation environments however, is the need for specialized resources and training to use the environments, which may be beyond the means of restricted budgets. This tradeoff needs careful consideration when deciding upon the appropriate toolset to use.

7.4.2 Technology Concerns

Technological concerns relate to how simulation supports users develop the technical product during a design and implementation phase of a project. The tools common to this phase have similar characteristics to the target implementation. Similar to the business concerns, standards do prevail in many cases.

For example, UML (Unified Modeling Language) is widely adopted to model many software systems—especially in the object-oriented space (Fowler and Scott 2001). UML borrows many concepts prevalent in other areas of computer science and is amicable to the use of simulation. An example of this is behavioral simulation; where the interactions of various software elements can undergo simulation before physical code creation or test. In such cases, simulation identifies important design constraints such as

- The dependencies between different software components based upon their interactions. Discrete-event simulation can execute the flow between software objects to determine bottlenecks based upon the message flowing between each.
- Detection of redundancies is possible but often difficult. Simulation models the flow and specialized test tools detect patterns, which may indicate duplicate behavior in the design.
- Reachability analysis of behavioral constructs such as methods by simulation or simulation-based petri-nets. In such cases all-possible execution paths of the code is simulated which permits detection of code that is not reachable (dead code) or other problematic code behavior such as infinite loops.

To enable these types of analyses, simulators work in close conjunction with the design tools. For example, if the design utilizes UML as a model notation, tools which support this, may incorporate (among others) state-chart simulators. Using a state based simulator, all possible interactions between states can be exhaustively generated and virtually tested to ensure that the overall software behavior is compliant to its specifications.

When software is to be part of an embedded system, it is cost-effective to simulate its functionality and physical characteristics (such as performance and latency) before the fabrication process. Simulation is a technique that is an integral part of the toolsets used in this process as well [such as VHDL or Verilog simulators] (Navabi 2007).

Another standard that is usable at this stage is the Business Process Executionable Language BPEL. BPEL foundation is XML and web services and is a process oriented executable language, primarily used in web based integration and design. It fits well in organizations that have adopted a Service-Oriented Architecture (SOA) approach to software design and integration. Major software vendors support BPEL and consequently their tools have BPEL simulators to aid in the design of web services. A major benefit of simulation at this stage is the ability to simulate orchestration or choreography

- Orchestration of web services implies the use of a central web service, which systematically requests services from other web services and then generates some sort of output or result.
- Choreography of web services does not require a central control. In contrast, each service accepts and sends messages to a limited set of services—their combined interaction or "choreography" results in the desired overall behavior.

Constructing and integrating web services in the above manner becomes a difficult task as the number of webs services increase and the dependencies between each and existing services becomes difficult to manage. The BPEL process models define these interactions—since their interactions can readily be described in terms of processes. Once the BPEL model creation is complete, essentially describing the overall behavior, the simulation activities will:

- Validate the flow of information between web services, which the BPEL processes depict. This validation ensures that the logical flow within each service contributes the correct sequence of activities that will generate the overall process behavior.
- Create an impact analysis of all existing and proposed web services. Rather that detecting interaction problems between services in the field, simulation of their BPEL counterparts permits design engineers to detect potential design issues in the lab. The simulation activities can uncover
  - performance impacts (some service may be a bottleneck) due to too many dependencies or poorly designed logical flow in the service,
  - logical flaws (a process is not flowing as expected),
  - reliability concerns (the failure of certain services may be critical to many other system processes),
  - and inefficient services (too many calls and time spent on a service may indicate it needs to be decomposed to simpler ones).

7.4.3 People Concerns

People concerns are part of the project management role; however, they affect the success of a software engineering project significantly. In particular, planning and estimating the capacity to perform work at different stages in the software life cycle can be difficult. The main tool used by managers is still spreadsheets and Gantt charts; however, simulation has made headway in capacity planning and estimation.

General-purpose simulation tools are dominant in this practice space. Some vendors do provide a combination of project management and simulation capabilities; however, the simulation features are mostly cosmetic or primitive. The functionality can improve in the future as the market for such features increase.

Current general-purpose simulation techniques use a combination of process simulation and resource optimization scenarios. Main use cases for this are

- Decision-making for multiple project and resource trade-offs: There can be cases where different planned projects compete for the same resources at different time lines. Optimization or near-optimization of timelines and resources across multiple projects is a difficult task which simulation is well suited to perform. The constraint is that simulations in this area require specialized skill-sets, which may be too costly or difficult to acquire.
- Capacity planning within a project: Simulation can produce capacity scenarios from common constrains such as hourly wages, scheduling rules, resource availability, skill-set modifiers, and task times (based on heuristics or statistical data).
- Automation versus manual work: In many cases there may be concerns with the trade-offs between automation of certain tasks or managing them manually. Sometimes referred to "people versus technology" scenarios, simulation can provide insight on cost and time implications.

7.4.4 Governance

Governance, as it relates to simulation use in software engineering, is often an oversight. Some simulation activities may be dispensable and used only for a particular decision point in the overall software design and never again. However, in more mature software teams many of the knowledge acquired through simulation is retained or reused. BPEL-driven simulation is one example where a rich knowledgebase of models develops over time. The data and accuracy of the simulations become better with the addition of new models. Governance of the models becomes a necessary part of the software design life cycle. Some common concerns to address are

- Which role is accountable for the model? In cases where the simulation is ad hoc, then it may be the simulationist or their manager. However, for environments that reuse simulation objects, then a role is required to ensure model compliance with the organization's rules and principles. This is usually part of an Enterprise Architecture role, but can also be the accountability of a technology or service manager.
- Which role approves the model? A governance process is required to ensure that the principles related to the design and output of the model are valid. This is typically different from the accountable resource above.
- Terms of engagement of simulation or the conditions when simulation is used and its expected outcomes. It is very important that the expectations of a simulation exercise are the same for all the stakeholders. If no clear definition and boundaries are set for the simulation activity, it can grow or not meet expectations.

The above are typical of software engineering governance issues and it makes sense to utilize similar governance practices for the simulation practice within the software engineering framework. Some important artifacts that need to be defined are

- Model design principles: These are basic principles to abide by when constructing models. They will represent the adopted notation and basic guidelines as well as design patterns when constructing a simulation.
- Data principles: These are basic rules to ensure that data that is used in simulation models is vetted properly and it is handled and interpreted correctly by the modeler. Rules for selecting the correct random distribution which fits a given data set is an example of this.
- Simulation execution principles: The rules determining how long to run simulations and iterations of experiments are common concerns.
- Requirements for a simulation exercise: There can also be guidelines for establishing is a certain project is suitable to undergo a simulation exercise. Concerns such as the quality of the data, complexity of the system under consideration and correct level of expectations are typically addressed.
- Roles and responsibility charts: The expected resources and responsibilities for the simulation project needs to be defined and allocated.
- Decision-making principles or process: The decision-making principles or "rules of engagement" solidify when key decisions resulting from simulation outcomes are made.
- Escalation procedures: A mechanism needs to be in place to resolve issues quickly (i.e., lack of data, resources, time).

As part of the governance, the validity of the simulation studies also needs close examination and understanding. Studies in various industrial fields have shown that the validity of simulation results within a software engineering activity can be quantified and its risks mitigated (França and Travassos 2015). Hence the body of research in this area can help support practitioners successfully compete their simulation ventures.

7.5 Domain-Driven Software Design

Domain-driven software design is a model-based approach in software engineering and simulation is an intricate component at different stages of the methodology. The typical previously mentioned software engineering concerns are applicable here as well. However, the model-driven approach lends itself to model reuse (which permits the creation of a more permanent simulation expertise) and subsequently the application of simulation in this case is methodological and systematic.

There are variations of the approach, but the high-level elements (not all of which need to be used—depending upon the methodology) are illustrated in Fig. 7.3.

The major components are

- **Domain model**: The starting point is the capture of abstract ideas in a domain model. Such models provide a domain specific language or notation suitable to represent the structure and behavior of the design concepts that are of importance (Erdogmus and Tanir 2002). The model must be familiar and versatile enough so that the user captures and validates ideas quickly before undertaking any detailed design decisions. Representations at this level are the high-level specifications for the conceptual system. Successful environments will have strong visual user interfaces to improve productivity and reduce the learning curve. When using simulation, the selection of the appropriate domain model is important. As a requirement, the model needs to support the simulation language or formalism it will be utilizing.
- **Model Checker**: A domain model often relies on a model checker. Such functionality is composed of established formal verification techniques to ensure a sound basis for the model that is developed. A formal verification ensures that the downstream steps to follow will be less prone to errors due to inconsistent representation of model elements.

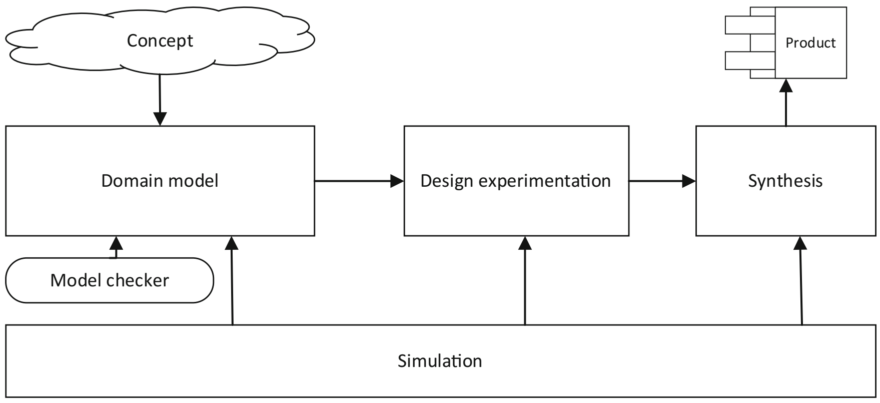

Fig. 7.3 Domain-driven software design components

While model checkers use formal verification techniques, some may employ simulation when formal techniques are not feasible. This can occur in cases where the domain model uses a nonformal language and cannot be validated using such techniques (Tanir et al. 1996). This is often the case when the language supports very abstract concepts to allow a versatile user experience. In such instances, model objects may not bind to a specific behavior or structure early in the design, but missing details populate and complete the missing pieces as the design progresses.

Simulation can bridge the conceptual gap by using semi-formal notations to provide statistics-founded analysis. For example, high-level, colored, or statistical petri-net based formalisms (Haas 2002) are often applicable using a combination of traceability analysis and token simulations to validate the structure or behavior of the model (Tanir and Erdogmus 1999).

- **Design Experimentation**: Experimentation can imply many types of activities. However, it encapsulates those that permit
  - Design space exploration: Tools to support analysis of alternative designs.
  - What if analysis: Such tools will let users change parameters and structure and compare outputs or outcomes.
  - Scenario decision-making tools: Statistical support and directed scenario analysis leverage mathematical techniques to validate scenario outcomes to guide decisions to optimal or near-optimal design choices (Miranda 2002).

Many of the tools employed at this stage will utilize simulation as the principle method of execution of models and comparison of multiple designs. The tools are either specialized or general-purpose and depend upon the design space that is under consideration.

- **Synthesis**: The eventual objective of the modeling approach is to produce executable code from high-level specifications. Synthesis tools accomplish this task. The high-level model that has been verified and validated through model checkers and simulation can now be "synthesized" into code. Synthesizers are technology specific and require a design represented in a formal specification language particular to a domain. For example, Java code synthesizers may require a UML (Universal Modeling Language) based model, whereas an embedded application will use a synthesizable VHDL (Very high-speed integrated circuit Hardware Description Language) model.

The synthesis stage requires a set of different technologies to accomplish the tasks. While most of these are beyond the scope of this chapter, simulation based tools are often part of the synthesis package (Tanir 1997). Synthesis implies a transition from a semi-formal notation that is prevalent during design experimentation to a structured standard one for the target code. The latter is generally not based on a formal representation, but more on a standard notation. Hence, simulation validates the resulting "synthesized" target model.

There are also new simulation concerns at this stage. For example, simulating latency, cycle times, and probability of failures are part of the validation practices. The notation in the simulators in this case are close to the target formalism (i.e., VHDL or UML) and therefor part of a synthesis software offering. BPEL, which was introduced earlier, can also be part of the simulation and synthesis activities that are related to the design of web services.

- **Software Package**: The Software package is the resulting product. At the minimal it is executable software code, but it will typically also contain supporting software by products such as
  - Automated test cases that validate the functionality of the code against the initial (domain language) specifications or business requirements.
  - Service level agreements that may be part of the requirements of the software.
  - Self-test code can be included for systems that will be synthesiz-able to silicon. This will permit the testing of fabricated compo-nents in a non-intrusive manner.
  - Software documentation that describes the functionality of the code and any changed components (if a history exists).

As can be seen, the use of simulation within software engineering is quite open and applicable across a broad range of activities and domains.

7.6 Conclusion

Simulation use in software engineering has progressed from ad hoc throwaway models to reusable ones. This trend will further improve in the future as more vendors adopt or improve their simulation offerings. General-purpose simulators will still prevail in many software engineering activities since specialized simulators do not meet all the needs across a software design life cycle. Many of the concepts developed in the artificial intelligence domain is now technically and financially feasible to be applied in certain circumstances to utilize simulation and design project in new and novel ways (Elzas et al. 1989). For example, models could potentially adapt to proposed design changes based on design patterns and best practices to aid the experimentation process.

As with any software tool, standardization of the use of simulation tools and the way in which they are integrated into the software engineering processes is important for the success of any software project.

Review Questions

1. What are some strong arguments that can be made to bolster a business case for adopting simulation within a software engineering project?
2. Under what circumstances would simulation not be a feasible choice within the context of software engineering?
3. Define and elaborate upon the basic elements of a holistic view of software engineering.
4. Which area of software engineering has simulation played the most prevalent role and considered a mature practice?
5. What are key deliverables that should be part of a governance process for simulation?

References

Bai, X., Zhang, H., & Huang, L. (2011). Empirical Research in Software Process Modeling: A Systematic Literature Review. 2011 International Symposium on Empirical Software Engineering and Measurement. doi:10.1109/esem.2011.43.

Boehm, B. W. (1981). Software engineering economics. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, Barry W. & Papaccio, Philip N. "Understanding and Controlling Software Costs," IEEE Transactions on Software Engineering 14, 10 (October 1988): 1462–1477.

Elzas, M. S., Ören, T. I., & Zeigler, B. P. (1989). Modelling and simulation methodology: Knowledge systems' paradigms. Amsterdam: North-Holland.

Endres, A., & Rombach, H. D. (2003). *A handbook of software and systems engineering: Empirical observations, laws, and theories*. Harlow, England: Pearson Addison Wesley.

Erdogmus, H., & Tanir, O. (2002). Advances in software engineering: comprehension, evaluation, and evolution. New York: Springer.

Fowler, M., & Scott, K. (2001). UML. Paris: Campus Press.

Haas, P. J. (2002). Colored Stochastic Petri Nets. Stochastic Petri Nets, 385–445. doi:10.1007/0-387-21552-2_9.

Hait, C. (2003). Economic impacts of inadequate infrastructure for software testing: final. Place of publication not identified: Diane Pub Co.

França, B. B., & Travassos, G. H. (2015). Simulation Based Studies in Software Engineering: A Matter of Validity. CLEI electronic journal, 18(1). doi:10.19153/cleiej.18.1.4.

Miranda, E. R. (2002). Computer sound design: synthesis techniques and programming. Oxford: Focal.

Navabi, Z. (2007). VHDL: modular design and synthesis of cores and systems. New York: McGraw-Hill.

Saultz, J., & Kalathil, B. (n.d.). Rapid prototyping of application-specific signal processors (RASSP) approach to meeting 4X. Annual Reliability and Maintainability Symposium. doi:10.1109/rams.1997.570023.

Tanir, O. (1997). Modeling complex computer and communication systems: a domain-oriented design framework. New York: McGraw-Hill.

Tanir, O., Agarwal, V., & Bhatt, P. (1995). A specification-driven architectural design environment. Computer, 28(6), 26–35. doi:10.1109/2.386983.

Tanir, O., Agarwal, V. K., & Bhatt, P. C. (1996). DASE: An environment for system level telecommunication design exploration and modelling. Computer Aided Systems Theory—CAST '94 Lecture Notes in Computer Science, 302–318. doi:10.1007/3-540-61478-8_85.

Tanir, O., & Erdogmus, H. (1999). A Framework for Topological Re-use. In M. Fayad, D. C. Schmidt, & R. E. Johnson (Authors), Implementing application frameworks: object-oriented frameworks at work. New York: Wiley.

TOGAF: version 8.1, Enterprise Edition. (2005). San Francisco: Open Group.

White, S. A., & M. (2008). BPMN: modeling and reference guide. Florida: Future Strategies, Incorporated.

Zhang, H., Jeffery, R., Houston, D., Huang, L., & Zhu, L. (2011). Impact of process simulation on software practice. Proceeding of the 33rd international conference on Software engineering - ICSE '11. doi:10.1145/1985793.1985993.

Author Biography

Oryal Tanir is the Enterprise Architect in the Office of the CIO and professor at McGill University. He has been involved in simulation since 1984 and has practiced both as a practitioner and as a researcher in software engineering since 1986. He has worked in different industries such as telecommunications, travel, and higher education with experience in many facets of IT such as; Simulation, Software Design, Enterprise Architecture, System engineering, and Business Process Management. In his current capacity at McGill University, Dr. Tanir influences the strategies and directions of software engineering within the Higher Education area. His research interests have been concentrated in various topics in simulation and the automated synthesis of complex systems. He has numerous publications in software engineering and simulation and has lead or collaborated in work in this area with McGill University, University of Toronto, Ottawa University, and University of Waterloo. He is also the author of the novel "The Falcon's Arrow." He frequently contributes to the Canadian Research granting boards in different capacities.