Software Process Modeling for an Interactive, Graphical, Educational Software Engineering Simulation Game

Emily Oh Navarro and André van der Hoek
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
emilyo@ics.uci.edu, andre@ics.uci.edu

Abstract

SimSE is an educational software engineering simulation game that uses a unique software process modeling approach. This approach combines both predictive and prescriptive aspects to support the creation of dynamic, interactive, graphical models for software engineering process education. This paper describes the different constructs in a SimSE process model, the associated model builder tool, and discusses the underlying tradeoffs and issues involved in this approach.

1. Introduction

SimSE is an interactive, graphical, educational software engineering simulation game designed to teach students the *process* of software engineering [13]. In traditional software engineering educational approaches, students are exposed to software engineering concepts and theories in lectures, but have limited opportunity to put these ideas into practice in an associated small software engineering project. SimSE aims to fill this gap by providing students with virtual experiences of realistic, large-scale software engineering processes.

SimSE is a single-player game in which the player takes on the role of project manager of a team of developers. As the player manages the process to complete (a particular aspect of) a software engineering project, they can, among other things, hire and fire employees, assign tasks to them, monitor their progress, and purchase tools. Because a visually interesting graphical user interface is considered essential to any successful educational simulation [7], the user interface of SimSE is fully graphical, displaying a virtual office in which the software engineering process takes place. This display includes typical office surroundings, such as desks, chairs, computers, and meeting rooms, as well as information about employees (e.g., productivity, current task, energy level), artifacts (e.g., size, completeness, correctness), customers (e.g., satisfaction level), projects (e.g., budget, time), and tools (e.g., number of users, productivity increase factor). Players use this information to make decisions and take actions, driving the simulation accordingly.

One of the fundamental goals of the SimSE project is to support customization of the software process models it simulates. Real-world software processes vary with different application domains, organizations, and cultures, and therefore SimSE must be able to portray different processes as well. Furthermore, instructors using SimSE may belong to different schools of thought regarding best software engineering practices, and may have different teaching objectives that require different types of models. Therefore, an integral part of SimSE is a software process modeling language with associated tool support.

The educational, interactive, and graphical nature of SimSE imposes three unique requirements upon its process modeling language: First, it must be both *predictive*—allow the modeler to specify causal effects that the player's actions will have on the simulation, and *prescriptive*—support the specification of the allowable next steps the player can take at a given time. Second, it must also be *interactive*, meaning that it should operate on a step-by-step basis, accepting user input and providing feedback constantly throughout the simulation. Finally, it must allow the modeler to specify the *graphical representations* of the elements in the model. However, a survey of existing process modeling approaches revealed that most are either predictive [1, 10] or prescriptive [4, 12], but not both; few are interactive [4, 12]; few support graphics [8, 11]; and none fulfill all of these requirements. The closest fit is the modeling language used in SESAM, another educational software engineering simulation environment [5]. However, despite the fact that the SESAM language is highly flexible and expressive, the model building process is learning- and labor-intensive and requires writing code in a text editor. Furthermore, the user interface for the simulation is text-based so the modeling language has no support for graphics.

SimSE's software process modeling approach combines and refines the applicable features in existing process modeling languages to create predictive, prescriptive, interactive, graphical models for use in SimSE. The remainder of this paper details this modeling approach. Section 2 describes the different components of a SimSE model, the associated model builder tool, and discusses issues and tradeoffs involved in the approach. We con-

clude in Section 3 with our current progress and directions for future work.

2. Approach

2.1 Modeling Constructs

A SimSE model consists of five parts: *Object types* define templates for all objects that participate in the simulation. The *start state* of a model is the collection of objects present at the beginning of a simulation. *Actions* refer to the activities that objects in the simulation can participate in. *Rules* define the effects that actions have on the rest of the simulation. *Graphics* refer to the graphical representations of all objects in the simulation and the layout of the virtual office. The remainder of this subsection discusses each of these modeling constructs in further detail.

Object types. The first step in building a SimSE model is to define the object types to be used in the model. Each major entity participating in the simulation will be an instantiation of an object type. Every object type defined must descend from one of five *meta-types*: Employee, Artifact, Tool, Project, or Customer. Each of these meta-types have very limited semantics in and of themselves, except for where objects of each type are displayed in the GUI of the simulation, and how the player can interact with each type of object. Specifically, only objects descended from Employee and Customer will display overhead pop-up messages during the game, and only objects descended from Employee will have right-click menus associated with them so the player can command their activities.

An object type consists of a parent meta-type, a name, and a set of typed attributes. For each attribute, in addition to the type (String, Double, Integer, or Boolean), the following metadata must be specified: *key* (a Boolean value indicating whether or not this attribute is the key attribute for the object type), *visible* (a Boolean value denoting whether or not this attribute should be visible to the player of the simulation), *minVal* (the minimum value for this attribute – for Double and Integer attributes only), and *maxVal* (the maximum value for this attribute – also for Double and Integer attributes only). Two sample object types, a Programmer Employee and a Code Artifact, are shown in Figure 1. (Note that the format of this example and the examples throughout this paper are shown in a "shorthand" version of the actual SimSE modeling language format, which is XML-like and difficult to read. However, since this language is completely hidden from the user by our model building tools, we have accordingly omitted it from this paper. See Section 2.3 for a more extensive discussion of this issue.)

```
Programmer Employee
{
  name:
    type: String
    visible: true
    key: true
  energy:
    type: Double
    visible: true
    minVal: 0.0
    maxVal: 1.0
    key: false
  productivity:
    type: Double
    visible: true
    minVal: 0.0
    maxVal: 1.0
    key: false
  error rate:
    type: Double
    visible: true
    minVal: 0.0
    maxVal: 1.0
    key: false
  hired:
    type: Boolean
    visible: true
    key: false
}
```

```
Code Artifact
{
  name:
    type: String
    visible: true
    key: true
  numUnknownErrors:
    type: Integer
    visible: false
    minVal: 0.0
    maxVal: boundless
    key: false
  numKnownErrors:
    type: Integer
    visible: true
    minVal: 0.0
    maxVal: boundless
    key: false
  size:
    type: Double
    visible: true
    minVal: 0.0
    maxVal: boundless
    key: false
  percentComplete:
    type: Double
    visible: true
    minVal: 0.0
    maxVal: 100.0
    key: false
}
```

Figure 1: Sample Programmer and Code Object Types.

Start state. Once the object types for a simulation have been defined, the start state for that simulation can be specified. The start state refers to the set of objects that are present when the simulation begins. Each one of these objects must be an instantiation of one of the object types defined for the model, and starting values for all attributes must be assigned. Figure 2 shows sample instantiated objects for the Programmer and Code object types from Figure 1.

```
Object Programmer
  Employee
{
  name = "Roger"
  energy = 0.9
  productivity =
    0.6
  error rate =
    0.3
  hired = true
}
```

```
Object Code Artifact
{
  name = "My Code"
  numUnknownErrors =
    18
  numKnownErrors =
    7
  size = 25600
  percentComplete =
    10.2
}
```

Figure 2: Sample Instantiated Programmer and Code Objects.

Actions. The next part of a SimSE model is the set of actions in which the objects in the simulation can participate. For example, a "Code" Artifact, with one or more "Programmer" Employees and one or more "IDE" (integrated development environment) Tools could participate in a "Coding" action, in which the programmers build a piece of code using an IDE. This example is shown in

detail in Figure 3. As another example, an Employee could participate in a "break" action, referring to the activity of taking a break, during which he or she rests and does not work.

For each action, the following information is specified: a name; one or more participants—roles in the action that can be filled by one or more objects of (a) specified object type(s), an action trigger, and an action destroyer. An action trigger refers to what causes the action to begin to occur in the simulation. Three distinct classes of triggers exist: *autonomous*, *user-initiated*, and *random*. Autonomous triggers specify a set of conditions (based on the attributes of the participants in the action) that cause the action to automatically begin, with no user intervention. For instance, an Employee may automatically take a break when his or her energy level drops below a certain threshold. User-initiated triggers also specify a set of conditions, but include a menu item text string, which will appear on the right-click menu for an Employee when these conditions are met. This menu item corresponds to this action, and when the menu item is selected, the action begins. For example, in the Coding action shown in Figure 3, a menu item with the text "Start coding" will appear on the menus of all Programmer and Tester employees who meet the specified conditions (hired and health level greater than or equal to 0.7) and are not already participating in a Coding action with the potential piece of code. Random triggers specify both a set of conditions and a frequency that indicates the percent chance of the action occurring whenever the specified conditions are met. For the sake of space, in Figure 3 participants in the trigger and destroyer are not shown if there are no conditions attached to them.

```
Action Coding
{
  Participant Coder
  {
    quantity: at least 1
    allowable types: Programmer, Tester
  }

  Participant CodeDoc
  {
    quantity: exactly 1
    allowable types: Code
  }

  Participant IDE
  {
    quantity: at least 1
    allowable types: Eclipse, JPad
  }

  Trigger
  {
    type: User-initiated
    menuText: "Start coding"
    overheadText: "I'm coding now!"
    conditions
    {
      Coder:
        Programmer:
          hired == true
        Tester:
          hired == true
          health >= 0.7

      IDE:
        Eclipse:
          purchased == true
          licenseValid == true
        JPad:
          purchased == true
          licenseValid == true
    }
  }

  Destroyer
  {
    type: Autonomous
    overheadText: "I'm finished coding!"
    conditions
    {
      CodeDoc:
        Code:
          percentComplete == 100.0
    }
  }
}
```

Figure 3: Sample "Coding" Action.

An action destroyer works in a similar manner as an action trigger, but has the opposite effect: whereas a trigger *starts* an action, a destroyer *stops* an action. Destroyers can be of the same types as triggers (autonomous, random, or user-initiated), but have one additional type: *timed*. A timed destroyer specifies a "time to live" value for an action—once an action starts, it exists for a number of simulation clock ticks equal to this value, and is then automatically destroyed. The "Coding" action shown in Figure 3 has associated with it an autonomous destroyer that will cause the action to stop when the code is 100% complete.

Rules. After all of the action types have been defined, the next task in building a SimSE model is to attach *rules* to each action type. A rule defines an effect of an action–how the simulation is affected when that action is active. Two example rules attached to the "Coding" action are shown in Figure 4.

We distinguish three types of rules in a SimSE model: *create objects rules*, *destroy objects rules*, and *effect rules*. As its name indicates, a create objects rule causes new objects to be created in the game. For example, as shown in Figure 4, a "Coding" action might have associated with it a create objects rule that creates a new Code Artifact object with its size and number of errors equal to zero. This would indicate that a new Code Artifact comes into existence as a result of programmers participating in a "Coding" action. A create objects rule is only fired once, at the point when its associated action begins.

In contrast to a create objects rule, the firing of a destroy objects rule results in the destruction of existing objects. For instance, a "Fire" action might have associ-

```
Coding Action Rules
{
  CreateObjectsRule
  {
    createdObjects
    {
      Object Code Artifact
      {
        name =“My Code”
        numUnknownErrors = 0
        numKnownErrors = 0
        size = 0.0
        percentComplete = 0.0
      }
    }
  }

  EffectRule
  {
    Coder:
      Programmer:
        name = // no effect
        energy = this.energy – 0.05
        productivity = this.productivity –
          0.0375
        errorRate = (1 - this.energy) * 0.4
        hired = // no effect
      Tester:
        // etc…

    CodeDoc:
      Code:
        name = // no effect
        numUnknownErrors =
          this.numUnknownErrors +
          allActiveProgrammerCoders.errorRate
        numKnownErrors = // no effect
        size = this.size +
          allActiveProgrammerCo-
          ders.productivity
        percentComplete = (this.size /
          allSEProjectProjects.targetCodeSize)
          * 100
  }
}
```

Figure 4: Sample Rules Attached to the “Coding” Action

ated with it a destroy objects rule that removes an Employee from the game, indicating that they have been fired. Like create objects rules, destroy objects rules are also fired only once, at the start of the action.

An effect rule is the most powerful and expressive type of rule in SimSE. Rules of this type specify the complex effects of an action on its participants’ states, including the values of their attributes and their participation in other actions. For instance, the effect rule attached to the “Coding” action, shown in Figure 4: a) causes the size of the code to increase by the additive productivity levels of all of the programmers currently working on it; b) causes the number of unknown errors in the code to increase based on the error rates of the currently active coders; and c) updates the completeness level of the code. At the same time, it decreases the energy and productivity levels of the coders as they work, and resets their error rates based on their current energy levels. As another example, a “Break” action might have an effect rule attached to it that: a) increases the energy of an employee; and b) deactivates the employee from all other actions in which he or she is currently participating for the duration of the “Break” action. Unlike create objects rules and destroy objects rules, an effect rule is fired once every clock tick that its associated action is active.

In specifying an effect, the modeler can use a number of different constructs, including participant attribute values, the number of participants in an action, the number of other actions a participant is involved in, the time elapsed in the simulation, random values, user inputs, numbers, and mathematical operators.

Graphics. Because the user interface of SimSE is fully graphical, graphics are an integral part of our modeling approach, and are woven throughout the different parts of a model. For instance, each action trigger and destroyer can have associated with it a string of text to appear in pop-up bubbles over the heads of that action’s Employee participants when the action either begins (trigger) or ends (destroyer). For example, “I’m coding now” may appear over the head of all “Coder” participants when they are beginning a “Coding” action (see Figure 3). Likewise, effect rules can have specified with them *rule inputs* that cause a dialog to appear during the simulation, prompting the user for input. For example, an effect rule attached to a “Give Bonus” action might prompt the user to enter the amount of the bonus they wish to give. In addition to these graphical aspects woven throughout the model, specific images must be assigned to each object in the start state, and the layout of the “office” must be specified. Because these graphical features of the modeling approach are rather trivial, and consist of simply assigning image filenames to objects and specifying coordinates for images, an example of these is omitted from this paper.

2.2 Discussion

In designing SimSE’s software process modeling approach, it became apparent that some tradeoffs would have to be made. We acknowledge that it is not as generic or flexible as some general purpose modeling and simulation approaches [2, 8], or even domain-specific languages designed specifically for modeling software processes [6, 9]. However, aside from the fact that none of these approaches met the unique needs of our educational game domain, we felt that such a level of genericity and flexibility was unnecessary for our purposes. The process by which we designed our modeling approach underscores this: We surveyed the software engineering literature and extracted the widely accepted process lessons and rules that would conceivably go into a SimSE model, and then designed the modeling approach with these rules

in mind. Although they include a wide range of different types of phenomena, from management issues, to organizational behavior theories, to corporate culture, to the traditional software engineering theories (e.g., Brooks' Law [3]), all of the rules that we have collected thus far can be modeled and simulated in SimSE. We will continue to gather more rules, see how well they can be modeled in SimSE, and refine the modeling approach accordingly.

We also believe that the educational nature of SimSE makes a low-level modeling approach inappropriate—too much detail and realism may overwhelm the user and distract from the lessons that the model is trying to teach. Another danger is that lessons may get expressed at too low of a level and not be brought out obviously enough in the simulation to be educationally effective [7]. At the expense of realism, effects need to be obvious and "over the top" at times in order to effectively illustrate and enforce the concepts being taught. Finally, although limited in some ways, the specificity of our modeling approach promotes a simplicity that makes it more usable and easier to learn than some more generic approaches.

2.3 Model Builder

To facilitate a high-level, rapid, and relatively easy model building process, we have developed a model builder tool. This model builder completely hides the underlying modeling language from the modeler, and provides a graphical user interface for specifying the object types, start state, actions, rules, and graphics for a model. Figure 5 shows the user interface for the object builder, the part of the model builder that supports defining object types. For the sake of space, the interfaces for the other parts of the model builder are not shown, but they are similar in appearance to the object builder in that they all facilitate building a model using buttons, drop-down lists, menus, and dialog boxes—no programming is required. Once a model is specified, the model builder then generates Java code for a complete, executable, customized simulation game based on the given model.

Although the model builder removes the inherent difficulties of a programming language (e.g., syntax), we recognize that the difficulty of collecting software engineering phenomena and rules and translating these into SimSE actions and rules still remains. To assist with this, we plan to provide example models, with accompanying documentation, as a part of SimSE so that instructors can use and adapt these models for their own purposes, rather than write one from scratch. We anticipate that these models, along with the model builder, will be valuable tools for instructors, who generally do not have a lot of

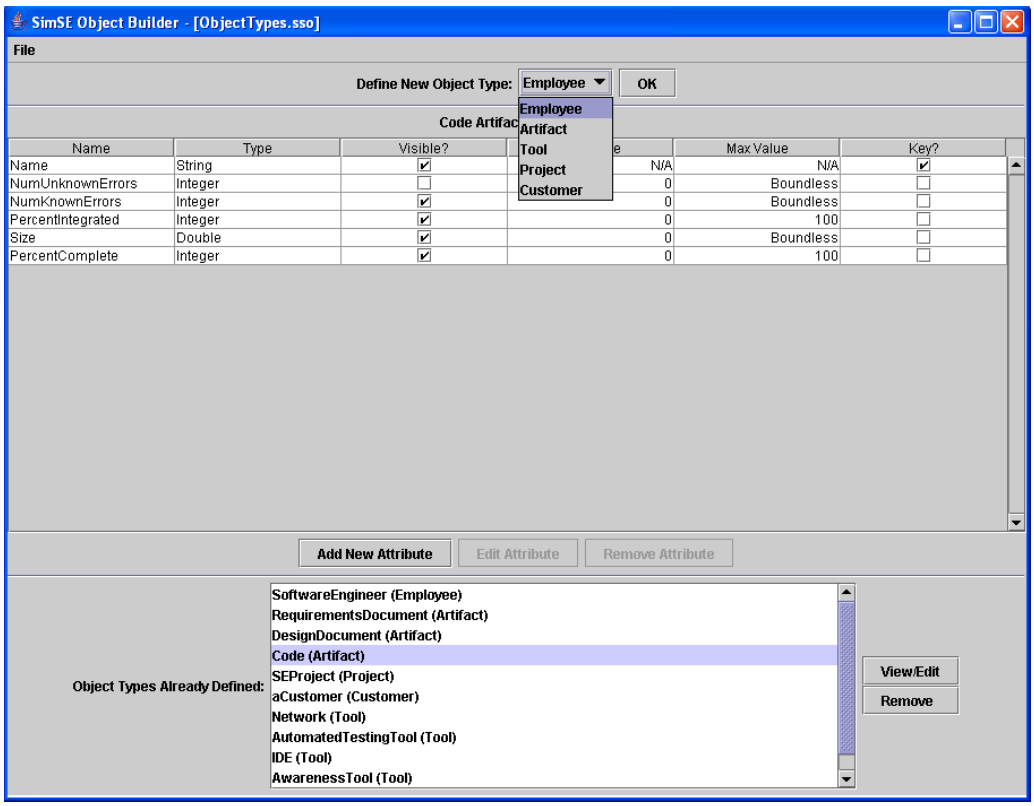

Figure 5: Object Builder User Interface.

time, and may not have a great deal of skill in, or desire for programming simulation models.

3. Conclusions and Future Work

The educational, graphical, and interactive nature of the SimSE software engineering simulation game necessitates a rather unique modeling approach. Our new predictive and prescriptive modeling language, along with its associated model builder tool, supports the rapid creation of interactive, graphical simulation models for software engineering education. We are currently nearing completion of a first version of SimSE, and, in parallel, are building two initial models: a high-level model in which an overall software engineering process is simulated using a waterfall model and a number of general lessons about the process as a whole are taught, and a second, more detailed model that teaches the roles and regulations of the inspection process by making the student organize and perform a code inspection. We plan to continue to build different types of models to demonstrate both specific situations, such as the roles of various forms of testing by making a student deliver high quality code, and overarching practices, such as the tradeoffs among different lifecycle models by letting the student vary the model by which to develop a product. Finally, we plan to evaluate the teaching potential of SimSE and the models we have developed by conducting experiments involving undergraduate computer science students at UC Irvine.

4. URL

More information about SimSE is available at:

http://www.ics.uci.edu/~emilyo/SimSE

5. Acknowledgements

We thank Ethan Lee, Calvin Lee, and Beverly Chan for their contributions to the implementation of SimSE.

Effort partially funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation. Effort also funded by the UC Irvine CORCLR program.

6. References

1. Abdel-Hamid, T. and S.E. Madnick, *Software Project Dynamics: an Integrated Approach*. 1991, Upper Saddle River, NJ: Prentice-Hall, Inc.
2. Birtwistle, G.M., *Discrete Event Modelling on Simula*. 1979, Houndmills, Basingstoke, Hampshire: MacMillan Education Ltd.
3. Brooks, F.P., *The Mythical Man-Month: Essays on Software Engineering*. 2 ed. 1995, Boston, MA: Addison-Wesley. 336.
4. Cass, A.G., et al., *Little-JIL/Juliette: A Process Definition Language and Interpreter*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000: Limerick, Ireland. p. 754-757.
5. Drappa, A. and J. Ludewig, *Simulation in Software Engineering Training*, in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 199-208.
6. Emmerich, W. and V. Gruhn, *FUNSOFT Nets: A Petri-Net Based Software Process Modeling Language*, in *Proceedings of the Sixth International Workshop on Software Specification and Design*. 1991, IEEE Computer Society. p. 175-184.
7. Ferrari, M., R. Taylor, and K. VanLehn, *Adapting Work Simulations for Schools.* The Journal of Educational Computing Research, 1999. **21**(1): p. 25-53.
8. Howell, F. and R. McNab, *simjava: a Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling*, in *Proceedings of the First International Conference on Web-based Modelling and Simulation*. 1998, Society for Computer Simulation: San Diego, CA.
9. Kaiser, G.E., S.S. Popovich, and I.Z. Ben-Shaul, *A Bi-level Language for Software Process Modeling*, in *Proceedings of the 15th International Conference on Software Engineering*. 1993, ACM. p. 132-143.
10. Lakey, P., *A Hybrid Software Process Simulation Model for Project Management*, in *Proceedings of the 6th Process Simulation Modeling Workshop (ProSim 2003)*. 2003: Portland, Oregon, USA.
11. MAPICS Inc., *AweSim*, 2004: http://www.pritsker.com/awesim.asp.
12. Noll, J. and W. Scacchi, *Specifying Process-Oriented Hypertext for Organizational Computing.* Journal of Network and Computer Applications, 2001. **24**(1): p. 39-61.
13. Oh, E. and A. van der Hoek, *Adapting Game Technology to Support Individual and Organizational Learning*, in *Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering*. 2001, Knowledge Systems Institute: Buenos Aires, Argentina. p. 347-354.