Software Process Modeling for an Educational Software Engineering Simulation Game

Emily Oh Navarro and André van der Hoek
Department of Informatics
Donald Bren School of Information and Computer Sciences
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, introduces the associated model builder tool, describes how we built an initial model of a waterfall process, and discusses the underlying tradeoffs and issues involved in our approach.

Keywords: Software process, Software process modeling and simulation, Software engineering education, Educational simulation games

1. Introduction

Simulation is a powerful educational technique that is frequently used in a wide range of educational domains to provide students with valuable hands-on experience of situations that would otherwise be too difficult or risky to practice in reality. Educational simulations provide students with the freedom to experiment with alternative approaches and repeat experiences, gaining a deeper insight with each simulation run. Although software engineering process education is one domain in which simulation would be an ideal educational tool, the field has yet to fully leverage this approach. There have been a few exceptions that have identified promising avenues (Collofello, 2000; Drappa and Ludewig, 2000; Nulden and Scheepers, 2000; Pfahl, et al., 2004; Sharp and Hall, 2000) , but have not yet fully pushed the boundaries of simulation in software engineering education. Namely, these approaches have been limited in one or more of the following areas: interactivity, customizability, and/or graphics.

To address these issues, we have developed SimSE, an interactive, graphical, educational software engineering simulation game designed to teach students the *process* of software engineering (Navarro and van der Hoek, 2004). SimSE ad-

dress the large gap that exists in traditional software engineering educational techniques—students are exposed to several software engineering concepts and theories in lectures, but have limited opportunity to put these ideas into practice in a typically small software engineering project. SimSE aims to fill this gap by providing students with virtual experiences of participating in quasi-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 (Ferrari, et al., 1999), the user interface of SimSE is fully graphical, displaying a virtual office in which the software engineering process takes place (see Figure 1). 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). Employees "communicate" with the player through pop-up speech bubbles over their heads, in which they inform the player of important information, such as when they have started or completed a task, when a random event has occurred, or to express a response to one of the player's actions (e.g., quitting because the player has been working them too hard). 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 allow customization of the software process models it simulates. Real-world 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 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 any given time. Second, it must 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. Our survey of existing process modeling approaches revealed that most are

either predictive (Abdel-Hamid and Madnick, 1991; Boehm, 2000; Lakey, 2003) or prescriptive (Cass, et al., 2000; Noll and Scacchi, 2001), but not both; few are interactive (Cass, et al., 2000; Noll and Scacchi, 2001); few support graphics (Howell and McNab, 1998; MAPICS Inc., 2004); and none fulfill all of these requirements. The closest fit is the modeling language used in SESAM, another educational software engineering simulation environment (Drappa and Ludewig, 2000). 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.

We build on SESAM and other approaches in the software process modeling approach used in SimSE. Our 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. Section 3 summarizes a waterfall process simulation model that we built using our approach, including some of its goals and how they were implemented. In Section 4 we present the model builder tool that is used to define a SimSE model. In Section 5 we discuss issues and tradeoffs involved in the approach. Section 6 provides a brief overview of related work, and we conclude in Section 7 with our current progress and directions for future work.

2. Modeling Constructs

A SimSE model consists of five parts. Figure 2 illustrates the relationships between the different parts of a model. *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 in which objects in the simulation can participate. *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 section discusses each of these modeling constructs in further detail.

2.1 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 descending from Employee and Customer will display overhead pop-up messages during the game, and only objects descending from Employee will have right-click menus associated with them so the player can command their activities.

An object type descends from a parent meta-type, and consists of a name and a set of typed attributes. For each attribute, in addition to the name and 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 this attribute should be visible to the player throughout the game.
- *VisibleAtEnd*: a Boolean value indicating whether or not this attribute should be visible at the end of the game–designed to give further insight to the player about why they received their particular score when such an attribute that was hidden throughout the game is revealed at the end.
- *MinVal*: the minimum value for this attribute (for Double and Integer attributes only).
- *MaxVal*: the maximum value for this attribute (also for Double and Integer attributes only).
- *MinDigits*: the minimum number of digits after the decimal point to display for this attribute's value (for Double attributes only).
- *MaxDigits*: the maximum number of digits to display (also for Double attributes only).

Three sample object types, a "Programmer" of type Employee, a "Code" of type Artifact, and an "SEProject" of type Project are shown in Figure 3. 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 4 for a more extensive discussion of this issue.

2.2 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—no default

values are automatically given. Figure 4 shows sample instantiated objects for the “Programmer”, “Code”, and “SEProject” object types from Figure 3.

2.3 Actions

The next part of a SimSE model is the set of actions in which the objects in the simulation can participate. For example, to model a situation in which programmers are building a piece of code using an IDE (integrated development environment), one would create a “Coding” action, in which the participants include a “Code” Artifact, one or more “Programmer” Employees and one or more “IDE” Tools. This example is shown in detail in Figure 5. As another example (not shown), an Employee of any type 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:

- *Name*: name of the action.
- *Visibility*: whether or not the player should be able to see that the action is occurring, and, if true, a textual description of that action to display in the game’s user interface.
- *Participant(s)*: roles in the action that can be filled by one or more objects of one or more possible object types.
- *Trigger(s)*: 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 5, 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, for testers, health level greater than or equal to 0.7). When this menu item is selected by the player, the action will begin. Random triggers introduce some chance into the model, specifying both a set of conditions and a frequency that indicates the likelihood of the action occurring whenever the specified conditions are met. For instance, a “Quit” action might have a 75% chance of occurring whenever an

Employee's energy level is below 0.2, meaning that employees are likely to quit when they have been worked too hard, but may not always do so.

- *Destroyer(s)*: 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 5 has associated with it two destroyers: an autonomous one that will cause the action to stop when the code is 100% complete, and a user-initiated one that allows the player to make the action cease at any time, in order to reallocate their employees to other tasks.

Triggers and destroyers also have two additional pieces of information that must be specified: *priority* and *game-ending*. The priority of a trigger or destroyer refers to the order in which that trigger/destroyer will be checked, and, if all conditions are met, executed. All triggers in a model are prioritized in relation to all other triggers in that model, and all destroyers are likewise prioritized in relation to all other destroyers. In the "Coding" action shown in Figure 5, the autonomous destroyer ("autoDestroyer") has priority 10, while the user-initiated destroyer ("userDestroyer") has priority 11, indicating that when a "Coding" action is occurring, the conditions for the autonomous destroyer will be checked first. This sequence is specified so that if the code is 100% complete, the action will cease (as a result of the autonomous destroyer) before the user-initiated destroyer is checked and the "Stop coding" choice is put on an Employee's menu. Any trigger or destroyer can also be designated as *game-ending*, meaning that when that trigger or destroyer occurs, the game will be over. A game-ending trigger or destroyer must have exactly one of its participant's attributes specified as the *score* attribute, indicating that the value of that attribute at the time that trigger or destroyer is executed will be given as the player's score. A typical game-ending trigger might be attached to a user-initiated "DeliverProductToCustomer" action in which the score is designated as the "score" attribute of an "SEProject" participant.

2.4 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 6.

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 6, 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 piece of code comes into existence as a result of programmers participating in a "Coding" action.

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 associated with it a destroy objects rule that removes an Employee from the game, indicating that they have been fired.

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 6: (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. 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 in which a participant is involved, the time elapsed in the simulation, random values, numbers, user inputs, and mathematical operators.

In addition to a rule's general type (*create objects*, *destroy objects*, or *effect)*, each rule is also assigned a *timing* type, indicating when and how often that rule will be executed. There are four possible timing types: *trigger*, *destroyer*, *singular*, or *continuous*. A trigger rule will execute only once, at the time the action is triggered, while a destroyer rule will execute once at the time the action is destroyed. A singular rule will also execute once, but during the first clock tick that an action is active (the clock tick immediately after the action starts/is triggered). A continuous rule, on the other hand, will fire once every clock tick that the action is active. Only *effect rules* can be continuous, since there is no need to create the same object multiple times (using a *create objects* rule), or destroy the same object multiple times (using a *destroy objects* rule). Table 1 summarizes these various combinations. In the rules shown in Figure 6, the new Code Artifact is

created once, at the time the action is triggered, since the create objects rule is assigned a trigger timing. Because the effect rule is assigned a continuous timing, however, its expressions are evaluated every clock tick that the action is active, and the “Coder” and “CodeDoc” attributes are updated accordingly.

2.5 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 5). 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 instance, 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—locations in the office for all employees, as well as for their surroundings (e.g., desks, walls, computers, and chairs). Because these graphical features of the modeling approach are straightforward, and consist of simply assigning images and coordinates to objects, an example is omitted from this paper.

3. Example Waterfall Model

As an initial attempt at building a simulation model using the SimSE modeling approach, we developed a model that emulates a waterfall-like process and teaches a number of overarching lessons about the software engineering process in general. Although the waterfall model is not the most interesting or challenging life cycle model that exists, it is still commonly taught, and its simplicity allows us to clearly demonstrate the principles of the environment and teach some overall lessons about the software engineering process in general. Clearly, many more models of various size and complexity need to be built and evaluated, which we are currently in the process of doing (see Section 7). In the remainder of this section, we first detail the main lessons we aimed to teach in designing the model, and then present how some of those lessons were implemented using our approach.

3.1 Lessons

In developing our model, we aimed to portray a waterfall process by making the model reward the player for following the waterfall process and penalize them for deviating from this process. In addition, we strove to teach a number of overall lessons about software engineering in general. We collected these lessons and phenomena by performing a survey of existing software engineering literature, talking to software engineering professionals, and collecting concepts and theories that are taught in the introductory software engineering class at UC Irvine. The result of these activities is a compendium of 86 "fundamental rules of software engineering" (Navarro, 2002). The following is a representative sample of the breadth of lessons that we implemented in our model, many of which are taken from these rules.

1. *Do requirements, followed by design, followed by implementation, followed by integration, followed by testing.*
2. *At the end of each phase, perform quality assurance activities (e.g., reviews, inspections), followed by correction of any discovered errors.*
3. *If you do not create a high quality design, integration will be problematic.*
4. *Developers' productivity varies greatly depending on their individual skills, and matching the tasks to the skills and motivation of the people available increases productivity* (Boehm, 1981; Bryan, 1997; Sackman, et al., 1968).
5. *The greater the number of developers working on a task simultaneously, the faster that task is finished, but more overall effort is required due to the growing need for communication among developers.* (Brooks, 1995).
6. *Software inspections find a high percentage of errors early in the development life cycle* (Tvedt, 1996).
7. *The better a test is prepared, the higher the amount of detected errors.*
8. *The use of software engineering tools leads to increased productivity* (Tvedt, 1996).

3.2 Implementation

We now present a few of the rules that implement some of the lessons described above. The first rule is a continuous effect rule attached to the "CreateDesign" action that modifies the "size" attribute of the design document artifact being created:

```
1  DesignDoc:
2    Design:
3      size = this.size + (allActiveSoftwareEngineerDesigners.productivityInDesign
4        * (1 – (.01 * (numDesigners * (numDesigners – 1) / 2)))
```

```
5          * (1 + this.completenessDiffRequirementsDoc)
6            * (1 + allActiveDesignEnvironmentTools.productivityIncreaseFactor))
```

In short, this rule says that as a design is being created, the size will increase by an amount dependent on the additive productivity of the designers (line 3), the communication overhead of the number of designers working on it (line 4), the difference in completeness between the requirements document and the design document (line 5), and the productivity increase factor of any design environment tool used (line 6). The amount of increase is primarily based on the productivity of the designers, and each of the other factors serve as multipliers to either raise or lower this amount. We can see in this rule the implementation of a number of the aforementioned software engineering lessons. First, we can see lesson #5 in the first multiplier (line 4). The amount of increase is reduced by 1% for each communication link between two people who are working on the design. (Note that because there exists no empirical data for this value, we assigned it to 1% after trying several different values and playing the game repeatedly in order to determine which value produced the most educationally effective result. This same process was used to formulate many of the rules in our model for which there exists no empirical data.) In the second multiplier (line 5) we can see the implementation of lesson #1 that enforces the sequential nature of the waterfall model. The design document's "completenessDiffRequirementsDoc" attribute is an integer attribute with minimum value 0 and maximum value 1 (hence, it must be either 0 or 1). This value is set in another effect rule that is executed before the one shown here, which sets it to 0 if the requirements document is less complete than the design document, or 1 otherwise. Hence, the amount of increase in the size of the design document is doubled if the features the developers are designing have been specified first. Otherwise, there is no effect.

In the third multiplier (line 6), we can see the implementation of lesson #8, which states that tools increase productivity. The amount of increase in the size of the design document is increased according to the productivity increase factor of the design environment tool.

The next rule is also a continuous effect rule attached to the "CreateDesign" action, but this one modifies the design document's "numUnknownErrors" attribute:

```
1  Design Doc:
2    Design:
3      numUnknownErrors = this.numUnknownErrors +
4       (allActiveSoftwareEngineerDesigners.errorRateInDesign
5         * (1 – (.01 * (numDesigners * (numDesigners – 1) / 2)))
6           * (1 + (allActiveRequirementsDocuments.PercentErroneous / 100 * 10))
7             * (1 + (1 – this.completenessDiffRequirementsDoc))
8               * (1 – allActiveDesignEnvironmentTools.errorRateDecreaseFactor))
```

This rule represents the effect that as the design is being created, a number of unknown errors are being introduced into the design document. This number is primarily based on the designers' additive error rate in design (line 4), and is affected by the communication overhead between the designers (line 5), the number of errors in the requirements document (line 6), the completeness of the requirements document (line 7), and the error rate decrease factor of any design environment tool used. In this rule we can again see lesson #5 implemented (line 5) in that the amount of errors the designers can introduce is tempered by the communication overhead. The next multiplier (line 6) illustrates lesson #2, which states that any errors that are not corrected in one artifact will be carried over into the next artifact. In this expression, the amount by which the design document's unknown errors will increase will be (x * 10)% higher, where x is the percentage of the requirements document that is erroneous. (The amount is multiplied by 10 in order to create a more obvious effect—during play testing of the model, we found that this lesson was not clearly visible enough without this amplification. See Section 5 for a discussion on the tradeoff between accuracy and educational effectiveness.)

The next multiplier (line 7) again illustrates the sequential nature of the waterfall model stated in lesson #1. It represents that the number of unknown errors introduced into the design document will be doubled if the requirements document is less complete than the design document (completenessDiffRequirementsDoc = 0), but will otherwise have no effect (completenessDiffRequirementsDoc = 1).

Finally, the last multiplier (line 8) again implements lesson #8, but affects the artifact's errors rather than the artifact's size, as in the previous rule. This expression represents that the number of unknown errors introduced into the design document will be decreased according to the error rate decrease factor of the design environment tool.

4. Model Builder

To facilitate a high-level, rapid, and easier model building process than writing the above model by hand, 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 7 shows the user interface for the model builder, with the tab for defining object types in focus. The tabs 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, 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 addition models beyond the waterfall, with accompanying documentation, as a part of SimSE. Instructors can then use and/or adapt these models for their own purposes, rather than write them from scratch.

It is important to note that use of the model builder also does not guarantee the model is a "good" model. Rather, a strongly iterative development cycle is required. In our experience so far, building a model involves a significant amount of time aside from the initial construction of the model in which the model is repeatedly played and refined in order to ensure that the desired lessons and effects are illustrated, as well as to achieve the desired balance between educational effectiveness and realism (see Section 5 for further discussion on this issue). To shorten this development time, as well as to provide more insight to players in regards to why they received their particular score, we plan to develop an explanatory tool that can be run in parallel to, or at the end of a game. This tool will provide such information as which rules were fired at which times, as well as graphically show a trace of events and the changing values of various attributes over time. We anticipate that this explanatory tool will significantly shorten the "play-testing" phase of model development by providing a more direct insight into the model's internal workings.

5. 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 (Birtwistle, 1979; Howell and McNab, 1998), or even domain-specific languages designed specifically for modeling software processes (Emmerich and Gruhn, 1991; Kaiser, et al., 1993). 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 in-

clude a wide range of different types of phenomena, from management issues, to organizational behavior theories, to corporate culture, to the traditional software engineering theories, nearly 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 (Ferrari, et al., 1999; Randel, et al., 1992). At the expense of some realism, effects need to be somewhat obvious and "over the top" at times in order to effectively illustrate and enforce the concepts being taught. This can be seen in the example model we presented in Section 3, for instance, in the rule that multiplied by 10 the effect of errors in the requirements document on the number of errors being introduced into the design document. Furthermore, 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.

It can be seen from the description of the modeling constructs and the example model that the SimSE software process modeling approach fits the requirements that we introduced in Section 1. It is *prescriptive* in that the modeler can limit the particular actions a player can take at any given time, and also determine when certain actions must cease—done through the use of conditions on triggers and destroyers. It is *predictive* in that it allows the modeler to specify exactly how the player's actions will affect the state of the simulation, through the use of rules attached to actions. It is *interactive* in that it operates on an incremental, time-step basis, calculating effects and allowing the user to take actions during every clock tick of the simulation. It is *graphical* in that it allows the modeler to specify the graphics to be used in the simulation.

We acknowledge that there are still some weaknesses to our approach, which lie mainly in the fact that it lacks many common programming language constructs, such as if-else statements, explicit data structures, loops, and predicates. This makes it necessary at times to use some non-intuitive, roundabout techniques to get the desired effect. One example of this is in the existence of the "completenessDiffRequirementsDoc" attribute attached to a design document object, discussed in Section 3.2. In any programming language, such an attribute would be unnecessary—an if-else statement with a predicate could simply be used to check whether the completeness of the requirements document was greater than or equal to the completeness of the design document, and, if so, adjust the multiplier in question accordingly. Instead, in our approach, we have to first create this hidden attribute, specify that it can only be equal to either 0 or 1 by making it an

integer with minimum value 0 and maximum value 1, and then create a rule that sets it to the correct value using more mathematical manipulations.

Another instance of this sort of limitation was revealed when we attempted to model the following software engineering rule: Error correction is done most efficiently by the document's author (Drappa and Ludewig). In a full-fledged modeling language this might be modeled by keeping an array of employee names or IDs with the document/artifact object, indicating that those people had been authors of that document. When an employee would then go to correct that document, this array would simply be searched for that employee's name/ID, and, if found, correction would speed up accordingly. In our approach, however, there is no way to perform such a task, due to the absence of arrays, loops, and if-else statements.

Another example of an effect that SimSE could not model is the influence of work environment aspects on productivity. For instance, (Tvedt, 1996) states that improving the work environment by doing such things as giving employees enclosed offices and providing common areas where employees can participate in "water cooler" conversations increases productivity. Because our modeling approach currently uses graphics mainly for decorative purposes, it cannot support this kind of phenomenon. However, we plan in the future to enhance SimSE by adding more semantics to the layout of the office.

Nevertheless, there were very few instances in which we found effects that could not be modeled at all, and none of them are considered the most fundamental principles of software engineering. Most of the effects we wanted to model could be modeled, but required somewhat of a different mode of thinking—in terms of the SimSE modeling constructs provided, rather than the programming language constructs to which most people are used. In order to assist with these difficulties, we plan to provide a "tips and tricks" document along with the model builder's documentation. This document will provide guidelines for how common effects can be modeled that might not be intuitive at first. We believe that the added simplicity of the model builder tool, along with its documentation, will be able to offset most of the drawbacks of our approach. We will be able to informally evaluate whether or not this is true when, in the near future, we distribute the model builder to instructors at several different institutions and get their feedback about it.

6. Related Work

In recent years, there have been a number of new and innovative approaches in both software engineering education and software process simulation. Several have already combined the two to create software process simulations specifically for education (Collofello, 2000; Drappa and Ludewig, 2000; Nulden and Scheepers, 2000; Pfahl, et al., 2000; Sharp and Hall, 2000). To date, the most advanced of these is SESAM (Drappa and Ludewig, 2000) (mentioned previously), a software engineering simulation environment in which students manage a team of virtual employees to complete a virtual project on schedule, within budget, and at or above the required level of quality. SESAM represents a first example of a software process modeling language that is prescriptive, predictive, and interactive (but not graphical). We build on SESAM's approach in the two major ways: First, we simplify the modeling process by providing our graphical model builder tool, eliminating the need for writing source code in an explicit modeling language. Second, we provide support for including graphics in the simulation models, a feature that is considered essential to any successful educational simulation (Ferrari, et al., 1999). We also focus the modeling approach by limiting all objects to the five meta-types (Employee, Artifact, Customer, Project, and Tool), albeit at the expense of extra expressivity and flexibility. Furthermore, we have incorporated many of SESAM's well-documented software engineering rules of behavior into our SimSE models.

Other educational software engineering simulations have included OSS (Sharp and Hall, 2000), which includes extensive graphics, but is not customizable, and puts the player in more of an "observer" role, rather than that of an active participant in the software engineering process. Others make no use of graphics, and have limited interactivity, but are rigidly based on real-world software engineering process data (Collofello, 2000). Still others make various additional tradeoffs between graphics, interactivity, accuracy, and customizability (Nulden and Scheepers, 2000; Pfahl, et al., 2000).

7. 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 creation of interactive, graphical simulation models for software engineering education. We have recently completed a first version of SimSE, along with a high-level waterfall model in which an overall software engineering process is simulated and a number of general lessons about the process as a whole are taught. The game generated by this model was used in an experiment to assess the teaching potential of SimSE. In this experiment, we had 29 undergraduate computer science students who completed an introductory software engineering course at UC Irvine

play the waterfall model version of the game and give us their feedback in a questionnaire. In general, students liked the game, found it enjoyable, believed that it did a relatively good job of illustrating and reinforcing the lessons they learned in lectures, and felt that it should be added as a standard component of the introductory software engineering course. Because this experiment is outside the scope of this paper, its details are not included here, but can be found in (Navarro and van der Hoek, 2005 (to appear)).

We are currently in the process of building three more models: Two of these are high-level models—one teaches the Rational Unified Process model of software development (Kruchten, 2000) and one teaches the Extreme Programming process (Beck, 2000). The third one is a 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 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). We plan to continue conducting experiments with SimSE, in order to evaluate: (1) the teaching potential of the models we have developed (and will develop) by conducting more experiments similar to the one we already completed, as well as incorporating it as a part of the introductory software engineering course at UC Irvine; and (2) the usability, effectiveness, and usefulness of the modeling approach and the model builder tool by having software engineering instructors at various institutions use our tool and provide us with their feedback.

8. More Information

More information about SimSE, as well as downloads, are available at: http://www.ics.uci.edu/~emilyo/SimSE

9. 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 DUE-0341280, 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.

10. References

[1] T. Abdel-Hamid and S. E. Madnick.1991. *Software Project Dynamics: an Integrated Approach*. Prentice-Hall, Inc.: Upper Saddle River, NJ.

[2] K. Beck.2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley: Reading, MA.

[3] G. M. Birtwistle.1979. *Discrete Event Modelling on Simula*. MacMillan Education Ltd: Houndmills, Basingstoke, Hampshire.

[4] B. Boehm, Abts, C., Brown, W., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., and Steece, B.2000. *Software Cost Estimation with COCOMO II*. Prentice Hall: New Jersey.

[5] B. W. Boehm.1981. *Software Engineering Economics*. Prentice Hall, Inc.: Upper Saddle River, NJ.

[6] F. P. Brooks.1995. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley: Boston, MA.

[7] G. E. Bryan, "Not All Programmers are Created Equal," in *Software Engineering Project Management*, R. H. Thayer, Ed. Los Alamitos, CA: IEEE Computer Society, 1997, pp. 346-355.

[8] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, J. Sutton, Stanley M., and A. E. Wise, "Little-JIL/Juliette: A Process Definition Language and Interpreter," in *Proceedings of the 22nd International Conference on Software Engineering*. Limerick, Ireland, 2000, pp. 754-757.

[9] J. S. Collofello, "University/Industry Collaboration in Developing a Simulation Based Software Project Management Training Course," in *Proceedings of the Thirteenth Conference on Software Engineering Education and Training*, S. Mengel and P. J. Knoke, Eds.: IEEE Computer Society, 2000, pp. 161-168.

[10] A. Drappa and J. Ludewig, "Simulation in Software Engineering Training," in *Proceedings of the 22nd International Conference on Software Engineering*: ACM, 2000, pp. 199-208.

[11] W. Emmerich 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*: IEEE Computer Society, 1991, pp. 175-184.

[12] M. Ferrari, R. Taylor, and K. VanLehn, 1999.Adapting Work Simulations for Schools. *The Journal of Educational Computing Research*, **21**(1): 25-53.

[13] F. Howell 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*. San Diego, CA: Society for Computer Simulation, 1998.

[14] G. E. Kaiser, 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*: ACM, 1993, pp. 132-143.

[15] P. Kruchten.2000. *The Rational Unified Process: An Introduction (2nd Edition)*. Addison-Wesley:

[16] P. Lakey, "A Hybrid Software Process Simulation Model for Project Management," in *Proceedings of the 6th Process Simulation Modeling Workshop (ProSim 2003)*. Portland, Oregon, USA, 2003.

[17] MAPICS Inc., "AweSim," http://www.pritsker.com/awesim.asp, 2004.

[18] E. O. Navarro, ""The Fundamental Rules" of Software Engineering," http://www.ics.uci.edu/~emilyo/SimSE/se_rules.html, 2002.

[19] E. O. Navarro and A. van der Hoek, "Design and Evaluation of an Education Software Process Simulation Environment and Associated Model," in *Proceedings of the Eighteenth Conference on Software Engineering Education and Training*. Ottawa, Canada: IEEE, 2005 (to appear).

[20] E. O. Navarro and A. van der Hoek, "SimSE: An Interactive Simulation Game for Software Engineering Education," in *Proceedings of the Seventh IASTED Internation Conference on Computers and Advanced Technology in Education*, V. Uskov, Ed. Kauai, Hawaii: ACTA Press, 2004, pp. 12-17.

[21] J. Noll and W. Scacchi, 2001.Specifying Process-Oriented Hypertext for Organizational Computing. *Journal of Network and Computer Applications*, **24**(1): 39-61.

[22] U. Nulden and H. Scheepers, "Understanding and Learning about Escalation: Simulation in Action," in *Proceedings of the 3rd Process Simulation Modeling Workshop (ProSim 2000)*. London, United Kingdom, 2000.

[23] D. Pfahl, M. Klemm, and G. Ruhe, "Using System Dynamics Simulation Models for Software Project Management Education and Training," in *Proceedings of the 3rd Process Simulation Modeling Workshop (ProSim 2000)*. London, United Kingdom, 2000.

[24] D. Pfahl, G. Laitenberger, G. Ruhe, J. Dorsch, and T. Krivobokova, 2004.Evaluating the Learning Effectiveness of Using Simulations in Software Project Management Education: Results From a Twice Replicated Experiment. *Information and Software Technology*, **46**: 81-147.

[25] J. M. Randel, B. A. Morris, C. D. Wetzel, and B. V. Whitehill, 1992.The Effectiveness of Games for Educational Purposes: A Review of Recent Research. *Simulation and Gaming*, **23**(3): 261-276.

[26] H. Sackman, W. J. Erikson, and E. E. Grant, 1968.Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Communications of the ACM*, **11**(1): 3-11.

[27] H. Sharp and P. Hall, "An Interactive Multimedia Software House Simulation for Postgraduate Software Engineers," in *Proceedings of the 22nd International Conference on Software Engineering*: ACM, 2000, pp. 688-691.

[28] J. D. Tvedt, "An Extensible Model for Evaluating the Impact of Process Improvements on Software Development Cycle Time," Ph.D. Dissertation, Arizona State University, 1996.

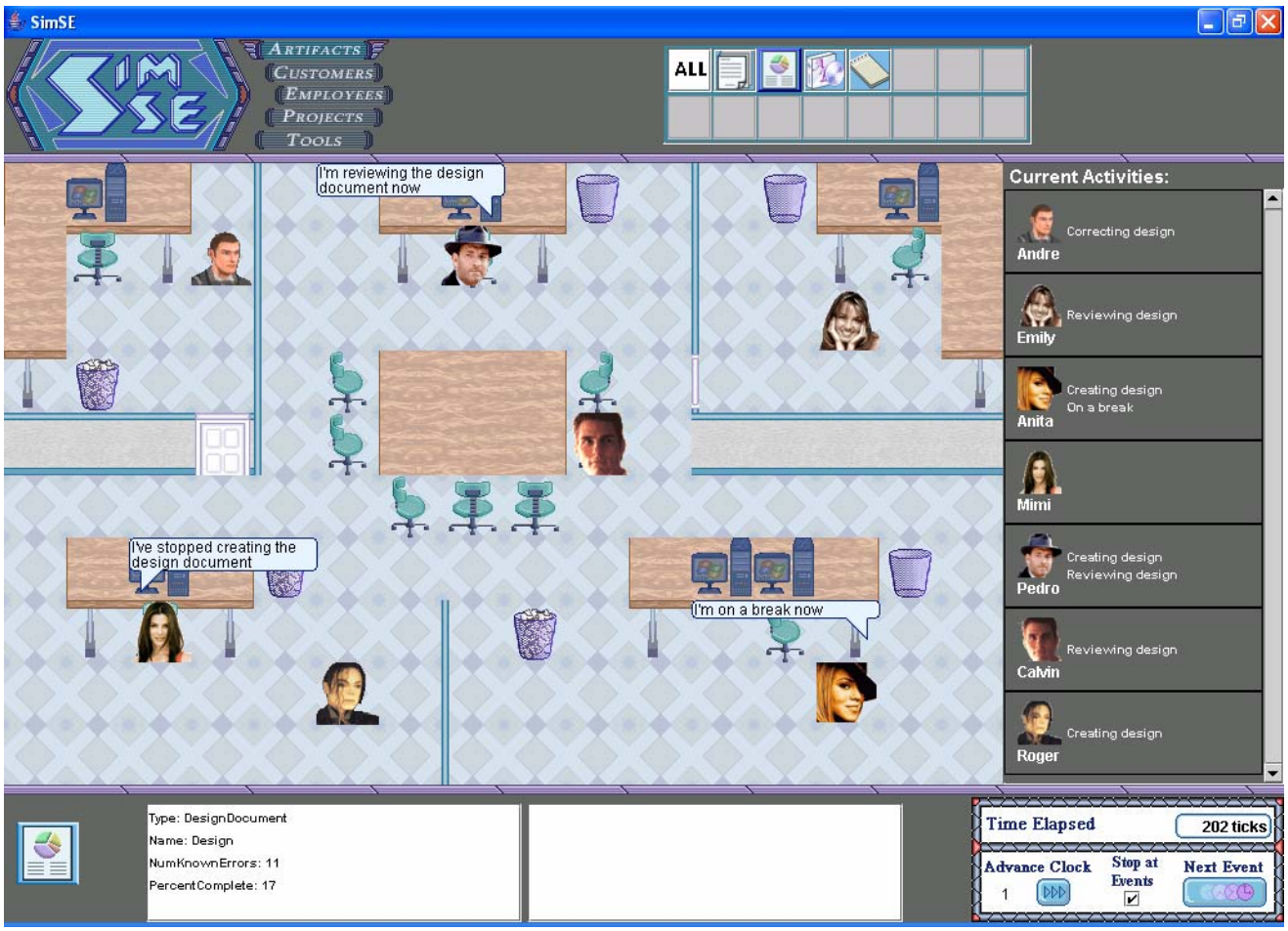

Figure 1: SimSE's Graphical User Interface.

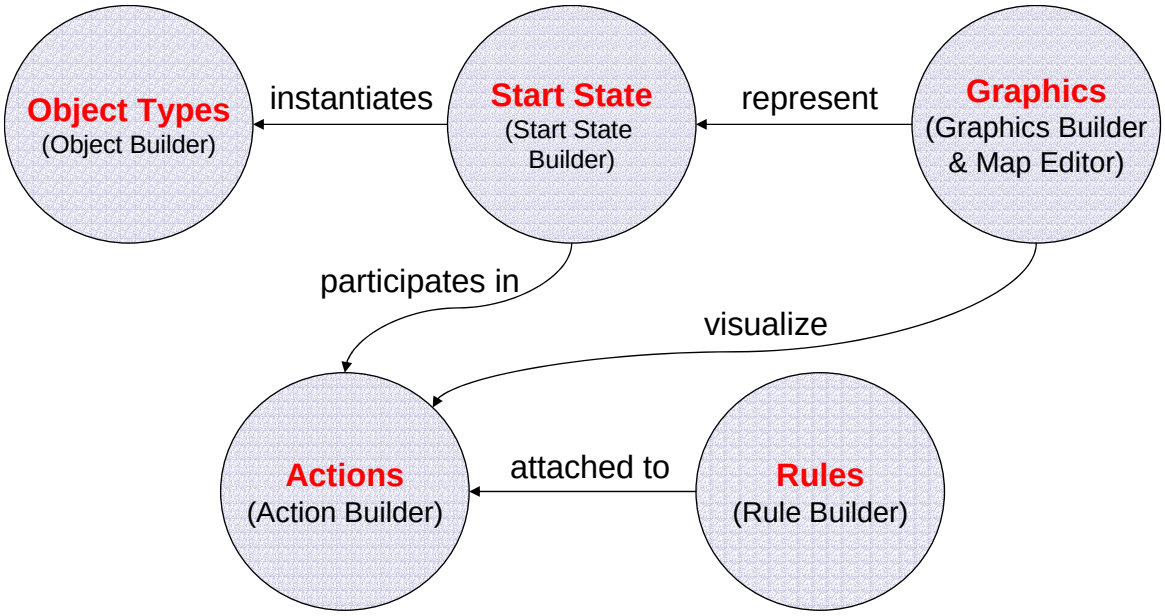

Figure 2: Relationships Between Modeling Constructs.

```
Programmer Employee
{
  name:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  energy:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  productivity:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  error rate:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 1.0
    minDigits: 1
    maxDigits: 2
  hired:
    type: Boolean
    key: false
    visible: true
    visibleAtEnd: true
}
```

```
Code Artifact
{
  name:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  numUnknownErrors:
    type: Double
    key: false
    visible: false
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  numKnownErrors:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  size:
    type: Double
    key: fakse
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 1
    maxDigits: 1
  percentComplete
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: 100.0
    minDigits: 1
    maxDigits: 1
}
```

```
SEProject Project
{
  description:
    type: String
    key: true
    visible: true
    visibleAtEnd: true
  requiredSizeOfCode:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 0
  budget:
    type: Double
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
    minDigits: 0
    maxDigits: 2
  allottedTime:
    type: Integer
    key: false
    visible: true
    visibleAtEnd: true
    minVal: 0.0
    maxVal: boundless
  score:
    type: Integer
    key: false
    visible: false
    visibleAtEnd: true
    minVal: 0
    maxVal: 100
}
```

Figure 3: Programmer, Code, and Project Object Types.

```
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 = 25600.0
  percentComplete = 10.0
}
```

```
Object Project SEProject
{
  description = “Rocket
    Launcher Software”
  requiredSizeOfCode =
    256000
  budget = 2500000.00
  allottedTime = 692
  score = 0
}
```

Figure 4: Instantiated Programmer, Code, and SEProject Objects.

```
Action Coding
{
  Visibility: true
  Description: “Creating code”

  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 userTrigger
  {
    type: User-initiated
    menuText: “Start coding”
    overheadText: “I’m coding now!”
    game-ending: false
    priority: 8
    conditions
    {
      Coder:
        Programmer:
          hired == true
        Tester:
          hired == true
          health >= 0.7

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

  Destroyer autoDestroyer
  {
    type: Autonomous
    overheadText: “I’m finished coding!”
    game-ending: false
    priority: 10
    conditions
    {
      CodeDoc:
        percentComplete == 100
    }
  }

  Destroyer userDestroyer
  {
    type: User-initiated
    menuText: “Stop coding”
    overheadText: “I’ve stopped coding”
    game-ending: false
    priority: 11
    conditions {}
  }
}
```

Figure 5: Sample “Coding” Action with Associated Triggers.

```
Rules
{
  Action: Coding // action that these rules are attached to
  CreateObjectsRule
  {
    timing: trigger
    createdObjects
    {
      Object Code Artifact
      {
        name =“My Code”
        numUnknownErrors = 0
        numKnownErrors = 0
        size = 0.0
        percentComplete = 0.0
      }
    }
  }

  EffectRule
  {
    timing: continuous
    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
        size = this.size + allActiveProgrammerCoders.productivity
        numUnknownErrors = this.numUnknownErrors + allActiveProgrammerCoders.errorRate
        numKnownErrors = // no effect
        percentComplete = (this.size / allSEProjectProjects.targetCodeSize) * 100
  }
}
```

Figure 6: Example Create Objects Rule and Example Effects Rule.

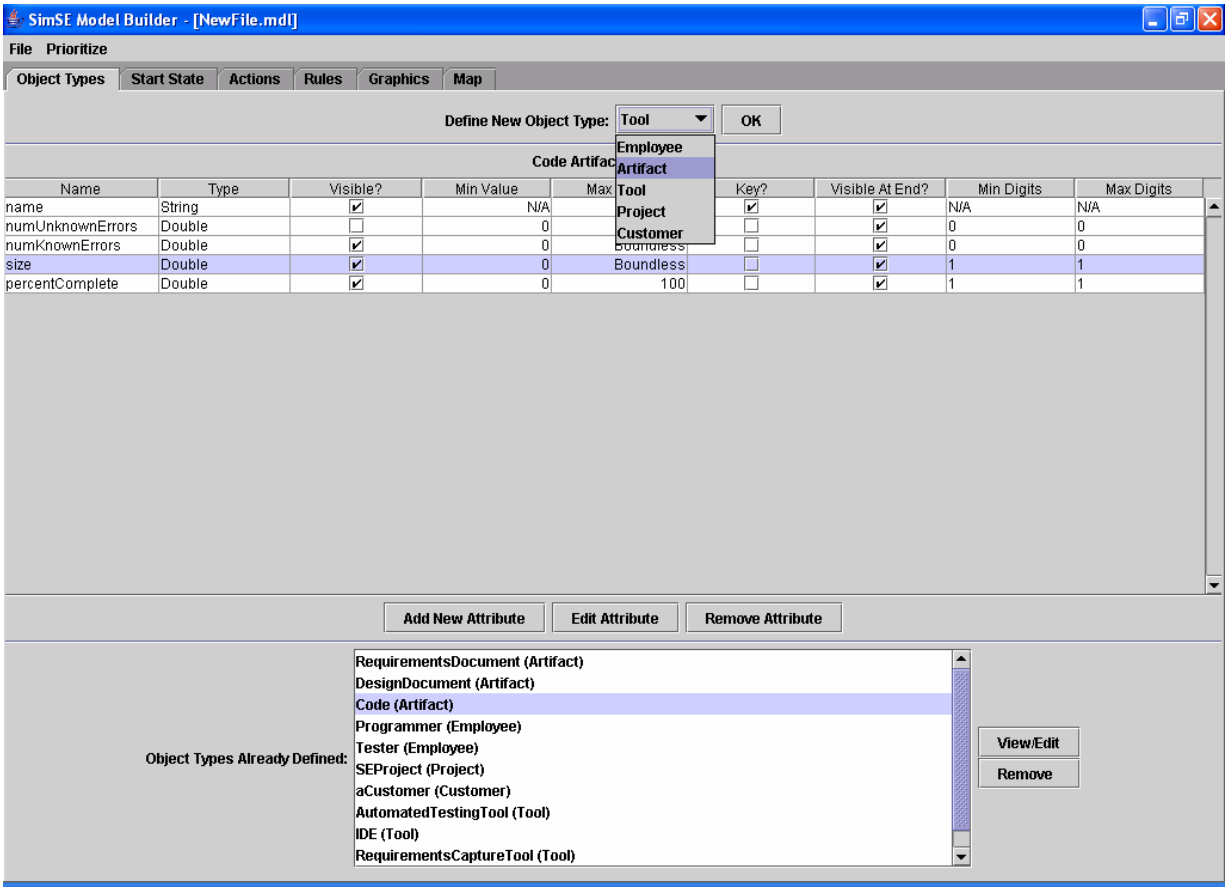

Figure 7: Model Builder User Interface.

Table 1: Timing of Execution of Each Different Type of Rule.

		Rule Type		
		Create Objects	**Destroy Objects**	**Effect**
Rule Timing Type	**Trigger**	Once, at trigger time	Once, at trigger time	Once, at trigger time
	Destroyer	Once, at destroyer time	Once, at destroyer time	Once, at destroyer time
	Singular	Once, during the first clock tick the action is active (after trigger)	Once, during the first clock tick the action is active (after trigger)	
	Continuous			Once every clock tick that the action is active