Paper presented at ESM 2001 – European Simulation Multiconference 2001
Prague, Czech Republic, June 6-9, 2001

ADVANCED OBJECT-ORIENTED TECHNOLOGIES IN MODELING AND SIMULATION: THE VSEit FRAMEWORK

Kai-H. Brassel
Department of Sociology
Technical University of Darmstadt
Residenzschloss
D-64283 Darmstadt, Germany
E-mail: brassel@vseit.de

KEYWORDS

Object-oriented models, hybrid simulation, interactive simulation, model design, software engineering, Java, web-based simulation.

ABSTRACT

Modeling and simulating complex systems for explorative purposes is indispensable in research and teaching. Such activity can greatly benefit from a middle-ground tool that is more specific than universal programming languages on the one hand, and more general than modeling tools devoted to a distinct model type on the other. Such a tool could provide good support for simulation specific tasks while facilitating the integration of different model types. Advances in software technology help strike that middle ground. The VSEit simulation framework extends standard object-oriented concepts to support simulation control, flexible editing of model structure, data sampling and output display. The working of these concepts is illustrated by two examples.

1 LOOKING FOR A MIDDLE-GROUND SIMULATION TOOL

Building and simulating models is indispensable for research and teaching. But it is a rare person who can be all: a good scientist, teacher, and programmer. In fact, a common adage says that "building simulation models often turns good scientists into bad programmers."

Scientists typically deal with this dilemma in one of two ways. Either they team up with software developers to create case-specific simulation programs, or they employ one of the high-level modeling tools for specific types of simulation models, like Stella for systems dynamics models (http://www.hps-inc.com/edu/stella/stella.html), SimProcess for queuing systems (http://www.caciasl.com/simprocess.cfm), or AgentSheets for simple agent-based models (Repenning and Ioannidou 2000).

Both can be limiting. The first approach seems to afford the scientist a maximum degree of freedom and flexibility for realizing her ideas – at least in principle. In practice, however, developing a model and making changes tends to be cumbersome and time-intensive. In the second case, the scientist restricts herself to a specific class of model; when she wants to try out a different model type, she will have to switch to another tool. This makes it especially difficult to link models of different types and problem domains (e.g., a hydrology model representing the water cycle in a river basin with an agent-based model of the people using the basin's natural resources).

But it need not be like this. Not only did breath-taking advances in processing speed, memory capacity, and communication facilities dramatically improve our hardware capabilities to perform simulations. The past few years have also seen great advances in *software* technology. They allow us to strike a better balance between the two approaches, enabling people who are not proficient programmers to still develop adequate, well-structured, and efficient computer simulations *without* being restricted to a specific model class.

This paper presents some advances in object-oriented technology, as embodied in the Java-based VSEit simulation framework (*V*ersatile *S*imulation *E*nvironment for the *i*nterne*t*, pronounced as "use-it"). A first version of VSEit was created while the author was working as a computer scientist in a team of economists and sociologists on simulation models of environmental policies. The development of the VSEit framework benefited from the close contact to domain experts and students, both of whom were avid users who provided valuable feedback on the design of the software.

We first discuss requirements for tools to support explorative modeling and simulation of complex domains (section 2), then demonstrate how advanced object-oriented concepts can be exploited when implementing such a tool (section 3), and illustrate the working of VSEit with two examples (section 4). We also offer some conclusions (section 5).

2 WHAT WE WANT FROM A SIMULATION TOOL

This section contains a wish list of features that we would want a simulation tool to have. The list is not meant to be exhaustive; rather, it reflects the experience of researchers engaged in *explorative* modeling of *complex* real-world processes, as is necessary for policy modeling and integrated assessment.

Scope. The most fundamental property of a simulation tool may be its scope, that is, the set of simulations that it is potentially able to execute. Figure 1 indicates the scopes of different software applications. There are many tools designed for simulating a specific type of model, like systems dynamics models or cellular automata. These tools may give very good support for the purpose they are designed for, but they tend to be too restrictive for explorative modeling and simulation of complex domains. This is for two reasons: First, the very complexity of the target system often calls for combining different modeling approaches within one simulation model. Second, explorative modeling implies that the model's type and structure is not known beforehand, but evolves, step by step, during the modeling process. As a result, it may be necessary to invent new classes of simulation models "on the fly."

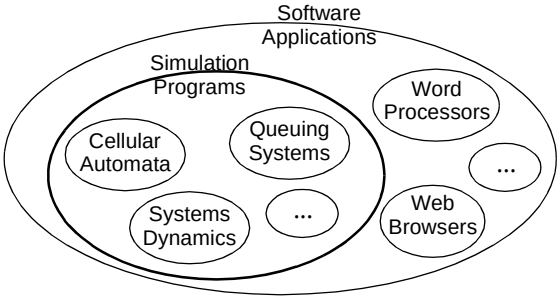

Figure 1: Simple Topology of Software Applications

At the other end of the spectrum, we have numerous universal programming languages and development environments supporting the implementation of *any* application, including computer simulations of all kinds. What these tools necessarily are missing is simplicity and special support of those features that are common to *all* simulation programs, such as model initialization or display of simulation results.

Before we present the ideas embodied in the VSEit approach for striking the middle ground between high-level simulation tools and universal programming languages, let us discuss, in the remainder of this section, some requirements that a tool for modeling complex systems in an explorative setting should satisfy.

Complexity. Truly complex domains exhibit both structural and temporal complexity. *Structural complexity* is due to the presence of many model objects, usually of different types, connected by different types of relationships, and placed at different levels of the system's hierarchy. *Temporal complexity* results from the existence of more or less independent (concurrent) processes, e.g. autonomous agents pursuing their individual goals in a common environment. A tool for simulating complex systems should provide powerful capabilities for designing and implementing complex structures (e.g. a type-based network editor) and processes (e. g. an efficient event scheduler).

The simulation tool has to meet even greater challenges when one wants to let substructures of a model be generated or modified by simulated processes or, vice versa, let specific substructures control the execution of a process. In the first case, the simulation tool must be able to deal with *variable structure models* (Uhrmacher and Zeigler 1996), i.e. models with a variable number of objects (maybe due to birth and death processes) and changing relationships between objects. In the second case, the user may provide or modify parts of the model structure in order to specify certain aspects of the model's behavior, not via program code at compile time, but during model initialization or even while the simulation is running. The simulation of intelligent agents, by the way, requires both: The acquisition of new (cognitive) structures through on-line learning *and* the on-line interpretation of those structures for performing actions.

As we will see below, a carefully designed simulation tool may indeed be able to support modeling and simulation of all these facets of complexity.

Exploration. Following Troitzsch (1997), explorative modeling and simulation serves to build an understanding of the principle behavior and functioning of a target system, rather than making exact predictions about its future states. An explorative mode of inquiry is particularly well supported by tools with comprehensive communication capabilities. These include a *modern graphical user interface* (GUI) allowing for interactive model initialization and simulation control as well as for the animation of model objects, flexible data sampling and output, and, last not least, a user-friendly model documentation.

The advantages of *web-based simulation models* for the communication between modelers and model users are obvious. Web-based models are platform-independent and can easily be shared around the globe. This affords transparency and accessibility which is especially important when models are used to support policy analysis. For the same reasons, the modeling language of a tool should be as *adequate and expressive* as possible. Model code should be easy to create, understand, and maintain.

3 EXPLOITING ADVANCED OBJECT-ORIENTED TECHNOLOGIES FOR MODELING AND SIMULATION

Among all universal approaches to software development, indicated by the outmost oval in Figure 1, object-oriented languages, concepts, and tools are commonly recognized to be quite well suited for developing computer simulations. This is mainly because they employ concepts like "object," "class," "inheritance," and "message" that often correspond directly to the way in which we view things in a real-world target domain. The goal pursued with the development of VSEit was to add support for modeling and simulation in general with a special emphasis on explorative modeling of complex domains. In this section, we discuss how some advanced object-oriented technologies did help us to achieve

this goal. The subsequent section illustrates how this looks in practice.

3.1 Representation and Editing of Complex Model Structures: VSEit's Network Editor

An important part of the VSEit framework is the Network Editor, which graphically represents the model objects and their relationships, and which allows modelers and model users to manipulate objects and edit model structure. Objects are given graphical symbols which appear on screen. With a simple mouse click, objects can be created and accessed for editing. Likewise, relationships that are represented by lines or arrows connecting the object-shapes can be created and edited this way. See Section 4.2 for an illustration.

The Network Editor relies on an enhancement of standard object-oriented semantics. It introduces an enriched concept of "object" and "class". Leaning on data base terminology, we call the enriched object concept "entity," and the enriched class concept "entity type." Each object in the model has one entity assigned to it. The entity is like an alter ego of the object that fulfils two functions: It internally represents the enhanced semantics of model objects ("attributes" and "roles," explained below). Also, it organizes the user's interaction with the object in the Network Editor and keeps track of all changes in the object's state, performs updates accordingly and thereby supports animation.

The new concepts of "entity" and "entity type" rely, in turn, on enriched concepts for "instance variable" and "object reference." These are called "attribute" and "role," respectively. They are more powerful and versatile than their standard counterparts, especially for modeling tasks. Attributes allow for a more intuitive and comprehensive description of an object's state. They accommodate:

- multiplicity (how many values are allowed or required for the attribute),
- explicit handling of missing values,
- specification of valid values,
- explicit annotation of data units.

Likewise, roles accommodate multiplicity and explicit handling of missing values. In addition, they allow for the distinction between directed and undirected relationships, ensuring consistency of references over the entire network at each point in time.

From a technical point of view, it was important to integrate these enhanced static semantics into the standard class concept, thus preserving the possibility to attach arbitrary behavior to the entities by defining methods in the usual fashion, including inheritance. A seamless integration was achieved by utilizing the advanced feature of "reflection," that is, the ability of an application to inspect its program while it is running and subsequently process information about its classes, variables and methods. Java's application programmers interface (API) for coding reflective operations has become more sophisticated with recent versions. However, a careful usage of this mechanism is called for, to avoid too big an overhead in runtime.

3.2 Handling of Concurrent Processes

Modern programming platforms usually enable application programmers to implement concurrent processes. Java, for instance, provides *threads* that are simple to use and well suited for most programming tasks. Employing threads, VSEit is capable of executing many runs of a simulation model in parallel with one thread for each run, plus one extra thread for managing the GUI. However, implementing *model processes* by means of threads is problematic. Since threads are scheduled by the runtime environment, their timing, and thus, the whole simulation, usually is not reproducible.

In order for the simulation environment to retain fine-grained control over the scheduling of events, including the reproducible solution of conflicts by a pseudo-random number generator, it has to provide its own scheduler. The efficient implementation of such a scheduler, as realized in the VSEit framework, requires that executable code is stored within an event queue for delayed execution. "No big deal," anyone will say who knows Lisp expressions, Smalltalk code blocks, or C function pointers. But note that these are inherently unsafe and therefore should not be used for web-based simulation. Java, on the other hand, is type-safe. But this implies that the only place for storing executable code is a class. Coding every potential model event as a full-fledged class would make the model code quite unreadable. Java, since version JDK 1.1, provides a special syntax to address this problem, labelled "inner classes." With this special notation, modelers elegantly can bind chunks of model code to arbitrary objects without loss of safety and type information.

3.3 Data Sampling and Output

Since the early days of Smalltalk, the so-called Model-View-Controller (MVC) design pattern is the basis for most object-oriented implementations of interactive GUIs. This is also the case with the VSEit framework, whose rich output facilities include dynamic plotting of time series and other charts, as well as animation of model entities and relationships. However, some of the specific requirements for explorative modeling deserve special attention.

One is the handling of variable structure models. In those models, entities may be created or destroyed in the course of the simulation (see section 4.2 for an illustration). To allow for data about such transient entities to be gathered and interactively displayed, the classic MVC approach had to be made more dynamic. Once this is done, it also becomes possible to let the user decide, *during* a simulation run, for which of the various entities in the model data shall be sampled and displayed. For example, in an agent-based model, the user can, upon observing the fate of some agents, begin to check what is going on with other agents. This is very useful for exploring the behavior of complex simulation models.

Another requirement, not easy to satisfy, is to improve the readability of the model code by separating all data-related

aspects from describing the model's structure and behavior. Intuitively, the most transparent approach would be to encapsulate data definitions and data sampling methods within special (data) classes. Putting this intuition into practice required the design of a generic and efficient interface between those classes and the output windows that display the instances of those classes (i.e. data records). Implementing this interface greatly profited from Java's reflection API that allows the running application to determine what instance variables are defined in a data class.

3.4 Miscellaneous Benefits Due to Recent Enhancements of the Java Platform

Since its introduction in 1995, the Java computing platform, that is the programming language together with several APIs and development tools, has evolved rapidly. The high frequency of new releases certainly caused some problems while developing the VSEit framework, since it had to be re-implemented several times. On the other hand, the Java platform seems to have matured by now. Moreover, several of the recent improvements are especially useful for addressing the requirements for a simulation tool, as formulated in section 2.

One major addition to the Java platform was the "Swing" framework for implementing professional GUIs. Besides improving the look-and-feel of a Java application or applet, it provides practical features like tool-tips that can be attached to all interface elements and help the user to understand the application. Swing also facilitates the implementation of custom windowing systems. Since simulation programs tend to be quite "window intensive," it is always a good idea to provide special support for window manipulation (for the model user's benefit) and for implementing custom types of windows (for the modeler's benefit).

Since VSEit simulations are web-based, it is only natural that HTML files be used for model documentation. Enhanced text processing capabilities of the Java platform allow the display of HTML documents within an application. That way, the user can easily switch from a simulation run to the relevant parts of the model documentation or, vice versa, execute a new run from within the documentation.

With the 2D Graphics API, Java provides a comprehensive tool box for creating all kinds of drawings. VSEit exploits this feature, for instance by allowing the modeler to define arbitrary geometric shapes to represent model entities in the Network Editor. Utilizing the 2D Graphics API, these shapes can very easily be scaled in size or otherwise be transformed – a feature very helpful for animating model entities.

Besides the "Math" class that implements the common arithmetic operations, newer Java versions also come with a class called "StrictMath". This is used throughout VSEit because it guarantees total, that is bitwise, reproducibility of arithmetic operations, including floating point operations and generation of pseudo-random numbers, *regardless* of what hardware and operation system is in use. Only this nice feature makes simulation runs truly reproducible "around the globe."

The overhauled Collections API of Java provides a well-sorted library of ready-made data structures like Lists, Sets, Hashtables, and so on. It even contains a generic method called "shuffle" that rearranges the objects in a given list randomly, a function often used by simulations to avoid implementation artefacts.

Last, not least, many powerful integrated development environments for Java were made available over the past few years, some free of charge. They provide advanced features like automatic code completion or the creation of classes from templates, which are especially helpful for occasional users, like VSEit modelers might be.

4 ILLUSTRATION: TWO APPLICATIONS

About a dozen simulation models were implemented within the VSEit framework up to April 2001. They include systems dynamics models of ecological and economic development, multi-level models and irregular cellular automata illustrating opinion formation processes, variable structure models of firm behavior and technological change, and some experimental multi-agent simulations.

This section presents two VSEit models to illustrate the working and versatility of that tool. Both models were originally implemented with other tools, also aiming at the middle ground between high-level simulation systems and universal programming languages. Comparing different implementations of the same models will provide first insights about the performance of VSEit. Source code and executable JAR-files for the example models are available at "http://www.vseit.de" in the World Wide Web.

4.1 Heatbugs

This model has already been implemented as a demo application of the SWARM simulation toolkit (Swarm Development Group 2000). The theme is a population of bugs, each characterized by an "output heat," specifying the amount of heat it radiates, and an "ideal temperature," that is the outside temperature it prefers most. These bugs are superimposed on a cellular automaton called the "heat space," which regulates heat diffusion. The bugs radiate the heat to the cell on top of which they are sitting, from which it diffuses to surrounding cells according to a diffusion constant. There is also some "heat evaporation." The bugs move along the automaton, one cell at a time, in search for their ideal temperature. Thus, this model can be considered as a combination of a cellular automaton with simple agents. Figure 2 presents a snapshot of the GUI after VSEit has executed a typical Heatbugs simulation for 162 time steps.

The main window of the simulation is dominated by the Network Editor. It shows model entities of different types on the left and the current attribute values of a selected entity (in this case a heat bug) on the right. Note that the bugs are not simple blobs on the display, but individually selectable

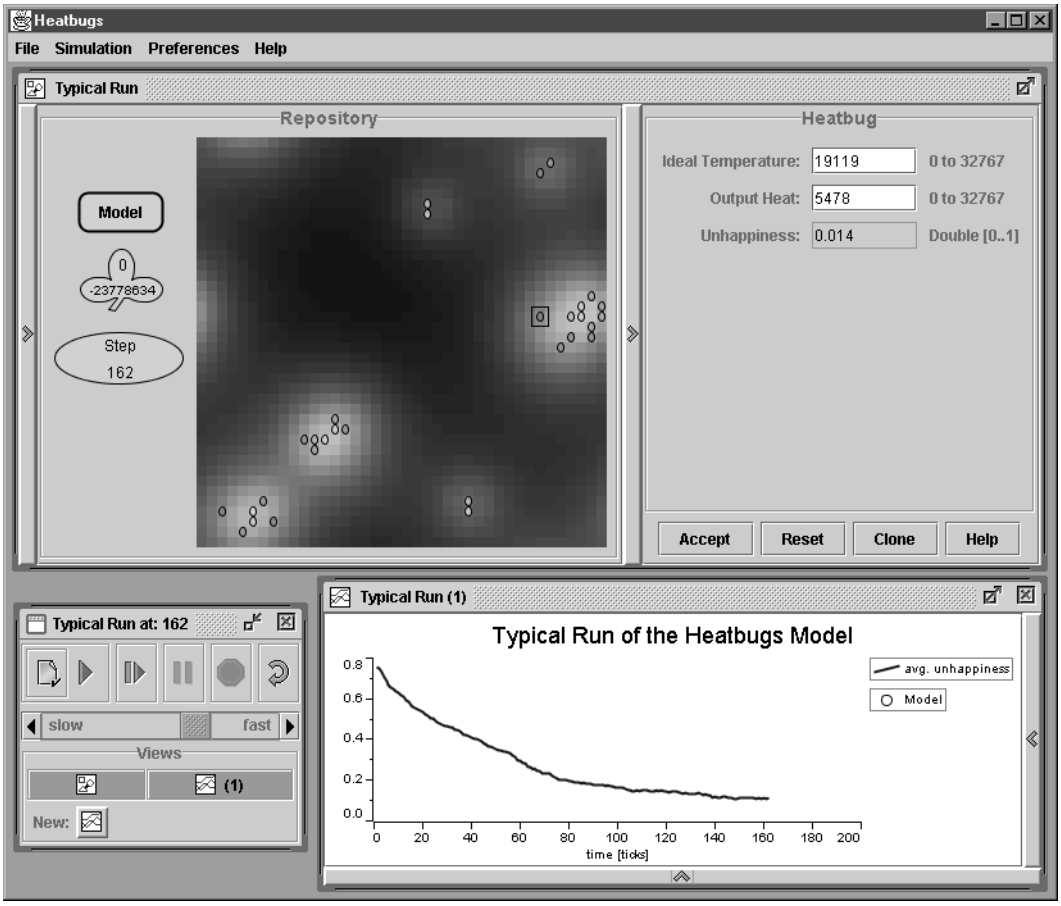

Figure 2: Heatbugs Simulation Model Implemented with VSEit

entities. The user may select one or more bugs for better tracking their movements or even to manually shift them to another position and watch how they react. Since bugs are represented as true model entities, it would also be simple to enhance the model by introducing non-local relationships between bugs and displaying them as links, just like Epstein and Axtell (1996, p. 81) did in their Sugerscape model.

If you look carefully, you will notice that bugs are displayed in different shades. The brighter a bug, the higher its ideal temperature. Altering a bug's ideal temperature, manually or by simulation, instantly triggers an update of its brightness.

The bugs' common environment, the heat space, is implemented as a specific type of model entity, too (see Brassel et al. 1997 on a discussion about different notions of "environment" in multi-agent models). Thus, it would be easy to insert additional heat spaces, if model design required it. The attributes of a heat space specify its dimension, i.e. the number of horizontal and vertical cells making up the space, the extension of the cells, the diffusion constant, and the evaporation rate. The modeler controls which attribute values a model user might change at runtime, and which not. In the Heatbugs model, for instance, allowing the cells' extension to be changed at runtime makes sense, as it lets the model user adjust the display, while changing the dimension of a heat space is only allowed during model initialization.

The VSEit version of a model with 100 heat bugs distributed in a 80 times 80 cell heat space runs smoothly on a 500 MHz Pentium PC. However, with all animation features switched on, it runs about three times more slowly than the SWARM implementation. This is not a bad result, considering that the Java code is interpreted for execution, while the SWARM code was compiled by a native compiler. Since graphical output and animation are the most greedy consumers of computing time, performance can be improved instantly by hiding some of the heat bugs or increasing the update interval. If speed still matters and remote execution via web-browsers is not intended, the VSEit model may of course also be compiled by a native Java compiler generating a comparably fast executable file.

As for model specification, the VSEit version is much more compact and readable than its SWARM counterpart. This is remarkable, since the former permits a higher degree of interactivity, and specifies the cellular automaton and its diffusion process explicitly, while the latter delegates this part of the simulation to specific library classes. Having said that, the author acknowledges that such a judgement is necessarily subjective. Therefore, the reader is invited to judge for himself by looking at the model's source codes located at "http://www.vseit.de/de/vseit/examples/heatbugs" and "http://www.swarm.org/release-apps.html", respectively.

4.2 Greening Investors

The Greening Investors model was built to reproduce some of the stylized facts that can be observed around the processes of invention, innovation, and diffusion of new technologies in an economy. The model was designed by stepwise refinement, starting with a macro-model of two competing production technologies (representing diffusion), introducing, in a second step, an unlimited reservoir of new technologies (invention), and ending up with a multi-level model, where firms explicitly choose to adopt a new technology (innovation). This set of models was originally implemented using the MIMOSE environment (Möhring 1996). For a detailed model description, including MIMOSE code, see Brassel et al. (2000).

Figure 3 shows a Greening Investors simulation run as executed in the VSEit framework. The Network Editor

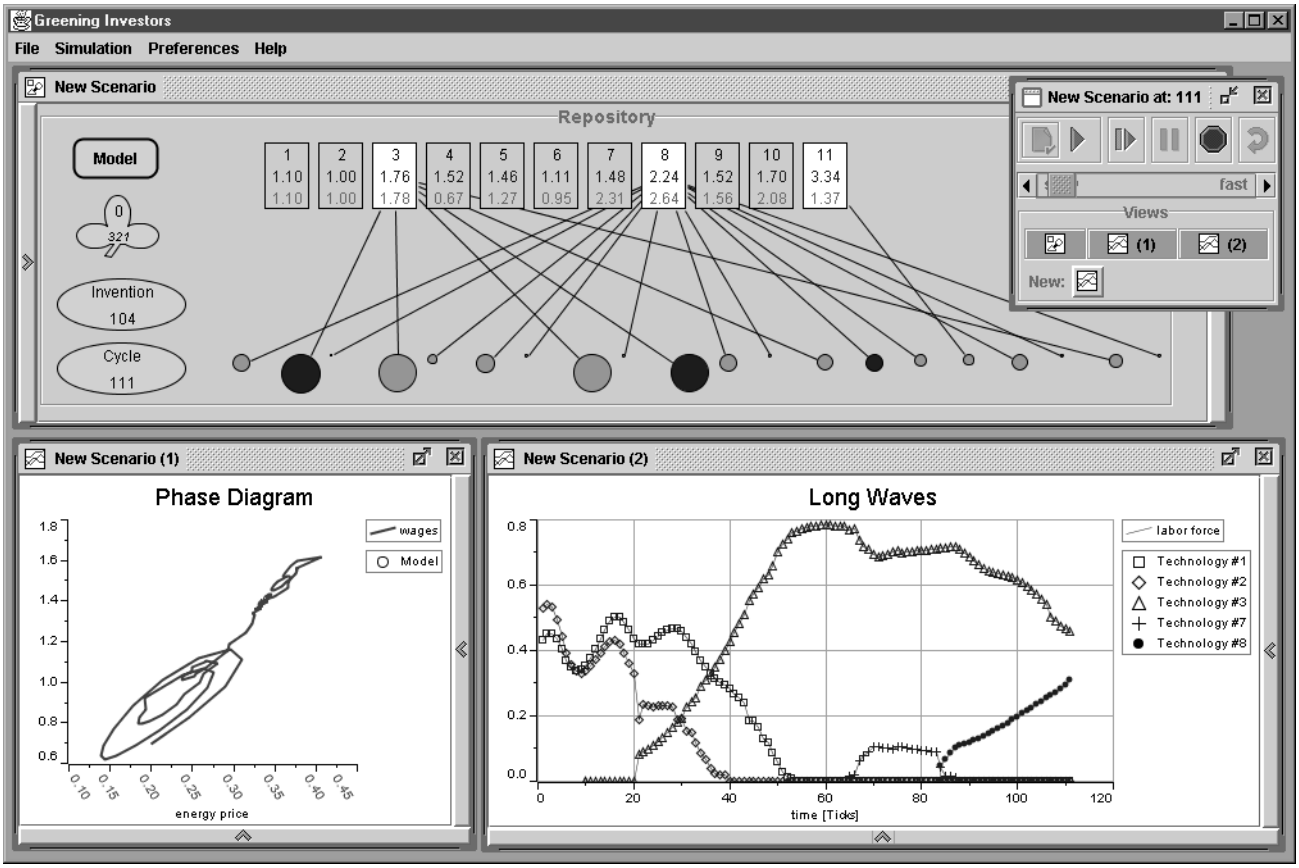

Figure 3: Snapshot of a Greening Investors Simulation Run

contains a series of rectangles, each representing one of the technologies that are available at the current time step. A technology is characterized by its labor and energy productivity, both displayed as floating point numbers. From time to time, a new technology is added randomly, according to a Poisson distribution. Following given trend parameters, new technologies will on average be more efficient than older ones.

The circles below the technologies represent individual firms. Each firm is linked to the technology that it currently uses for production. Firms utilizing more efficient technologies than their competitors experience a greater output growth. Since the size of the circles is proportional to a firm's output, some of them will grow over time, while others will shrink. Firms that fall below a threshold start looking for a better technology. They are modeled to differ in their preference for energy and labor productivity (a feature that departs from conventional economic analysis). As soon as a new technology that suits such a firm's preference is available, its link is redirected to that new technology, indicating an innovation.

Note that production and innovation on the one hand, and invention on the other, are modeled as concurrent (independent) processes.

The chart titled "Long Waves" illustrates the working of the advanced data sampling and output facilities of VSEit, as discussed in section 3.3. It shows the rise and fall of technologies by plotting the sum of all goods produced with one technology over time. Whenever a new technology is created during a simulation run, a new curve is inserted automatically.

Since MIMOSE provides a functional language especially designed for multi-level-modeling, the specification of the Greening Investors model in that tool is much more compact than the VSEit version. However, implementing the dynamics

of the invention process turned out to be somewhat complicated in MIMOSE, since the creation of new model objects is not directly supported.

Compared to VSEit, MIMOSE runs about ten times more slowly, and that even though animation of model objects is not supported. Especially when models become more complex, as is the case with Greening Investors, display and animation of model objects and structure is very helpful for understanding the model's dynamics and for detecting logical errors.

5 CONCLUSIONS

Experience with re-implementing the Heatbugs and the Greening Investors simulation models indicates that the VSEit framework constitutes a good compromise between universal programming languages on the one hand and specific simulation tools on the other. The utilization of advanced object-oriented concepts and technologies did help to strike that middle-ground and also allowed the implementation of features that are especially useful for the explorative modeling of complex domains.

The enhancement of the standard object-oriented concepts "object," "class," "instance variable," and "object reference" was the basis for creating a powerful Network Editor that supports initialization and animation of model "entities" as well as model structure. By defining suitable types of entities and relationships, VSEit becomes a generic tool for the graphical definition of models belonging to different specific model classes. Being implemented in the same environment, models of different classes could easily be integrated, at least from a technical point of view. See Peters and Brassel (2000) for a proposal of such a hybrid model class that could be applied to economic policy analysis.

Despite VSEit's capability to produce highly complex and easy-to-handle simulations, developing a new simulation model, or even inventing a new model class, only requires moderate programming skills. Of course, quality of documentation is crucial in this respect.

Analysis and testing of the VSEit framework show that its simulations are time and space efficient. In part, this is due to a cautious application of time consuming mechanisms like reflection.

Java as a simulation platform has more to offer than is suggested by the numerous simulation applets developed ad hoc that can be found in the World Wide Web. This is mainly due to recent enhancements of the Java platform.

Certainly, a more systematic assessment of the strengths and weaknesses of the proposed architecture and the VSEit framework would be desirable. We also intend to provide building blocks for more challenging model classes in the near future, e.g. to implement intelligent behavior based on decision networks or re-enforcement learning. New versions of the VSEit framework will support distributed simulation and gaming as well as the specification and analysis of simulation experiments.

ACKNOWLEDGEMENTS

I am very grateful to Irene Peters for her encouragement, many fruitful discussions, and valuable comments on several versions of the draft. I thank Michael Möhring for clarifying comments.

REFERENCES

Brassel, K.-H. 1996. "Erfahrungen mit der objektorientierten Implementierung komplexer Modelle." In *MASSIM-96 – Multiagent Systems and Simulation Workshop* (Ulm, Germany, March 5-6). ASIM Communications, Vol. 53.

Brassel, K.-H.; O. Edenhofer; M. Möhring; and K.G. Troitzsch. 2000. "Modelling Greening Investors: Economic Development, Opinion Formation, and Technological Change in a Multilevel Simulation Model." In *Tools and Techniques for Social Science Simulation*, R. Suleiman, K.G. Troitzsch, and N. Gilbert (Eds.). Physica, Heidelberg, 317-343.

Brassel, K.-H.; M. Möhring; E. Schumacher; and K.G. Troitzsch. 1997. "Can Agents Cover All the World?" In *Simulating Social Phenomena*, R. Conte, R. Hegselmann, and P. Terna (Eds.). Springer, Berlin, 55-72.

Epstein, J.M. and R. Axtell. 1996. *Growing Artificial Societies – Social Science from the Bottom Up*. MIT Press.

Flanagan, D. 1997. *Java in a Nutshell*. 2nd ed., O'Reilly.

Gilbert, N. and K.G. Troitzsch. 1999. *Simulation for the Social Scientist*. Open University Press.

Minar, N.; R. Burkhart; C. Langton; and M. Askenasi. 1996. "The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations." Santa Fe Institute, http://www.swarm.org/intro-papers.html.

Möhring, M. 1996. "Social Science Multilevel Simulation with MIMOSE." In *Social Science Microsimulation*, K.G. Troitzsch, U. Mueller, N. Gilbert, and J.E. Doran (Eds.). Springer, Berlin, 123-137.

Peters, I. and K.-H. Brassel. 2000. "Integrating Computable General Equilibrium Models and Multi-Agent Systems – Why and How." In *2000 AI, Simulation and Planning In High Autonomy Systems*, H.S. Sarjoughian et al. (Eds.). SCS, 27-35.

Repenning, A. and A. Ioannidou. 2000. "AgentSheets: End-User Programmable Simulations." *Journal of Artificial Societies and Social Simulation* 3, No. 3, http://www.soc.surrey.ac.uk/JASSS/3/3/forum/1.html.

Swarm Development Group 2000. "Documentation Set for Swarm." http://www.swarm.org/release-docs.html.

Troitzsch, K.G. 1997. "Social Science Simulation – Origins, Prospects, Purposes." In *Simulating Social Phenomena*, R. Conte, R. Hegselmann, and P. Terna (Eds.). Springer, Berlin, 41-54.

Uhrmacher, A.M. and B.P. Zeigler. 1996. "Variable Structure Modelling in Object-Oriented Simulation." *International Journal on General Systems* 24, No. 4, 359-375.