CHAPTER 1

INTRODUCTION

Simulation has made possible systems that would otherwise be impracticable. The sophisticated controls in modern aircraft and automobiles, the powerful microprocessors in desktop computers, and space-faring robots are possible because simulations reduce substantially the need for expensive prototypes. These complicated systems are designed with the aid of sophisticated simulators, and the simulation software itself has therefore become a major part of most engineering efforts. A project's success may hinge on the construction of affordable, reliable simulators.

Good software engineering practices and a serviceable software architecture are essential to building software for any purpose, and simulators are no exception. The cost of a simulator is determined less by the technical intricacy of its subject than by factors common to all software: the clarity and completeness of requirements, the design and development processes that control complexity, effective testing and maintenance, and the ability to adapt to changing needs. Small software projects that lack any of these attributes are expensive at best, and the absence of some or all of these points is endemic to projects that fail.[1]

It is nonetheless common for the design of a complicated simulator to be driven almost exclusively by consideration of the objects being simulated. The project begins with a problem that is carefully circumscribed: for example, to calculate the time-varying voltages and currents in a circuit, to estimate the in-process storage requirements of a manufacturing facility, or to determine the rate at which a disease

[1]Charette's article on why software fails [22] gives an excellent and readable account of spectacular software failures, and Brooks' *The Mythical Man Month* [14] is as relevant today as its was in the 1970s.

Building Software for Simulation: Theory and Algorithms, with Applications in C++, By James J. Nutaro

will spread through a population. Equipped with an appropriate set of algorithms, the scientist or engineer crafts a program to answer the question at hand. The end result has three facets: the model, an algorithm for computing its trajectories, and some means for getting data into and out of the simulator. The first of these are the reason why the simulator is being built. The other two, however, often constitute the majority of the code. Because they are secondary interests, their scope and size are reduced by specialization; peculiarities of the model are exploited as the simulator is built, and so its three aspects become inextricably linked.

If the model is so fundamental as to merit its exact application to a large number of similar systems, then this approach to simulation can be very successful.[2] More likely, however, is that a simulator will be replaced if it does not evolve in step with the system it mimics. A successful simulator can persist for the lifetime of its subject, changing to meet new requirements, to accommodate new data and methods of solution, and to reflect modifications to the system itself. Indeed, the lifetime cost of the simulator is determined primarily by the cost of its evolution. A simulation program built solely for its immediate purpose, with no thought to future uses and objectives, is unlikely to flourish. Its integrated aspects are costly to reengineer and replacement, probably after great expense, is almost certain when new requirements exceed the limits of an architecture narrowly conceived. Conversely, a robust software architecture facilitates good engineering practices and this, in turn, ensures a long period of useful service for the software, while at the same time reducing its lifetime cost.

1.1 ELEMENTS OF A SOFTWARE ARCHITECTURE

Four elements are common to nearly all simulation frameworks meant for general use: a concept of a dynamic system, software constructs with which to build models, a simulation engine to calculate a model's dynamic trajectories, and a means for control and observation of the simulation as it progresses. The concept a dynamic system on which the framework grows has a profound influence on its final form, on the experience of the end user, and on its suitability for expansion and reuse.

Monolithic modeling concepts, which were employed in the earliest simulation tools, rapidly gave way to modular ones for two reasons: (1) the workings of a large system can not be conceived as a whole. Complex operations must be broken down into manageable pieces, dealt with one at a time, and then combined to obtain the desired behavior; and (2) to reuse a model or part of a model requires that it and its components be coherent and self-contained. The near-universal adoption by commercial and academic simulation tools of modular modeling concepts, and the simultaneous growth of model libraries for these tools, demonstrates the fundamental importance of this idea.

[2] Arrillaga and Watson's *Computer Modelling of Electrical Power Systems* [6] provides an excellent example of how and where this approach can succeed. In that text, the authors build an entire simulation program, based on the principles of structured design, to solve problems that are relevant to nearly all electrical power systems.

The simulation engine produces dynamic behavior from an assemblage of components. Conceptually, at least, this is straightforward. A simulator for continuous systems approximates the solution to a set of differential equations, the choice of integration method depending on qualitative features of the system's trajectories and requirements for accuracy and precision. A discrete-event simulation executes events scheduled by its components in the order of their event times. Putting aside the details of the event scheduling algorithm and procedure for numerical integration, these approaches to simulation are quite intuitive and any two, reasonably constructed simulators provided with identical models will yield essentially indistinguishable results.

In models with discrete events—the opening and closing of switches, departure and arrival of a data packet, or failure and repair of a machine—simultaneous occurrences are often responsible for simulators that, given otherwise identical models, produce incompatible results (see, e.g., Ref. 12). This problem has two facets: intent and computational precision. The first is a modeling problem: what is the intended consequence of distinct, discrete occurrences that act simultaneously on a model? By selecting a particular solution to this problem, the simulation tool completes its definition of a dynamic system. This seemingly obscure problem is therefore of fundamental importance and, consequently, a topic of substantial research (a good summary can be found in Wieland [146] and Raczynski [113]). Simultaneous interactions are unavoidable in large, modular models, and the clarity with which a modeler sees their implications has a profound effect on the cost of developing and maintaining a simulator.

The issue of how simultaneous events are applied is distinct from the problem of deciding whether two events occur at the same time. Discrete-event systems measure time with real numbers, and so the model itself is unambiguous about simultaneous occurrences; events are concurrent when their scheduled times are equal. The computer, however, approximates the real numbers with a large, but still finite, set of values. Add to this the problem of rounding errors in floating-point arithmetic, and it becomes easy to construct a model that, in fact, does not generate simultaneous events, but the computer nonetheless insists that it does. The analysis problems created by this effect and the related issue of what to do with simultaneous actions (real or otherwise) are widely discussed in the simulation literature (again, see the article by Wieland [146] and the text by Raczynski [113]; see also Refs. 10, 107, and 130).

The concept of a dynamic system and its presentation as object classes and interfaces to the modeler are of fundamental importance. Effort expended to make these clear, consistent, and precise is rewarded in proportion to the complexity and size of the models constructed. In very small models the benefit of organization is difficult to perceive for the same reasons that structure seems unimportant when experience is confined to 100-line computer programs. For large, complicated models, however, adherence to a well-conceived structure is requisite to a successful outcome; organizing principles are important for the model's construction and its later reuse.

The modeling constructs acted on by the simulation engine are reflected in the interface it presents to the outside world. Large simulation projects rarely exist in isolation. More often, the object under study is part of a bigger system, and when the

simulator satisfies its initial purpose, this success creates a desire to reuse it in the larger context. Simulators for design can, for example, find their way into training and testing equipment, component-based simulations of a finished system, and even into the operational software of the machine that it models.

Looking beyond the very difficult problems of model validation and reuse (see, e.g., Ref. 32), issues common to the reuse of software in general can prevent an otherwise appropriate simulator from being adapted to a new context. The means for control and observation of a simulation run, and in particular the facilities for control of the simulation clock, for extracting the values of state variables, for receiving notification of important events, and for injecting externally derived inputs are of prime importance. The cost of retrofitting a simulator with these capabilities can be quite high, but they are invariably needed to integrate with a larger application.

1.2 SYSTEMS CONCEPTS AS AN ARCHITECTURAL FOUNDATION

Systems theory, as it is developed by various authors such as Ashby [7], Zeigler et al. [157], Mesarovic and Takahara [86], Wymore [149, 150], and Klir [68], presents a precise characterization of a dynamic system, two aspects of which are the conceptual foundation of our simulation framework. First is the state transition model of a dynamic system, particularly its features that link discrete-time, discrete-event, and continuous systems. Of specific interest is that discrete-time simulation, often described as a counterpart to discrete event simulation, becomes a special case of the state transition model. This fact is readily established by appeal to the underlying theory.

Second is the uniform notion of a network of systems, whereby the components are state transition models and the rules for their interconnection are otherwise invariant with their dynamics. This permits models containing discrete and continuous components to be constructed within a single conceptual framework. The consistent concept of a dynamic system—unvarying for components and networks, for models continuous and discrete—is also reflected in the facilities provided by the simulation engine for its control and observation. The conceptual framework is thereby extended to reuse of the entire simulator, allowing it to serve as a component in other simulation tools and software systems.

The small number of fundamental concepts that must be grasped, and the very broad reach of those same concepts, makes the simulation framework useful for a tremendous range of applications. It can also be used as an integrating framework for existing simulation models and as a tool for expanding the capabilities of a simulation package already in hand. Moreover, a simulation framework grounded in a broad mathematical theory can reveal fundamental relationships between simulation models and other representations of dynamic systems; the close relationship between hybrid automata, which appear frequently in the modern literature on control, and discrete-event systems is a pertinent example.

The approach taken here is not exclusive, nor is it unrelated to the established worldviews for discrete event simulation. For instance, Cota and Sargent's process

interaction worldview [29, 125] incorporates key elements of Zeigler's discrete-event system specification [152], from which the simulation framework in this book is derived. The activity-scanning worldview is apparent in models containing discrete events that are contingent on continuous variables reaching specific values. Discrete-event models constructed with any of the classic views can be components in a large model, and conversely models described within our framework can be components in other simulations. This capacity for composing a complex model from pieces in a variety of forms is, perhaps, the most attractive part of this book's approach.

1.3 SUMMARY

The modeling and simulation concepts developed in this book are illustrated with Unified Modeling Language (UML) diagrams and code examples complete enough to very nearly constitute a finished simulation engine; a finished product in C++ can be obtained by downloading the *adevs* software at `http://freshmeat.net/projects/adevs`. Implementing these simulation concepts in other programming languages is not unduly difficult.[3]

If this specific framework is not adopted, its major elements can still be usefully adapted to other simulation packages. The approach, described in Chapter 5, to continuous components can be used to build a hybrid simulator from any discrete-event simulator that embodies a modular concept of a system. Continuous system simulation tools can likewise make use of the separation of discrete-event and continuous components to integrate complex discrete-event models into an existing framework for continuous system modeling.

A programmer's interface to the simulation engine, by which the advance of time is controlled and the model's components are accessed and influenced, should be a feature of all simulation tools. Its value is attested to by a very large body of literature on simulation interoperability, and by the growing number of commercial simulation packages that provide such an interface. The interface demonstrated in this text can be easily adapted for a new simulator design or to an existing simulation tool.

Taken in its entirety, however, the proposed approach offers a coherent worldview encompassing discrete time, discrete event, and continuous systems. Two specific benefits of this worldview are its strict inclusion of the class of discrete-time systems within the class of discrete-event systems and the uniformity of its coupling concept, which allows networks to be built independent of the inner workings of their components. This unified world view, however, offers a more important, but less easily quantified, advantage to the modeler and software engineer. The small set of very expressive modeling constructs, the natural and uniform handling of simultaneity, and the resulting simplicity with which large models are built can greatly reduce the cost of simulating a complex system.

[3]Implementations in other programming languages can be found with a search for discrete-event (system) simulation (DEVS) and simulation on the World Wide Web.

1.4 ORGANIZATION OF THE BOOK

Chapter 2 motivates major aspects of the software design, the inclusion of specific numerical and discrete simulation methods, and other technical topics appearing in the subsequent chapters. The robotic tank developed in Chapter 2 has three important facets: (1) it is modeled by interacting discrete-event and continuous subsystems, (2) the parts are experimented with individually and collectively, and (3) its simulator is used both interactively and for batch runs.

Chapter 3 introduces state transition systems, networks of state transition systems, and builds from these concepts the core of a simulation engine. This is done in the simple, almost trivial, context of discrete-time systems, where fundamental concepts are most easily grasped and applied. The software is demonstrated with a simulator for cellular automata.

Chapter 4 builds on this foundation, introducing discrete-event systems as a generalization of discrete-time systems. Using these new concepts, the simulation engine is expanded and then demonstrated with a simulator for the computer that controls the robotic tank introduced in Chapter 2. Chapter 4 also revisits the cellular automata from Chapter 3 to show that they are a special case of asynchronous cellular automata, which are conveniently described as discrete-event systems.

Chapter 5 completes the simulation framework by introducing continuous systems. Numerical techniques for locating state events, scheduling time events, and solving differential equations are used to construct a special class of systems having internal dynamics that are continuous, but that produce and consume event trajectories and so are readily incorporated into a discrete-event model. The simulation framework from Chapter 4 is expanded to include these new models, and the whole is demonstrated with a complete simulator for the robotic tank. The cellular automata are again revisited, and it is shown that the asynchronous cellular automata of Chapter 4 are, in fact, a special case of differential automata, which have attracted considerable attention in recent years.

Chapter 6 has examples of engineering problems that exemplify different aspects of the simulation technology. The book concludes with a discussion of open problems and directions for future research. The appendixes contain supplemental material on the design and test of simulation models, the use of parallel computers for simulating discrete-event systems, and a brief introduction to system homomorphisms as they are used in the running discussion of cellular automata.