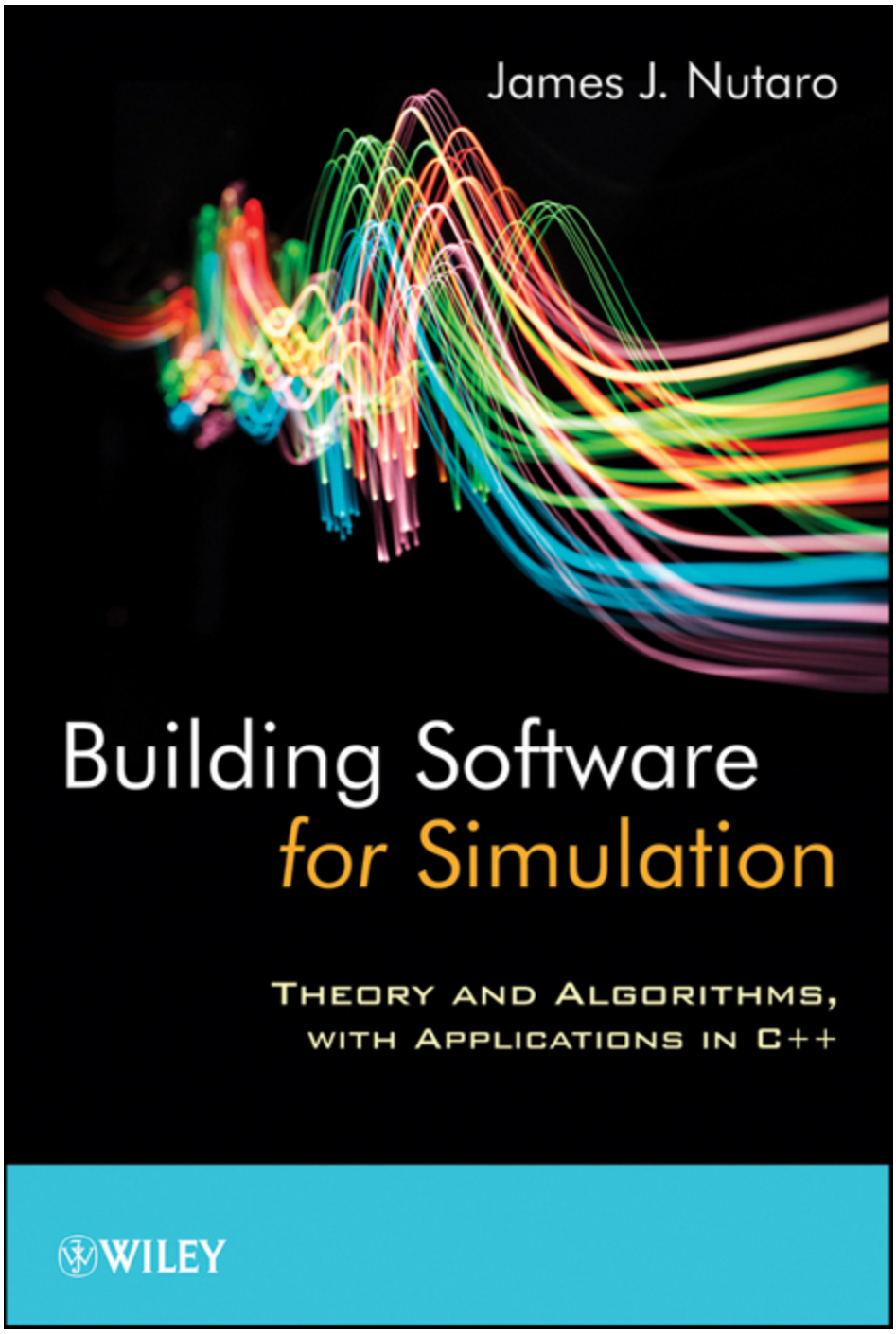
James J. Nutaro

Building Software *for Simulation*

Theory and Algorithms, with Applications in C++

WILEY


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

Contents

BUILDING SOFTWARE FOR SIMULATION

BUILDING SOFTWARE FOR SIMULATION

Theory and Algorithms, with Applications in C++

JAMES J. NUTARO
Oak Ridge National Laboratory

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic format. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Nutaro, James J.
Building software for simulation: theory and algorithms with applications in C++ / James J. Nutaro
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-41469-9 (cloth)

PREFACE

Building Software for Simulation is different from many other books on simulation because its focuses on the design and implementation of simulation software; by culminating in a complete system for simulation, this book makes itself unique. The design and construction of simulation software has been a topic persistently absent from textbooks even though many, if not most, simulation projects require the development of new software. By addressing this important topic, *Building Software for Simulation* will, I hope, complement other excellent textbooks on modeling and simulation. This book is intended as both an introduction to simulation programming and a reference for experienced practitioners. I hope you will find it useful in these respects.

This book approaches simulation from the perspective of Zeigler's theory of modeling and simulation, introducing the theory's fundamental concepts and showing how to apply these to problems in engineering. The original concept of the book was not so ambitious; its early stages more closely resembled a cookbook for building simulators, focusing almost exclusively on algorithms, examples of simulation programs, and guidelines for the object-oriented design of a simulator. The book retains much of this flavor, demonstrating each concept and algorithm with working code. Unlike a cookbook, however, concepts and algorithms discussed in the text are not disembodied; their origins in the theory of modeling and simulation are made apparent, and this motivates and provides greater insight into their application.

Chapters 3, 4, and 5, are the centerpiece of the text. I begin with discrete-time systems, their properties and structure, simulation algorithms, and applications. Discrete-time system will be familiar to most readers and if not, they are easily grasped. Discrete-time systems are generalized to introduce discrete event systems; this approach leads

naturally to Zeigler's discrete-event system specification, its properties and structures, and simulation procedures. The central three chapters conclude with methods for modeling and simulating systems that have interacting continuous and discrete-event dynamics.

The three main chapters are bracketed by applications to robotics, control and communications, and electrical power systems. These examples are more complicated than might be expected in a textbook; three examples occupy two complete chapters. They are, however, described in sufficient detail for a student to reproduce the printed results and to go a step further by exploring unanswered questions about the example systems. The book's appendixes discuss technical problems that do not fit cleanly into the narrative of the manuscript: testing and design, parallel computing, and a brief review of mathematical topics needed for the examples.

Many people contributed advice and guidance as the book evolved. I am particularly grateful to Vladimir Protopopescu at Oak Ridge National Laboratory for his review of and critical commentary on my rough drafts; his advice had a profound impact on the organization of the text and my presentation of much of the material. I'm also grateful to Angela, who reviewed very early drafts and remarked only rarely on the state of the yard and unfinished projects around the house. Last, but not least, thanks to Joe and Jake, who, in the early morning hours while I worked, quietly (for the most part) entertained themselves.

JIM NUTARO

Oak Ridge, Tennessee
December 2009

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 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. 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.

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.

[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.

[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.

[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.

CHAPTER 2

FIRST EXAMPLE: SIMULATING A ROBOTIC TANK

This example serves two purposes. First, it illustrates how hybrid dynamics can appear in engineering problems. The model has three main parts: the equations of motion, a model of the propulsion system, and a model of the computer. The first two are piecewise continuous with discontinuities caused by step changes in the motor voltage and the sticking friction of the rubber tracks. The third model is a prototypical example of a discrete-event system; the tank's computer is modeled with an interruptible server and queue. The equations of motion, propulsion system, and computer are combined to form a complete model of the tank.

Second, this example illustrates the basic elements of a software architecture for large simulation programs. The simulation engine is responsible solely for calculating the dynamic behavior of the model; other functions (visualization and interactive controls, calculation of performance metrics, etc.) are delegated to other parts of the software. This approach is based on two patterns or principles: model–view–control and the experimental frame.

Model–view–control is a pattern widely used in the design of user interfaces (see, e.g., Refs. 47 and 101); the simulation engine and model are treated as a dynamic document and, with this perspective, the overarching design will probably be familiar to most software engineers. The experimental frame (as described, e.g., by Daum and

Sargent [31])[1] is a logical separation of the model from the components of the program that provide it with input and observe its behavior. These principles simplify reuse; programs for two experiments illustrate how they are applied and the benefit of doing so.

The entirety of this example need not be grasped at once, and its pieces will be revisited as their foundations are established in later chapters. Its purpose here is to be a specific example of how the simulation engine is used, and to motivate the software architecture and algorithms that are discussed in the subsequent chapters of this book.

2.1 FUNCTIONAL MODELING

Fishwick [42] defines a functional model as a thing that transforms input into output. This view of a system is advantageous because it leads to a natural decomposition of the simulation software into objects that implement precisely defined transformations. Distinct functions within the model are described by distinct functional blocks which are connected to form a complete model of the system. The software objects that implement the functional blocks are connected in the same way to build a simulator.

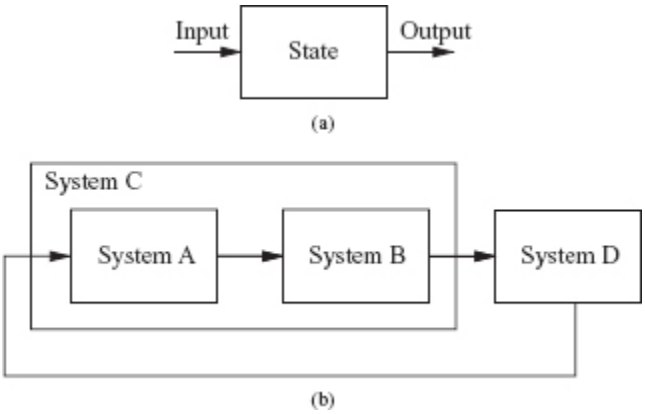
There are numerous methods for designing models. Many of them are quite general: bond graphs and state transition diagrams, for instance. Others are specific to particular problems: the mesh current method for electric circuits and the Lagrangian formulation of a rigid body. The majority of methods culminate in a state space model of a system: a set of state variables and a description of their dynamic behavior. Mathematical formulations of a state space model can take the form of, for example, differential equations, difference equations, and finite-state machines.

To change a state space model into a functional model is simple in principle. The state variables define the model’s

internal state; state variables or functions of state variables that can be seen from outside the system are the model's output; variables that are not state variables but are needed for the system to evolve become the model's input. In practice, this change requires judgment, experience, and a careful consideration of sometimes subtle technical matters. It may be advantageous to split a state space model into several interacting functional models, or to combine several state space models into a single functional model. Some state space models can be simplified to obtain a model that is easier to work with; simplification might be done with precise mathematical transformations or by simply throwing out terms. The best guides during this process are experience building simulation software, familiarity with the system being studied, and a clear understanding of the model's intended use.

Functional models and their interconnections are the specification for the simulation software. For this purpose, there are two types of functional model: atomic and network. An atomic model has state variables, a state transition function that defines its internal response to input, and an output function that transforms internal action into observable behavior. A network model is constructed from other functional models, and the behavior of the network is defined by the collective behavior of its interconnected components. The simulator is built from the bottom up by implementing atomic models, connecting these to form network models, combining these network models to create larger components, and repeating until the software is finished. This bottom–up approach to model construction is illustrated in Figure 2.1.

FIGURE 2.1 Bottom–up construction of a model from functional pieces: (a) input, output, and internal state of an atomic model; (b) a network model constructed from three atomic models.



The simulation engine operates on software objects that implement atomic and network models. To build a simulator therefore requires the parts of a dynamic system to be expressed in this form. Functional models need not be built in a single step. Atomic and network models are more easily obtained by a set of steps that start with an appropriate modeling technique, proceed to a state space description of the model's fundamental dynamics, combine these to create more sophisticated components, and end with a—possibly large—functional model that can be acted on by the simulation engine.

2.2 A ROBOTIC TANK

The robotic tank is simple enough to permit a thorough discussion of its continuous and discrete dynamics, but sufficiently complicated that it has features present in larger, more practical systems. The robot's operator controls it through a wireless network, and the receipt, storage, and processing of packets is modeled by a discrete event system. An onboard computer transforms the operator's commands into control signals for the motors. The motors and physical motion of the tank are modeled as a continuous system. These components are combined to create a complete model of the tank.

Our goal is to allocate the cycles of the tank's onboard computer to two tasks: physical control of the tank's motors and processing commands from the tank's operator. The tank has four parts that are relevant to our objective: the radio that receives commands from the operator, the computer and software that turn these commands into control signals for the motors, the electric circuit that delivers power to the motors, and the gearbox and tracks that propel the tank. The tank has two tracks, left and right, each driven by its own brushless direct-current (DC) motor. A gearbox connects each motor to the sprocket wheel of its track. The operator drives the tank by setting the duty ratio of the voltage signal at the terminals of the motors. The duty ratio are set using the control sticks on a gamepad and sent via a wireless network to the computer.

The computer generates two periodic voltage signals, one for each motor. The motor's duty ratio is the fraction of time that it is turned on in one period of the signal (i.e., its ON time). Because the battery voltage is fixed, the power delivered to a motor is proportional to its duty ratio. Driving the tank is straightforward. If the duty ratio of the left and right motors are equal then the tank moves in a straight line. The tank spins clockwise if the duty ratio of the left motor is higher than that of the right motor. The tank spins counterclockwise if the duty ratio of the right motor is higher than that of the left motor. A high duty ratio causes the tank to move quickly; a low duty ratio causes the tank to move slowly.

If the voltage signal has a high frequency, then the inertia of the motor will carry it smoothly through moments when it is disconnected from the batteries; the motors operate efficiently and the tank handles well. If the frequency is too low, then the motor operates inefficiently. It speeds up when the batteries are connected, slows down when they are disconnected, and speeds up again when power is

reapplied. This creates heat and noise, wasting energy and draining the batteries without doing useful work. Therefore, we want the voltage signal to have a high frequency.

Unfortunately, a high-frequency signal means less time for the computer to process data from the radio. If the frequency is too high, then there is a noticeable delay as the tank processes commands from the operator. At some point, the computer will be completely occupied with the motors, and when this happens, the tank becomes unresponsive.

Somewhere in between is a frequency that is both acceptable to the driver and efficient enough to give a satisfactory battery life. There are physical limits on the range of usable frequencies. It cannot be so high that the computer is consumed entirely by the task of driving the motors. It cannot be so low that the tank lurches uncontrollably or overheats its motors and control circuits. Within this range, the choice of frequency depends on how sensitive the driver is to the nuances of the tank's control.

An acceptable frequency could be selected by experimenting with the real tank; let a few people drive it around using different frequencies and see which they like best. If we use the real tank to do this, then we can get the opinions of a small number of people about a small number of frequencies. The tank's batteries are one constraint on the number of experiments that can be conducted. They will run dry after a few trials and need several hours to recharge. That we have only one tank is another constraint. Experiments must be conducted one at a time. If, however, we build a simulation of the tank, then we can give the simulator to anyone who cares to render an opinion, and that person can try as many different frequencies as time and patience permit.

2.2.1 Equations of Motion

The model of the tank's motion is adapted from Anh Tuan Le's PhD dissertation [74]. The model's parameters are listed in Table 2.1, and the coordinate system and forces acting on the tank are illustrated in Figure 2.2. The model assumes that the tank is driven on a hard, flat surface and that the tracks do not slip. The position of the tank is given by its x and y coordinates. The heading θ of the tank is measured with respect to the x axis of the coordinate system and the tank moves in this direction with a speed v.

FIGURE 2.2 Coordinate system, variables, and parameters used in the tank's equations of motion.

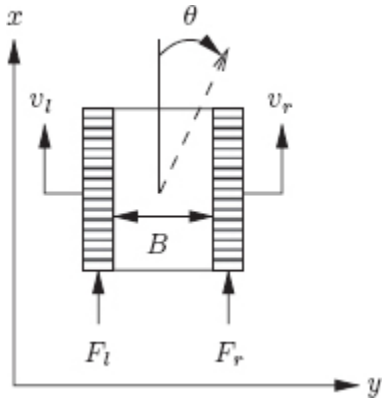


TABLE 2.1 Value of Parameters Used in the Tank's Equations of Motion

Parameter	Value	Description
m_t	0.8 kg	Mass of the tank
J_t	5×10^{-4} kg · m^2	Angular mass of the tank
B	0.1 m	Width of the tank from track to track
B_r	1.0 N · s / m	Mechanical resistance of the tracks to rolling forward
B_s	14.0 N · s / m	Mechanical resistance of the tracks to sliding forward
B_l	0.7 N · m · s / rad	Mechanical resistance of the tracks to turning
S_l	0.3 N · m	Lateral friction of the tracks

The left track pushes the tank forward with a force F_l; the right track, with a force F_r; and B_r is the mechanical resistance of the tracks to rolling. The tank uses skid steering; to turn, the motors must collectively create enough torque to cause the tracks to slide sideways. This

requires overcoming the sticking force S_l. When sufficient torque is created, the vehicle begins to turn. As it turns, some of the propulsive force is expended to drag the tracks laterally; this is modeled by an additional resistance B_l to its turning motion and B_s to its rolling motion.

The tank's motion is described by two sets of equations, one for when the tank is turning and one for when it is not. The switch from turning to not turning (and vice versa) has two discrete effects: (1) the angular velocity ω changes instantaneously to and remains at zero when the tracks stick and the turn ends, and (2) the rolling resistance of the tank changes instantaneously when the tank starts and ends a turn. The Boolean variable *turning* is used to change the set of equations. The equations that model the motion of the tank are

$$turning = \begin{cases} \text{true} & \text{if } \frac{B}{2}|F_l - F_r| \geq S_l \\ \text{false} & \text{otherwise} \end{cases} \tag{2.1}$$

$$\dot{v} = \begin{cases} \frac{1}{m_t}\left(F_l + F_r - (B_r + B_s)v\right) & \text{if } turning = \text{true} \\ \frac{1}{m_t}\left(F_l + F_r - B_r v\right) & \text{if } turning = \text{false} \end{cases} \tag{2.2}$$

$$\dot{\omega} = \begin{cases} \frac{1}{J_t}\left(\frac{B}{2}(F_l - F_r) - B_l\omega\right) & \text{if } turning = \text{true} \\ 0 & \text{if } turning = \text{false} \end{cases} \tag{2.3}$$

$$\dot{\theta} = \omega \tag{2.4}$$

$$\dot{x} = v\sin(\theta) \tag{2.5}$$

$$\dot{y} = v\cos(\theta) \tag{2.6}$$

$$\text{If } turning = \text{false then } \omega = 0 \tag{2.7}$$

When *turning* changes from false to true, every state variable evolves from its value immediately prior to starting the turn, but using the equations designated for *turning* = true. When *turning* changes from true to false, every state