Dynamics Simulation and Controller Interfacing for Legged Robots

Jesse A. Reichler
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
reichler@cs.uiuc.edu

Fred Delcomyn
Neuroscience Program
Department of Entomology
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
delcomyn@life.uiuc.edu

Abstract

Dynamics simulation can play a critical role in the engineering of robotic control code, and there exist a variety of strategies both for building physical models and for interacting with these models. This paper presents an approach to dynamics simulation and controller interfacing for legged robots, and contrasts it to existing approaches.

We describe dynamics algorithms and contact-resolution strategies for multibody articulated mobile robots based on the decoupled tree-structure approach, and present a novel scripting language that provides a unified framework for control-code interfacing, user-interface design, and data analysis. Special emphasis is placed on facilitating the rapid integration of control algorithms written in a standard object-oriented language (C++), the production of modular, distributed, reusable controllers, and the use of parameterized signal-transmission properties such as delay, sampling rate, and noise.

KEY WORDS—simulation, walking, dynamics, control, interfacing, robotics

1. Introduction

Although the increase in complexity and sophistication of robot-control algorithms signals the continuing progress being made in control systems research, the practical impediments to progress in this field are also growing. These impediments can be severe for researchers interested in the control of walking robots. This is true not only because the physical construction of appropriate robots in this case is especially difficult, time consuming, and expensive, but because the evaluation of controllers for such robots often requires prolonged training and frequent reconfiguration of both the controllers and the physical device, which requires the intervention of a human operator. A major additional impediment is the need to have control code operating in real time throughout the development cycle. This requirement becomes increasingly difficult and costly to meet as control algorithms grow more complex and need to process more extensive sensor information.

Simulation offers a potential solution to these problems by replacing the mechanical plant with a computer model. Traditionally cited benefits of simulation include the reduced cost of design and testing, the ability to conduct controlled repeatable experiments, and the ability to record precise and voluminous data. Indeed, simulation currently plays an important role in many aspects of robotics research, especially in mechanical design (Davidson 1996; Hollars, Rosenthal, and Sherman 1994; Marhefka and Orin 1996; Fisette, Peterkenne, and Smain 1998; Zeghloul, Blanchard, and Ayrault 1997; Elmqvist, Mattson, and Otter 1998), the validation of large-scale robotic devices (Ma et al. 1997), and in computer animation (Baca 1998; Multon, Cani-Gascuel, and Debunne 1999; Armstrong and Green 1985).

Despite the sophistication of existing simulators, most are not well suited for researchers interested in locomotor control of legged robots. Such researchers are currently forced to choose between “hyper” realistic mechanical simulators, which are slow, expensive, and not amenable to control-code experimentation, and mechanically simplistic simulators designed for high-level robotic task planning. For these researchers, what is needed is an appropriately realistic dynamics-simulation system that is specifically designed to facilitate rapid interfacing and testing of complex, nontraditional control algorithms (Reichler and Delcomyn 1998).

The International Journal of Robotics Research
Vol. 19, No. 1, January 2000, pp. 41-57,
©2000 Sage Publications, Inc.

In this paper, we explore several issues in the efficient simulation of walking robots, including contact-resolution strategies and strategies for interfacing with robotic simulators. We present a freely available simulation system designed to facilitate the rapid development and testing of control algorithms.

2. Overview of the Simulation System

Our simulation system consists of two main components: a dynamics engine, and an interfacing language. The dynamics engine is responsible for performing all mechanical simulation and contact resolution. The interfacing language component comprises modules for describing and supervising all aspects of a simulation experiment, including the physical configurations of robots and the environment, the attachment of sensors and actuators, the graphical user interface, datafile management, and the incorporation of and communication with external control algorithms. Figure 1 shows two different experiments run using the simulation system, one involving a three-degree-of-freedom robot arm and another involving a walking hexapod. It demonstrates the flexibility that the interfacing language brings to the design of custom experiments.

The entire simulation system consists of about 500 K lines of C++ source code, and has been compiled on a variety of Unix platforms. The X windows and OpenGL graphic libraries (the public-domain Mesa package for Linux) are used for visualization. On a 400-MHz Pentium II, simulation speed ranges from super real time when simulating a simple robot arm with visualization disabled, to 10^{-3} real time when simulating a complex mobile robot with sensors and actuators and solid three-dimensional visualization enabled.

In the following sections, we describe the two main components of the simulation system, namely the dynamics engine and the interfacing language, and explore the design choices faced when building a simulator for walking robots.

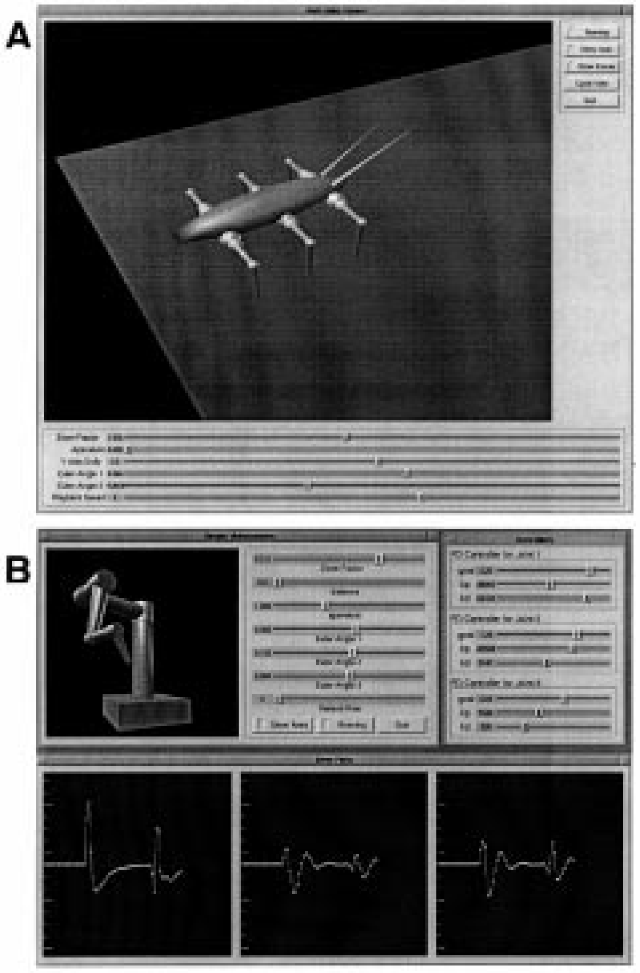

Fig. 1. Typical graphical displays during a simulation run, illustrating how the user interface can be customized to suit the nature of different experiments: (a) hexapod walker in the form of an insect, displayed during a movie playback; and (b) three-degree-of-freedom robot arm, with controls for the target angles and PD constants at each joint. Lower plots show deviations from target-joint angles over time.

3. The Dynamics Engine

Dynamics simulation involves the modeling of objects with mass and inertia as they interact with the environment and are affected by gravity and other forces. While kinematics simulators, which model trajectories and positions but not forces, are occasionally used to explore basic locomotor movements, only a three-dimensional dynamics simulation allows a researcher to investigate the ability of a control algorithm to maintain stability and locomote successfully under realistic conditions. A dynamics simulator takes as input the current state of the world and the applied controller forces, and outputs the subsequent state of the world.

The equations for exact dynamic simulation of rigid single-body objects and for simple chains of such objects that do not come into contact with other structures have long been known, and can be performed efficiently. However, for complicated multibody objects that come into contact with one another, exact simulation becomes difficult and time consuming. Dynamics simulation, like most other simulation paradigms, demands trade-offs of accuracy for speed and computational simplicity. In fact, much work in robotic simulation is devoted to elaborating and justifying efficient approximate algorithms for multibody simulation (Mirtich 1996; Manko 1992). The nature of the approximations and heuristics that are appropriate for a specific dynamics simulator depend heavily on the domain under study.

3.1. Handling Constraints

The critical factor affecting computational complexity in dynamics simulation is the accurate simulation of constraints. Constraints exist both as internal properties of a robot, where multiple rigid segments connect through joints, and as contact events between structures; these are constraints because they limit the motion of the structures. Different dynamics algorithms tackle the simulation of constraints differently. At one extreme are the purely analytical methods, which involve solving the equations of motion explicitly and exactly. For chains of rigid bodies connected through stereotyped joints and attached to a fixed base, these analytical methods can be performed efficiently (Lilly 1993), typically involving the solution of a small system of equations. An analytical solution becomes more difficult when more generic constraints and a mobile base are included. For a specific robotic structure within a specific, restricted environment, and in cases where simulation accuracy is critical and computation speed is not, such as in the validation of existing robotic mechanisms, analytical solutions may be appropriate. At the other extreme of realism lie purely heuristic methods. Heuristic methods are meant to capture the qualitative aspects of the physical world but trade accuracy for computational simplicity and speed. For some applications, such as animation, heuristic methods are commonplace and can produce believable visual simulations with the use of the simplest of simple constraint models (McKenna and Zeltzer 1990).

3.2. The Decoupled Tree-Structure (DTS) Approach

The dynamics algorithms we use for our simulator are based on the decoupled tree-structure approach (DTS) described by Freeman and colleagues (Freeman and Orin 1991; Freeman 1989; McMillan, Orin, and McGhee 1996), which is itself based on a number of well-established methods (Featherstone 1987; Walker and Orin 1982; Lilly 1993; Shih, Frank, and Ravani 1987; Brandl, Johanni, and Otter 1986). The DTS approach supports robots consisting of a series of branching chains (like legs or arms) connected to a single "reference member" that can be fixed or mobile. Chains themselves consist of a series of segments (limbs) connected via joints. Joints are described using modified Denavit-Hartenberg (MDH) parameters (Lilly 1993). The MDH parameters represent a compact and computationally friendly formalism (McMillan, Orin, and McGhee 1995) for describing how successive segments relate kinematically to one another (see Fig. 2). Although the DTS approach is amenable to arbitrarily branching chains and generalized joints (see, for example, the work of McMillan, Orin, and McGhee (1996)), additional parameters are needed to specify appropriate degrees of freedom, and our simulator is currently limited to serial (nonbranching) chains and revolute joints, which are completely specified by the MDH parameters.

The main steps of the DTS computations are shown in Figure 3, based on Freeman and Orin (1991). The first step is a forward-leg kinematics recursion that propagates kinematic information from the body to the end segments. A backward-

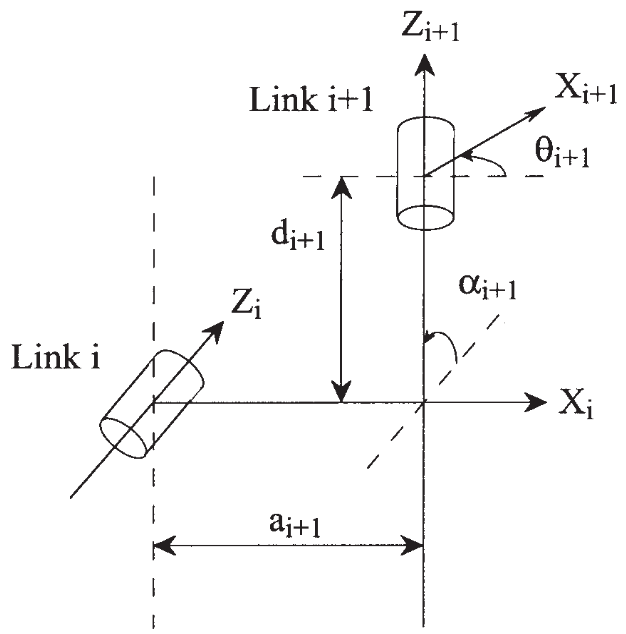

Fig. 2. Modified Dennavit Hartenberg parameters. Links are numbered starting at zero for the hip or body, and increasing outward. The modified scheme differs from the standard DH scheme mainly in that the parameters relate to the position and orientation of the frame preceding the link segment, rather than the frame following the link segment. Coordinate axes are aligned with Z_i along the axis of motion of joint i and X_i along the common normal between the extensions of Z_i and Z_{i+1}. The angle θ_i represents the angle about Z_{i+1} between X_i and X_{i+1}; length a_i is the perpendicular distance along X_i between Z_i and Z_{i+1}; offset d_i represents the perpendicular distance along Z_{i+1} between X_i and X_{i+1}; and twist α_i is the angle about X_i between Z_i and Z_{i+1}. (Adapted from Lilly (1993).)

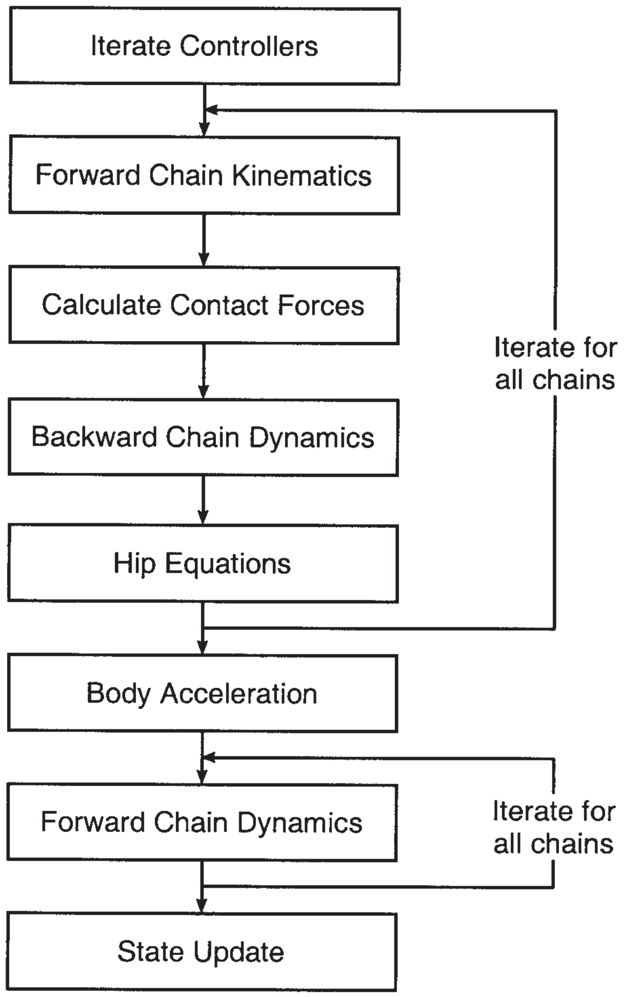

Fig. 3. The steps of the DTS method. The first step is a forward-leg kinematics recursion that propagates kinematic information from the body to the end segments. A backward-dynamics recursion then propagates forces from the chains (including contact forces and driven torques) back up to the hips of the body. The hip forces from the separate chains are then combined and used to update the acceleration of the reference body. The computed body acceleration is then used in a forward-dynamics recursion to determine the joint accelerations of the chains. Lastly, body and joint accelerations are integrated to yield the next state. (Adapted from Freeman (1989).)

dynamics recursion then propagates forces from the chains (including contact forces and driven torques) back up to the hips of the body. The hip forces from the separate chains are combined and used to update the acceleration of the reference body. The computed body acceleration is then used in a forward-dynamics recursion to determine the joint accelerations of the chains. Lastly, body and joint accelerations are integrated to yield the next state. The time complexity for these computations is $O(mN)$ in the total number of legs, m, and segments, N (Freeman and Orin 1991; McMillan, Orin, and McGhee 1995).

The DTS approach represents a balance between heuristic and analytical methods. Segments are considered tightly coupled to one another through joints, and chains are considered tightly coupled to the reference member body, meaning that these constraints are not at any time violated by the dynamics algorithms. The exact propagation of forces through the constraints between the segments and body of an articulated robot is possible because the equations of motion for these constraints are well understood and efficient implementations are known.

Contact with external objects, on the other hand, is modeled as a "decoupled" or compliant interaction, such that contact forces are calculated heuristically to approximate the expected forces. The motivation for employing this scheme, which is illustrated in Figure 4, is discussed in the next section.

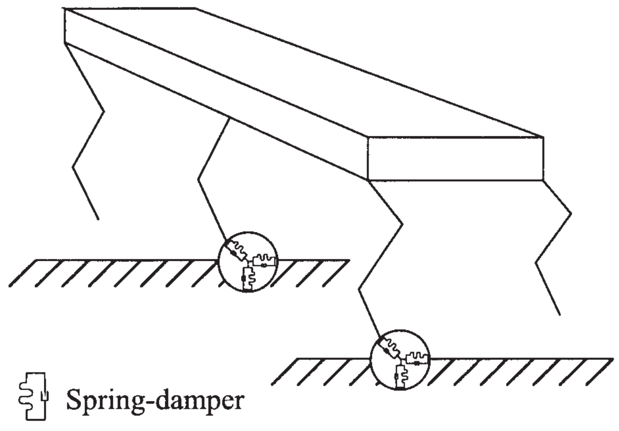

Fig. 4. Conceptualization of the DTS approach to simulation of a multilegged robot. Each leg is tightly coupled to the base, and each joint is considered to be powered. Ground contact is modeled using a penalty-based spring-and-damper method. It is the approximation of contact forces using compliant penalty methods that makes the DTS approach computationally efficient. (Adapted from Freeman (1989).)

3.3. Penalty-Based Contact Resolution

Precise detection of contacts and modeling of contact forces is complex, time consuming, and in the general case, an open area of research. There is no single best approach to simulating contact dynamics in arbitrary environments; the key is finding an approach that is appropriate for the domain under study. To reduce computational complexity, DTS uses a simple, penalty-based method of approximating contact forces between segments and the environment. Penalty-based methods treat contact between objects as compliant, so that the constraint of nonpenetration is "softened" and objects are allowed to interpenetrate. As objects interpenetrate, they generate perpendicular repellant forces that increase with the penetration distance (and possibly with velocity and acceleration) as if being pulled by a spring attaching the two objects.

Detecting arbitrary collisions within a system of polygonal volumes can be quite time consuming, and a substantial amount of effort, both in robotics and animation, has gone into developing sophisticated algorithms for the detection of collisions. Most methods involve building a dynamic database that keeps track of the small number of object pairs that could possibly collide in the near future, and checking only those pairs for actual contact. Fortunately, we can take advantage of the limited contact that normally occurs during locomotion to build a simple and efficient contact-detection system. We ignore the possibility of interlimb contact, and represent the ground using a simple fixed-spacing triangular tiling, where vertex heights define the topography of the substrate. Robots are assembled from simple geometric primitives, such as blocks, cones, and spheres, and each primitive component of a robot has a number of "boundary points" on its surface, which are checked for penetration of the substrate. Detecting penetration is remarkably simple, because for any given boundary point there is only one triangle tile it could possibly be in contact with, and that tile can be indexed directly given the point coordinates. By adjusting the number of points used on each geometric primitive, we can further tune the trade-off between accuracy and speed.

When a boundary point first breaks the ground surface, the initial point of interpenetration is recorded. This recorded "hookpoint" then acts like the anchor of a virtual spring, exerting a force that pulls the object out of the ground. Force increases as a function of interpenetration distance and velocity. (The velocity component plays the role of a damper, which is why this method is sometimes called the spring-damper method.) The ratio of normal to tangential forces determines whether the tangential force is sufficient to overcome the effects of friction that oppose movement. If the tangential force escapes some threshold, slipping is simulated by allowing the recorded hookpoint to slide along the ground surface, changing the position of the virtual spring for the next contact computation. By adjusting the spring and damper constants of a particular tile, the stiffness of that surface patch is adjusted. By adjusting the static friction threshold, the friction of the surface is adjusted.

Penalty-based methods are commonly used and commonly criticized (Mirtich 1996; Baraff 1992). The key difficulty with the penalty method is finding penalty constants (representing the stiffness of the virtual springs) that are consistently effective for different objects in different environments. These penalty terms are typically set using trial and error. If they are set too high, objects may rebound off one another inappropriately; if they are set too low, objects may sink into one another unacceptably. A complicating factor is that the choice of penalty terms interacts with the choice of the integration time step; to keep penetration to a minimum, penalty terms need to be set as high as possible, but high penalty terms imply large contact forces and these large contact forces demand small integration time steps.

Interestingly, while the common rule of thumb in setting penalty constants is to make them as high as possible, we have observed a phenomenon in our locomotor simulations that contradicts this intuition. Recall that hookpoints persist during prolonged contact, and are destroyed once an object ceases to interpenetrate the ground. In locomotion, it is the tangential springlike forces generated at the hookpoints that prevent feet from slipping. Therefore, contrary to the conventional notion that penalty terms should be set as high as possible to approximate a perfect nonpenetration constraint, it appears critical in locomotion that the penalty terms be set low enough to maintain interpenetration throughout normal stepping. This observation suggests a possible extension to the spring-damper penalty method that would explicitly take into account some nominal interpenetration distance, while maintaining high penalty-term constants beyond that distance.

Despite their limitations, penalty methods carry substantial advantages, chiefly their computational simplicity and efficiency, the ease of incorporating static friction models, and the ability to simulate qualitatively a variety of surface characteristics.

3.4. Alternatives to Penalty-Based Methods

There are alternatives to penalty-based models of contact resolution. For simple systems of perfectly rigid bodies contacting without friction, it may be possible to solve the equations of motion analytically. In practice, however, analytical methods are rarely employed in three-dimensional dynamics simulations. The computations are too complicated, and the assumptions of perfectly rigid bodies interacting without friction are too restrictive. A relatively recent approach to contact resolution that is gaining in popularity is the so-called "impulse-based" model (Mirtich 1996). In the impulse-based model, nonpenetration is strictly enforced. As in the penalty-based approach, state variables are integrated over discrete time steps. However, in impulse-based modeling, the basic strategy is to calculate estimated collision times for (otherwise) ballistic objects, integrate in between these predicted collision events, and calculate, at the moment of each collision, a pair of equal and opposite impulsive forces that when applied to the objects will preserve the nonpenetration constraint. For systems of rigid, passive objects undergoing momentary collisions, the performance of impulse-based simulations is extremely impressive. Most importantly, such simulations do not require the hand tuning of stiffness parameters the way that penalty methods do.

Despite the obvious success of impulse-based contact resolution, it should be remembered that this method is not an exact model of the underlying reality in the way that analytical methods are; the method possesses a number of shortcomings that are particularly troubling when used for the simulation of locomotion. The problems stem from the nature of the contacts that dominate locomotion. First, walking robots are multibody, nonpassive systems. Formulations of impulse-based algorithms for handling multibody, tree-shaped systems exist (Mirtich 1996), but are significantly more complex than the basic impulse-based calculations, which are already quite complex. The computations for multibody systems are also time consuming, a fact that takes on greater significance in light of the locomotor domain. To see why this is true, recall that impulse-based methods depend on being able to calculate the occurrence times of sporadic collision events. When no two objects are in contact over some integration period, computations are simple and fast, but the moment that contact occurs, the system must truncate the integration step and perform extensive calculations of the impulsive forces. In a system where collisions are momentary and rare, this scheme works well. But in locomotion, it is likely that some part of the robot will be in contact with the ground at every instant in time, bringing the impulse computations to a virtual standstill.

An equally troubling fact concerns the nature of the contacts that dominate locomotion. The foundation of impulse-based methods of contact resolution is the treatment of contacts as momentary collisions, where two bodies are made to separate by the employment of a brief impulsive force. Locomotion, however, is characterized by a marked lack of discrete, momentary collisions, and is instead dominated by a preponderance of prolonged, static contact events with substantial tangential forces that are dependent on frictional modeling.

Heuristic extensions to the impulse-based approach have been advanced to support static contact (Mirtich 1996). The basic idea behind these extensions is straightforward: two objects, which in the real world would be in prolonged static contact, are simulated as if they were constantly separating and recolliding. Unfortunately, these heuristic methods suffer from some of the same drawbacks that penalty-based methods do (such as requiring manual parameter tuning). And while they work well when simulating a passive object on a flat surface, a phenomenon known as "creeping" can occur when an object in static contact experiences tangential forces. Mirtich (1996) illustrates this with the example of simulating a block resting on a sloped platform (see Fig. 5). The friction of the slope should keep the block from moving, but in the simulation, the block gradually creeps down the incline. This happens because constant microcollisions are, in effect, bouncing the block off the surface at a high frequency. This means that the frictional forces that would normally prevent the block from sliding are only acting intermittently (during collisions but not while the block is airborne); thus, the block slides, regardless of the frictional properties of the surface. The phenomenon of creeping is particularly troublesome for the simulation of locomotion, because in locomotion the feet are constantly exerting tangential forces while in static contact with the ground.

In many ways, the impulse-based approach and the penalty method are complementary. The impulse-based approach excels at simulating occasional, brief rigid-body collisions, but has difficulty adequately simulating frequent, prolonged contact. The penalty method, in contrast, is a poor choice for simulating brief rigid-body collisions, which demand high spring constants that produce unstable accelerations, but provides an efficient and flexible qualitative model of compliant, prolonged contact. Although recent developments in impulse-based contact resolution represent a tremendous advance toward efficient general-purpose contact resolution, we doubt the appropriateness of impulse-based simulation (at least in its current form) for the simulation of locomotor dynamics. In contrast, we argue that the penalty method has a number of characteristics that make it a good candidate for contact resolution when performing locomotor simulations: it is trivial to

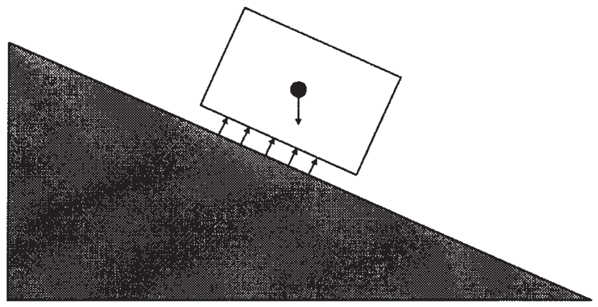

Fig. 5. Illustration of "creeping." The friction of the slope should keep the block from sliding downward, but the use of micro-impulses to prevent interpenetration can end up bouncing the block off the surface at a high frequency. This means that the frictional forces that would normally prevent the block from sliding are only acting intermittently (during collisions but not while the block is airborne), and the block gradually falls down the slope due to the force of gravity, regardless of the frictional properties of the surface. (Adapted from Mirtich (1996).)

implement, is amenable to parallelization, supports multiple simultaneous constraints with no modification, and is easily parameterized to suit a variety of surface properties such as friction and stiffness.

3.5. Integration Methods

The job of the dynamics algorithms is to compute the current linear and rotational accelerations of all structures. To be useful for simulation, these accelerations must be used to produce ongoing estimates of the velocities and positions of objects. There are several ways this can be done. The simplest way, known as Euler's method, simply integrates accelerations over discrete time steps to yield velocity estimates, and integrates velocity to yield position. Because the accelerations are integrated over discrete time steps, the dynamic simulation is necessarily an approximation of the physical world, and one is again faced with having to balance speed and simplicity of computation against accuracy of simulation. The less frequently it is necessary to recalculate accelerations (i.e., the larger the time step), the faster the simulation will run. But since conditions that might have changed within a single time period are essentially ignored, the larger the time step, the less accurate the calculations. Euler's method is computationally trivial, and in the limit (as time step size is reduced) will converge to the true solution (Garcia de Jalon and Bayo 1994). In practice, however, choice of step size is difficult.

The problem of integrating differential equations is not by any means unique to dynamics simulation—it arises in a wide variety of engineering applications and many sophisticated schemes have been proposed to perform it efficiently (see, for example, the work of Garcia de Jalon and Bayo (1994)). Perhaps the most commonly employed schemes are the Runge-Kutta methods, which, like Euler's method, involve recalculating estimates of velocity and position in discrete time steps. In Runge-Kutta, however, the dynamics algorithms are used to produce multiple estimates of accelerations within some fixed time step, and these estimates are combined (averaged) to yield an estimate of velocity and positional change over the fixed time step. The more intermediate acceleration calculations that are performed within a time step, the more accurate (and the more time consuming) are the final estimates of velocity and position. A different approach to the integration problem is taken by the predictor-corrector methods (Garcia de Jalon and Bayo 1994). These methods work by performing iterative cycles of prediction and refinement, given some tolerance threshold. An important property of the predictor-corrector methods is that the error estimates can be used to adjust the size of the time step dynamically to minimize excess calculations.

Which method is most appropriate for the dynamics simulation of legged robots? In the general case of dynamics simulation, the predictor-corrector methods are extremely appealing. In particular, the ability to tune the time step dynamically can yield substantial speedups when simulating systems that enter into periods of differing "stiffness." Differential equations are called stiff if they change rapidly within small time windows (Burden and Faires 1993). In the case of dynamics simulation, differential equations become stiff at points of contact and external force exertion, where accelerations would be expected to change rapidly within a small period of time. In the case of locomotion with soft constraints, however, it is not clear that there would be the opportunity for meaningful adjustment of the time step within a simulation run. This is because such a system would not be expected to undergo dramatic changes in overall stiffness, since it is in a state of constant contact with the environment and actuated joints provide a constant stream of external perturbations (Shih 1986).

There is another important reason why a predictor-corrector method might not be appropriate for locomotor-control testing, even if it were expected to yield more efficient state estimates. Predictor-corrector integration methods work by performing iterative refinement until estimates fall within some tolerance threshold, requiring different numbers of iterations (and therefore calculation times) from step to step. If the simulation is to support online interaction with a user, this shifting of speed is likely to be unacceptable. Even if the simulation does not depend on user interaction, any modification of the dynamics-calculation time steps would have to be synchronized with all other components in the simulation. Both of these considerations are relevant if the simulation is to be used as a controller test bed for legged robots. In addition to being able to support online user interaction, the simulation system would need to coordinate the integration step size with sensor-, actuator-, and controller-model updating. In other words, the dynamics engine cannot simply choose arbitrary integration steps, since sensor, actuator, and controller models can be expected to have competing and less flexible needs regarding the regularity of their updating. Although Runge-Kutta methods would be expected to yield more accurate results, we currently use Euler's method in our simulation system to take advantage of its simplicity and speed.

4. Interfacing with the Simulator

A critical element of any simulator is the ease with which users can interact with it. How easy it is to set up and reuse parts of an experiment and what mechanisms exist for manipulating the simulation online and for analyzing data produced by the simulation are important issues. If the objective is to test control algorithms, there must be a convenient way to incorporate and evaluate such algorithms. If nontraditional robotic mechanisms are to be studied, there must be an easy way to incorporate third-party sensor and actuator models. These are all issues related to interfacing with the simulation system. We believe that interfacing facilities are critical to the design of a useful tool for developing and evaluating robotic control systems, and that existing robotic simulators do not provide convenient interfacing. In this section, we briefly review some common approaches to interfacing, paying special attention to the issue of interfacing control code, and describe the interfacing scheme we have implemented.

4.1. Common Approaches to Interfacing

There are four traditional approaches to interfacing with a simulation. We discuss each in turn.

4.1.1. Graphical Drag-and-Drop

In the graphical drag-and-drop approach, the simulator includes a substantial user-interface component that functions like a computer-aided drafting (CAD) tool, allowing a user to configure physical robots simply by connecting parts on the screen. This enables users to assemble robots and conduct simulations without having to understand anything about the internal simulator code. Commercial software developers sometimes devote considerable resources to the design of friendly and intuitive user interfaces. The commercial product Working Model (Davidson 1996) is a good example of this approach. However, a common problem with this approach is that to make the simulator easy to use by novices, flexibility is sacrificed. It may be difficult or impossible to extend such simulators, and little attention is paid to the interfacing of control code, since these are primarily physical modeling systems for use by nonprogrammers.

4.1.2. Proprietary Language

Although simulation systems that are designed for complex physical modeling often ignore the issue of control code entirely, many simulation systems designed for simple, stereotyped, fixed-configuration robots take the opposite tack by providing a proprietary (sometimes pictorial) control language. This language is typically structured so that certain common functions and actions are easy to express. If one adheres to the control methodology embodied by the proprietary language, such as a behavior-based approach (Konolige 1997; Brooks 1991a), powerful controllers can be developed rapidly. The main disadvantage associated with these proprietary languages is that they often cannot provide the resources and computational power of an established high-level programming language, and it may be impossible to reuse existing control code without first rewriting it (if indeed this is at all possible). If extensive and novel control research is involved, a proprietary language can rapidly become an impediment to doing efficient work.

4.1.3. Code Looping

In the code-looping approach, a basic simulation engine (for example, one performing dynamics modeling) passes control periodically to a user function written in the language of the simulator. Alternatively, a user-written program passes control periodically to the simulation engine. The commercial product SD/FAST (Hollars, Rosenthal, and Sherman 1994) employs this approach. SD/FAST reads a file describing the mechanical configuration of a robotic plant, and produces C code that, after it is compiled, simulates the robot dynamics. In SD/FAST, the user can add control code by writing additional procedures in C, appending these procedures to the code produced by SD/FAST, and then recompiling.

The code-looping approach has the advantage that the control-code programmer is able to utilize the full resources of the parent language, a feature that facilitates the use of existing or third-party control algorithms. However, significant disadvantages are associated with this approach. First, the burden placed on the programmer with regard to understanding the internal structure of the simulator can be prohibitive. Detailed knowledge of internal simulator variables and procedures may be required to send outputs to a specific actuator on a robot or to query a specific state variable. Second, this approach encourages the programmer to design code that is inherently nonreusable. That is, accessing sensors and actuators may require reference to specific variables in the simulator code, tying a specific control algorithm to a specifically configured robot, and vice versa. Third, the code-looping approach puts the burden of managing distributed controllers and simulating sensor and actuator transmission properties squarely on the shoulders of the control-code programmer. As with the code-library approach described below, user-interface components for visualization, on-line manipulation, and data analysis are often missing from these robotic simulators. Hence, third-party programs may be necessary, and routines for online interaction with the simulator may have to be custom written by the user.

4.1.4. Code Library

In the code-library approach, the simulation system itself exists as a set of functions and objects that must be assembled by a programmer to construct a given simulation experiment. Robotica for Mathematica (Nethery 1993) and the Matlab Robotics Toolkit (Corke 1996) are good examples of this approach. A code library may be ideal for building a one-shot custom simulation, but a substantial programming investment will be necessary before development of control code can begin. While some libraries do provide functions for reading symbolic robot assembly files, others do not, and require robots to be built using a series of cumbersome procedure calls. Even when routines are provided for reading robot assembly files, these files are helpful only for the physical configuration of robots and do not address the interfacing of control code or the attachment of sensors and actuators. Like the code-looping approach, the burden of managing distributed controllers and simulating signal-transmission properties falls on the control-code programmer. User-interface components are again left to the programmer assembling the simulation, although in some cases the code library is provided in a language environment such as Matlab or Mathematica that provides significant built-in facilities for visualization and data analysis.

4.2. A New Interfacing Scheme

The interfacing approaches described in the previous section are either designed to be easy to use by nonprogrammers but are not suited for advanced control systems design, or they are designed for hard-core programmers, require a substantial programming investment before control-code development can begin, and encourage the design of nonreusable components. Ideally, one would like some combination of these two approaches, whereby it would be easy for nonprogrammers to assemble reusable components and design custom control experiments, but also easy for control programmers to write and interface sophisticated control algorithms, and would likewise be easy for simulation designers to extend the simulation with new sensor and actuator models.

In the remainder of the paper, we describe one approach to providing such a framework. The key element of this approach is the use of a custom object-oriented configuration language that mediates all interactions between the different components of the simulation system (see Fig. 6). The configuration language facilitates the description of hierarchical structures and provides a uniform mechanism for describing the assembly of, and communication among, simulated robots, graphical user-interface components, sensors and actuators, and control algorithms.

The underlying principles embodied by the configuration language and our interfacing scheme are well-established in the object-oriented programming (OOP) literature, but are not widely employed within the robotic simulation or control-engineering community. This is unfortunate, because these domains are uniquely situated to benefit from an OOP interfacing framework. In the sections that follow, we describe the specific interfacing scheme that we have implemented for our simulation system, and discuss the underlying design principles. It is especially important to note that these design principles are applicable to a wide range of simulation systems, not just our own.

4.2.1. Beyond Object-Oriented Encapsulation

A core principle underlying object-oriented design is encapsulation. Encapsulation dictates that objects be manipulable and observable via a well-defined interface, without regard to their internal implementation. Encapsulation is important because it supports the composition and reuse of independent components. The general principle of encapsulation and object-oriented design can be found in existing robotic libraries. These libraries provide a set of classes written in an object-oriented language (for example, in C++), which can be used by a programmer to build a larger program. However, we have observed that writing programs using robotic library classes is a complicated and delicate process involving substantial programming. Although the basic classes are provided, actually instantiating hierarchical structures and wiring together large numbers of sensors, actuators, controllers, and user-interface components is messy and error prone. Much of the difficulty lies with the underlying language used to implement the components. In looking at the ways in which these components are assembled and manipulated in robotic simulations, it has become clear to us that a domain-specific intermediary language, dedicated to supporting the assembly of and communication between hierarchical components, would solve these difficulties and provide a more robust framework for doing control-systems research. Such an intermediary language would allow programmers to design components (sensors, actuators, controllers, GUI displays, dynamics algorithms) in an established language, in our case C++, while allowing nonprogrammers to assemble and reassemble such

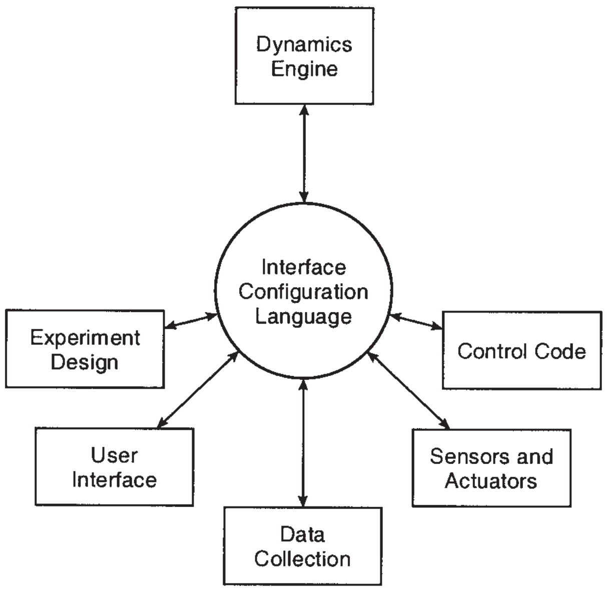

Fig. 6. The relationship between the configuration language and the other components of the simulation system. The decomposition of components makes it easy to extend the simulation system or to replace specific modules. The configuration language makes it easy for the individual modules to have access to arbitrary state information and to communicate with one another.

components as desired in constructing specific experiments. In the jargon of programming languages, such intermediate languages are sometimes called scripting, macro, or gluing languages.

4.2.2. An Interfacing/Configuration Language for Robotic Simulation

Figure 7 shows the description of a complete simulation using the configuration language we have developed. As can be seen, the configuration language provides a uniform way of describing all aspects of an experiment. Configuration files describe the physical parameters of the robots to be simulated; the size, construction, and placement of their limbs; and the number, location, and properties of sensors, actuators, and controllers. Configuration files also describe the physical environment (the ground model and other objects), the arrangement of graphical displays and plots, the data files to read or write, and the kinds of interactions allowed online during the simulation. Typically, a simulation is begun by specifying a configuration file that describes an experiment. This initial configuration file may invoke other configuration files that in turn define specific robots, controllers, and environments.

The configuration language is chiefly used to assemble and wire together different component building blocks. The building blocks used by the configuration language are C++ classes that support an elaborate protocol specialized for the configuration language. It is this protocol that lets the configuration language know how different components can be parameterized, assembled, observed, and controlled by nonprogrammers.

In the configuration language, all objects are described in a similar hierarchical format and all object parameters (such as the length of a segment or the mass of an object) are clearly labeled. Variables may be declared and passed as arguments between configuration files, allowing for the modular reuse of component structures. This allows individual robots or robot components, as well as custom interactive panels and displays, to be stored in their own configuration files and selectively assembled in the design of individual experiments. A benefit of implementing an intermediary-interpreted language on top of the underlying C++ components is that configuration files are interpreted at run time, and do not have to be recompiled after they are modified.

4.2.3. Connecting Components

User-interface toolkits frequently employ sophisticated object-oriented design patterns for supporting interobject communication. This is because they are designed to work with a wide variety of applications, and must support flexible means of interfacing with arbitrary objects. Typically, a user-interface toolkit provides special functions for one object to register with another object so that it receives

```
// Simple two-joint arm with PD controllers at each joint
World SimpleWorld
  {
  gravity=9.81;
  deltat=.001;

  Robot simplebot
    {
    Base.fixed
      {
      location=0,0,0;
      Shape.Cube
        dimensions=1,1,.3;
      }
    Chain leg1
      {
      hip_distance=0,0,0;
      Link link1
        {
        dhlength=0; dhtwist=1.57; dhoffset=.15;
        mass=1;
        Shape.Legcone
          {
          length=1;
          proximal_radius=.15;
          distal_radius=.1;
          }
        }
      Link link2
        {
        dhlength=.1; dhtwist=-1.57; dhoffset=.4;
        mass=1;
        Shape.Legcone
          {
          length=.65;
          proximal_radius=.1;
          distal_radius=.05;
          }
        }
      }
    }
  }

// A camera showing the arm and controls to manipulate
it

Window
  {
  Window.grob videocam
    {
    size=330,330;
    drawer=SimpleWorld;
    }
```

Fig. 7. (continued on next page)

```
  Window
   {
   width=316; height=330;
   Valuator.Slider zoom_factor
     Wire TO videocam.zoom;
   Valuator.Slider distance
     Wire TO videocam.distance;
   Valuator.Slider Euler1_angle
     Wire TO videocam.euler1;
   Valuator.Slider Euler2_angle
     Wire TO videocam.euler2;
   Button.Light pause
     Wire TO SimpleWorld.running;
   Button quit
     Wire TO Simulator.quit;
   }
  }
```

// **Proportional-Derivative controllers attached to arm joints**

```
Controller.PD pd1
  {
  Wire FROM SimpleWorld.simplebot.leg1.link1.angle
        TO current;
  Wire TO SimpleWorld.simplebot.leg1.link1.torque
        FROM output;
  }

Controller.PD pd2
  {
  Wire FROM SimpleWorld.simplebot.leg1.link2.angle
        TO current;
  Wire TO SimpleWorld.simplebot.leg1.link2.torque
        FROM output;
  }
```

// **Sliders to manipulate arm (wired to the PD controllers)**

```
Window controllers
  {
  width=300;
  Window
   {
   label="PD Controller for Joint 1";
   Valuator.Slider goal_angle
     {
     range=-3.2,3.2;
     Wire TO pd1.goal;
     }
   Valuator.Slider pconstant
     {
     range=100,60000;
     Wire TO pd1.pconstant;
     }
   Valuator.Slider dconstant
     {
     range=0,30000;
     Wire TO pd1.dconstant;
     }
   }
  Window
   {
   label="PD Controller for Joint 2";
   Valuator.Slider goal_angle
     {
     range=-3.2,3.2;
     Wire TO pd2.goal;
     }
   Valuator.Slider pconstant
     {
     range=50,30000;
     Wire TO pd2.pconstant;
     }
   Valuator.Slider dconstant
     {
     range=0,5000;
     Wire TO pd2.dconstant;
     }
   }
  }
```

Fig. 7. (continued from previous page)

notification when the target object is modified. In our configuration language, we have taken this idea a step further by implementing a generalized referencing scheme that allows any variable or parameter of an object to be specified as the target or source of a subsequent operation. Most objects in a simulation are named, and state variables are referenced via the standard C++ convention of listing a hierarchical trail of parent objects followed by the name of the parameter to access. For example, to refer to the angle of a specific joint on a multileg robot, we might write something like "simple bot.leftfrontleg.knee.Joint_Angle."

Objects in the simulation exist as black boxes with input and output ports. Objects are connected by routing information between an output port on one object and an input port on another. This provides a single uniform interface to all internal simulation parameters. In addition to the control-code classes, all displays and interactive panels use this scheme to specify their inputs and outputs. For example, a configuration file might specify that the current angle of a specific revolute joint on a specific leg of a specific robot should be saved periodically to a data file, be made available for online manipulation by the user, be plotted graphically on-screen, or be provided as an input variable to a specific controller. The configuration language therefore provides a uniform interfacing mechanism not just between the simulation and the user-written control code, but between the internal simulation state variables and the "outside world" of user interaction and data analysis.

4.2.4. Design of the Graphical User Interface

Unlike most programs, ours does not use a fixed graphical user interface for our simulation system. Instead, the user interface is completely specified by experiment configuration files, with custom displays and interactive panels set up to suit the nature of each experiment. Display windows are used to view simulation state information on-line, and like all other structures in the simulation, are specified using the configuration language. A plotting facility is provided for producing two- and three-dimensional plots of arbitrary state variables. Plots can be set to resize and rescale automatically in a variety of ways so that relevant information is always visible, and multiple plots can be overlaid with one another and other display windows for easy comparisons. Displays for arbitrary text information and raw data are also available. Because the generalized referencing scheme allows access to all state information, the displays can also be used for debugging or for visualizing information about the internal state of a controller. Interaction panels allow the user to manipulate the simulation and exert forces online. Like displays, interaction panels use the generalized referencing scheme to allow any state variable, including internal controller parameters, to be manipulated if desired.

The OpenGL library is used for online solid three-dimensional rendering. OpenGL (Woo, Neider, and Davis 1997) is supported on a variety of computer architectures, and many implementations are freely available. While some existing robotic simulators are capable of producing more realistic and detailed images of robots, there is often a high price to be paid for such images. First, because the images are so detailed, they take substantial time to generate. Frequently, an independent commercial package may be required to perform the graphic rendering, and the results of a simulation can only be visualized after an experiment has been completed, eliminating the possibility of online manipulation. Second, the production of detailed graphical images necessitates the specification of detailed graphical models of robot components (geometries, reflectance properties, etc.). This is precisely the work we are trying to save the control-code programmer from having to do. Hence, we use extremely simple primitives to assemble robots (boxes, cones, spheres, etc.) and do not require the separate specification of graphical models.

In addition to providing on-line visualization, we have also implemented procedures that allow movies to be recorded and played back using a custom format that preserves all internal simulation data rather than simply saving graphic images of the simulation. This allows the user to change camera views during playback and to examine at any instant arbitrary state information such as disturbance forces. State information can also be saved in numerical form at designated intervals in Matlab or Mathematica formats, in files that are automatically annotated to include information about variable types, ranges, and units. Data may also be read from a file to provide a predetermined sequence of perturbations; for example, to provide a repeatable application of forces.

4.2.5. Control-Code Interfacing

To provide a testbed for researchers interested in complex, adaptive, nontraditional controllers, and to provide the greatest flexibility for control programmers, a simulation system should allow control code to be written in an established object-oriented programming language. Although the interfacing of control code shares much in common with the interfacing of graphical user interface and sensor/actuator models, in the following sections we describe in more detail the process by which control algorithms (written in C++) interface with the configuration language and are employed in the design of control experiments. We also introduce a mechanism for varying performance characteristics and simulated transmission properties to study the sensitivity of controllers to real-world constraints. The basic scheme can be seen as consisting of two tiers (see Fig. 8).

4.2.6. Controller Stubs

The first tier consists of controller stubs. Controller stubs are attached to robots from within configuration files, and specify how external C++ control-code classes interface with robot sensors and actuators. They are part of the configuration-language syntax, and provide the bridge between the simulation system and the external control-code algorithms.

A controller stub calls for the instantiation and attachment of a controller object. It specifies the name of an external, user-written C++ controller class, the inputs and outputs (sensors and actuators) the controller has access to, and the transmission properties to be simulated as information passes back and forth between the simulator and the controller. Sensor and actuator transmission delays, sampling rates, and noise can all be specified within controller stubs.

In keeping with the black-box formalism, the controller stub does not require any information about how the actual controller code functions—it simply specifies the input and output variables. It is the separation of external controller code and controller stubs that makes possible the modular reuse and parameterization of control algorithms. As configuration files are parsed and controller stubs encountered,

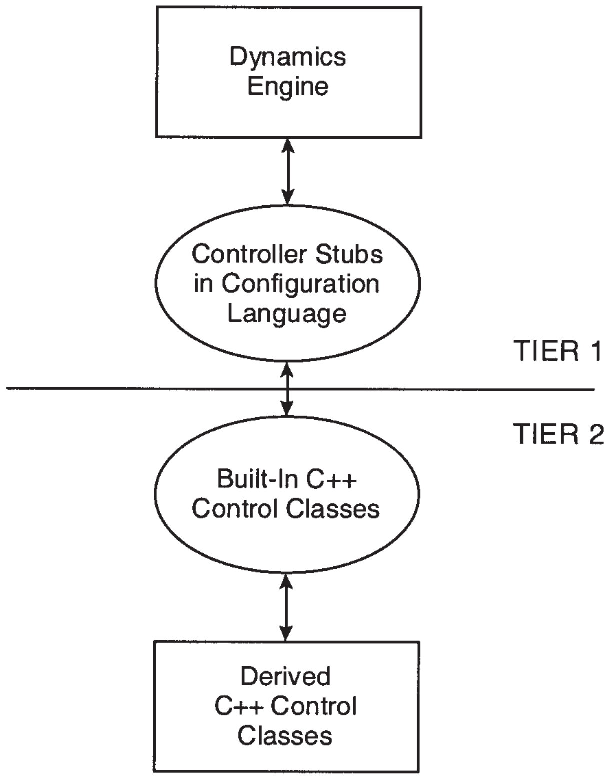
Dynamics Engine
Controller Stubs in Configuration Language
TIER 1
TIER 2
Built-In C++ Control Classes
Derived C++ Control Classes

Fig. 8. The two-tier framework for interfacing user-written controllers. In the first tier, controller stubs, written in the configuration language, are used to attach controllers to robots in the dynamics engine, specifying their inputs, outputs, sampling rates, and transmission properties (top). In the second tier, user-written controllers are implemented as C++ classes that are derived from built-in parent classes (bottom), and the parent classes handle all communication and synchronization with the simulation system configuration language (middle).

the configuration-language interpreter locates the user-written C++ classes and instantiates copies of the controllers. Each instantiation of a controller is a distinct object with a unique name and its own independent internal state variables.

The use of controller stubs allows some simple but useful manipulations of controller input and output data as the data pass between the simulator and the control algorithms. Both input and output variables can have associated sampling rates and transmission delays that instruct the simulator to buffer and pipeline signals to simulate more realistically the properties of a physical system. A simple noise model can also be applied to both controller inputs and outputs. The simulation of sampling rates, transmission delays, and noise is handled automatically by the simulator, and does not require any special design on the part of the control-code programmer, meaning that transmission properties can be specified for any controller class without modification or recompilation of the control code itself. By varying simulated transmission properties, the control-systems engineer can explore more difficult and realistic control tasks. Because the transmission properties are adjustable, real-world constraints can be introduced in a controlled fashion.

4.2.7. C++ Controller Classes

Extensible C++ classes constitute the second tier of the control-code interfacing scheme. In C++, a class is defined as having certain variables and implementing certain procedures or functions; classes may be derived from other classes, inheriting the variables and functions of the parent class and possibly adding new variables and implementing new functions. This basic principle of inheritance is used to provide the control-code programmer with the building blocks for creating new controllers.

The simulator code includes a set of extensible C++ classes that handle communication and coordination between the simulation and user-written control algorithms. These extendable parent classes provide the foundation for all control-code development, and ensure that all control algorithms are well behaved and present a uniform interface to our simulation engine (and to any other related simulation engine). The built-in parent classes provide the C++ functions and objects that manage communication with the simulator, and handle simulated transmission properties. New control classes are derived from these built-in parent classes, thereby inheriting the supervisory functions of these classes and freeing the control-code developer from having to worry about the internal structure of the simulator. Rather than being tied to specific sensors and actuators on a specific robot, controller classes exist as encapsulated black boxes, specifying only the kinds of inputs and outputs with which they expect to connect. Thus, they can be plugged into robots and parameterized as needed for a given experiment.

Controller classes are designed to self-register with the simulator automatically so that no modification of the simulator code is necessary. The automatic self-registration of user code is accomplished via an OOP design pattern known as a "factory" (Gamma et al. 1995), which keeps track of all user classes and is used to create new instances of the controllers as needed. When a controller stub specifies the attachment of a controller to a robot, the factory is asked to locate the user class and create a new controller. Self-registration allows users to add controllers without having to modify and recompile the entire simulator every time (Beveridge 1998), and it allows users to share controllers and custom modules.

It is important to remember that actual controller code is not contained within configuration files, but is written in standard C++ and placed in separate class files that may be compiled independently. When the simulation starts, all user-written controller classes are automatically registered with the simulator. As configuration files are parsed, controller stubs specify which user-written controller classes should be attached to which robots and where. The simulator locates the user-written control classes, instantiates new copies of the controllers, and binds the input and output variables. While the simulation runs, input and output variables are automatically updated by the simulator as specified by the controller-stub parameters (incorporating sampling rates, delays, and noise). All of this is done behind the scenes and without any modification of the actual control code.

A benefit of the two-tier approach is that once a controller has been written, it can be tested on arbitrarily configured robots and with widely varying operating parameters without any modification of the control code itself. Controllers can be instantiated on the same or different robots (for example, by placing independent copies of a simple joint controller on multiple joints of a robot). The two-tier approach also makes it easy to interface control algorithms that have been written for another application and to study the effects of signal-transmission properties on such algorithms. Because the controller classes exist as encapsulated black-box structures, it becomes possible for nonprogrammers to assemble and evaluate robot-control schemes using third-party controller classes in a plug-and-play fashion. Furthermore, since configuration files do not need to be compiled, the turnaround time for modifying controller-stub parameters and rerunning an experiment is short.

A key goal in the design of the simulation system was to support the use of third-party sensor and actuator models. The same mechanisms used to design black-box controllers are used to "wrap" third-party sensor and actuator models so that they can be employed as building blocks in the simulation system. Essentially, the strategy we have followed has been to require some extra work on the part of the component programmer, in ensuring that all code presents a uniform interface to the simulation, so that the job of experiment designer is made easier.

5. Discussion

Robotics researchers often argue that robotic simulation is a poor substitute for hardware implementation, because simulation inevitably fails to capture the harsh reality of the real world (Brooks 1991b). While we readily acknowledge that simulation involves making simplifying assumptions about the world, we take issue with the underlying premise that the role of robotic simulation must be to provide a testing environment that is as "unforgiving" as the real world. In our opinion, this is an overly restrictive view. In large-scale controller design, we suggest that there is a great need for tools that permit a gentler development environment.

Ideally, a simulation should allow one to investigate, before actually building the hardware, the limitations of control algorithms under different assumptions about processor speed, I/O performance, transmission-line properties, and so forth. And while real-time constraints play an important role in the final implementation of control code, there are substantial benefits to a more flexible analysis of performance characteristics, especially during the development phase, when researchers have a justifiable reluctance to commit to a specific implementation of a controller in hardware. Circumventing the hard constraint of real-time performance is especially important for researchers investigating control algorithms for walking robots, algorithms that are frequently intended to run on custom-distributed processors but are often first prototyped using off-the-shelf PCs.

For example, in designing a controller to recover from missteps, an engineer might like to be able to test the basic algorithm without worrying about execution speed, and then, given a candidate algorithm, explore implementation details like how fast and accurately the algorithm would have to execute to be effective. Simulation can address issues concerning how well a control algorithm will perform if the transmission delay of a specific sensor is doubled, if its sampling rate is halved, or if a noiseless transmission line is not practical. These are questions that need to be answered when building a real robot, but that because of hardware's fixed characteristics, are paradoxically the most difficult to answer without the use of a simulator.

Simulation can also be an invaluable tool for investigating nontraditional sensors, actuators, and robots that cannot economically be built or acquired, for the automated training of adaptable control algorithms, for performing tests that might damage a physical robot or injure a human operator, and for testing controllers that have been explicitly designed to work with robots of varying configurations.

5.1. Interfacing with a Simulator

Considering that simulation has such a great potential to facilitate the development of robotic control algorithms, it is surprising that most existing robotic simulation systems and dynamics simulators are so ill-suited for large-scale control-code development.

Many simulators provide no facility whatsoever for interfacing control code, and assume that the robots being simulated will employ traditional, well-established, hardware-based controllers, so that no software testing of control algorithms is necessary. For other robotic simulators, it is possible to interface control code, but the control engineer must possess an intimate knowledge of the internal simulator code to access specific state information. Control code becomes wedded to a specific robot model, and it can be difficult to reuse control algorithms or robot designs. For projects where the interfacing of control code is viewed as a one-time custom-programming job and where a dedicated control architecture will be incorporated into the actual dynamics-simulation code, this is an acceptable approach.

Still other robotic simulators include a proprietary control language, but because such languages have limited general appeal, they tend to be found in simulators for mechanically simple, fixed-configuration robots, such as commercial wheeled robots (for example, see the work of Mach and Albrecht (1992); although Liu and Qin (1997) and Strippgen, Peters, and Milde (1998) described sophisticated dynamics simulators with dedicated control languages). While such languages can be extremely useful in demonstrating the validity of a specific control approach (Brooks 1986), they tend to have limited appeal and are not suited for advanced control engineering. There do exist a number of powerful, flexible, stand-alone robot-control languages that exist independently of any physical robot or mechanical simulation. These languages often provide sophisticated algorithms for path planning, trajectory control, or other high-level robotic control tasks (Pelich and Wahl 1997), and could theoretically be incorporated into sophisticated physical modeling systems. However, designers of complex physical modeling simulators are reluctant to commit themselves to a specific robotic control language having a limited audience, so they tend to abandon the control-code interfacing issue entirely, leaving it instead as a project for the end user.

We have presented in this paper a simulation tool specifically designed for the development of control code for legged robots, concentrating in part on our novel scheme for interfacing with the simulation. As researchers interested in applying theories of biological motor control to robotics (and vice versa), we were particularly keen to reduce what we perceive as the prohibitive overhead incurred when interfacing with existing robot simulators. The interfacing scheme we have described could be added to virtually any simulation engine, yielding a substantial increase in utility with very little expenditure of effort. A standardized configuration language, whether or not bearing any resemblance to our interfacing scheme, along with self-registering sensor, actuator, and controller modules, would be useful because it would allow components of a robotic control system to be shared between researchers and different simulation engines in a plug-and-play fashion.

There are three particular aspects of the interfacing scheme that we believe may be of interest to robotics researchers and robotic simulation designers. First, we have described a high-level configuration language that acts as a bridge, or layer of abstraction, between simulation components. It facilitates the efficient description of robots, their actuators, sensors, and controllers, and provides a unified framework for control-code interfacing, user-interface design, data analysis, and extensibility. A generalized referencing scheme allows control structures and user-interface elements to communicate flexibly with one another and to access arbitrary state information. Second, we have described a simple mechanism that facilitates the integration of modular, distributed control code written in a popular object-oriented language, C++. Third, we have shown how controller stubs facilitate the simulation of signal properties such as sampling rates, transmission delays, and signal noise.

5.2. Simulation Trade-Offs

It should be remembered that in all simulation systems, difficult choices must be made regarding the detail at which to model underlying phenomena; trade-offs between accuracy and speed must be balanced. We have presented one approach to dynamics simulation, geared to supporting control-code development for legged walking robots. While the core articulated-body-dynamics algorithms employed by the simulation system are well founded, the contact-force routines that we currently use are penalty-based methods and only provide heuristic approximations of physical contact. We therefore consider the dynamics routines to be useful in producing qualitative models rather than quantitative ones. The same can be said about our current sensor and actuator models. Qualitative simulation is not appropriate in all situations. In some projects, a simulation may need to be meticulously constructed to reproduce an existing (or soon to exist) plant. In such cases, it is necessary that the model reproduce the physical plant as closely as possible, so that precise measurements and performance parameters can be gleaned from the simulation (Ma et al. 1997). Our simulation system is not appropriate for such work.

Fortunately, the very nature of research on the control of legged robots makes it amenable to efficient, qualitative dynamic simulation. There are two main reasons for this. First, contact resolution, which is by far the most complicated, time-consuming, and error-prone aspect of dynamic simulation, is typically quite restricted and stereotyped in locomotion. Second, unlike the situation for devices meant to operate within narrow tolerances in a well-defined workspace, walking robots are intended to be tolerant to variations in their environment and their internal configuration; because robustness and adaptability are central elements in the design of these control schemes, the importance of *quantitatively* reproducing a specific environment is overshadowed by the importance of being able to easily test control algorithms with a variety of robot configurations and under a variety of qualitatively different environmental conditions.

The simulation system described in this paper is freely available on the Internet for noncommercial uses at http://www.life.uiuc.edu/delcomyn.

Acknowledgments

This research has been supported in part by the Research Board of the University of Illinois and NSF grant BAC 92-16562.

References

Armstrong, W., and Green, M. W. 1985. The dynamics of articulated rigid bodies for purposes of animation. *Visual Comp.* 1(4):231–240.

Baca, A. 1998. Application of computer animation techniques for presenting biomechanical research results. *Comp. Biol. Med.* 28(4):449–454.

Baraff, D. 1992. Dynamic simulation of nonpenetrating rigid bodies. PhD thesis, Cornell University.

Beveridge, J. 1998. Self-registering objects in C++. *Dr. Dobb's J.* 288:38–41.

Brandl, H., Johanni, R., and Otter, M. 1986 (Vienna, Austria). A very efficient algorithm for the simulation of robots and similar multibody systems without inversion of the mass matrix. *Proc. of the IFAC/IFIP/IMACS Intl. Symp. on the Theory of Robots.*

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE J. Robot. Automat.* 2(1):14–23.

Brooks, R. A. 1991a. Intelligence without representation. *Art. Intell.* 471:139–159.

Brooks, R. A. 1991b. New approaches in cybernetics. *Science* 253(Sept.):1227–1232.

Burden, R. L., and Faires, J. D. 1993. *Numerical Analysis*, 5th ed. Boston, MA: PWS.

Corke, P. 1996. A robotics toolbox for Matlab. *IEEE Robot. Automat. Mag.* 3(1):24–32.

Davidson, H. 1996. Working model 3.0 and automation. *Design News* 52(6):172.

Elmqvist, H., Mattson, S. E., and Otter, M. 1998 (Manchester). Simulation using Modelica. *Proc. of the 12th European Simulation Multiconference*, pp. 127–131.

Featherstone, R. 1987. *Robot Dynamics Algorithms.* Norwell, MA: Kluwer Academic.

Fisette, P., Peterkenne, J. M., and Smain, J. C. 1998 (Manchester). MBSOFT: A symbolic/numerical multibody program for analyzing mechanical and mechatronic systems. *Proc. of the 12th European Simulation Multiconference*, pp. 566–570.

Freeman, P. 1989. Decoupled tree-structure approach to efficient dynamic simulation of a quadruped robotic vehicle. Master's thesis, Ohio State University.

Freeman, P., and Orin, D. 1991. Efficient dynamic simulation of a quadruped using a decoupled tree-structure approach. *Intl. J. Robot. Res.* 10(6):619–626.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley.

Garcia de Jalon, J., and Bayo, E. 1994. *Kinematic and Dynamic Simulation of Multibody Systems: The Real-Time Challenge*. New York: Springer-Verlag.

Hollars, M. G., Rosenthal, D. E., and Sherman, M. A. 1994. *SD/FAST User's Manual*. Symbolic Dynamics, Inc., Mountain View, CA.

Konolige, K. 1997 (Freiburg, Germany). COLBERT: A language for reactive control in Sapphira. *Proc. of the 21st German Conf. on Art. Intell.*, pp. 31–52.

Lilly, K. 1993. *Efficient Dynamic Simulation of Robotic Mechanisms.* Norwell, MA: Kluwer Academic.

Liu, J., and Qin, H. 1997. C4, a software environment for modeling self-organizing behaviors of autonomous robots and groups. *Robotica* 15:85–98.

Ma, C., Buhariwala, K., Roger, N., MacLean, J., and Carr, R. 1997. MDSF—a generic development and simulation facility for flexible, complex, robotic systems. *Robotica* 15:49–62.

Mach, R., and Albrecht, R. W. 1992. A mobile robot programming language directed by an expert system. *Robot. Manufact.* 5:170–173.

Manko, D. J. 1992. *A General Model of Legged Locomotion on Natural Terrain.* Boston, MA: Kluwer Academic.

Marhefka, D. W., and Orin, D. E. 1996. XAnimate: An educational tool for robot graphical simulation. *IEEE Robot. Automat. Mag.* 3(2):6–14.

McKenna, M., and Zeltzer, D. 1990. Dynamic simulation of autonomous legged locomotion. *Comp. Graphics* 24:29–38.

McMillan, S., Orin, D., and McGhee, R. 1995. Efficient dynamic simulation of an underwater vehicle with a robotic manipulator. *IEEE Trans. Sys. Man Cybernet.* 25(8):1194–1206.

McMillan, S., Orin, D., and McGhee, R. 1996. A computational framework for simulation of underwater robotic vehicle systems. *Autonomous Robots* 3:253–268.

Mirtich, B. 1996. Impulse-based dynamic simulation of rigid-body systems. PhD thesis, University of California at Berkeley.

Multon, F., Cani-Gascuel, M., and Debunne, G. 1999. Computer animation of walking: A survey. *J. Visualization Comp. Animation* 10(1):39–54.

Nethery, J. 1993. Robotica: A structured environment for computer-aided design and analysis of robots. PhD thesis, University of Illinois.

Pelich, C., and Wahl, F. M. 1997. ZERO++: An OOP environment for multiprocessor robot control. *Intl. J. Robot. Automat.* 12(2):49–57.

Reichler, J., and Delcomyn, F. 1998 (Manchester). A simulation testbed for biologically inspired robots and their controllers. *Proc. of the 12th European Simulation Multiconference*, pp. 437–442.

Shih, L. 1986. Dynamic modeling and simulation of mechanisms consisting of combined closed and open kinematic chains with compliance. PhD thesis, University of Wisconsin-Madison.

Shih, L., Frank, A., and Ravani, B. 1987. Dynamic simulation of legged machines using a compliant joint model. *Intl. J. Robot. Res.* 6(4):33–46.

Strippgen, S., Peters, K., and Milde, J. 1998 (Manchester). Situated communication with a simulated robot. *Proc. of the 12th European Simulation Multiconference*, pp. 448–452.

Walker, M. W., and Orin, D. E. 1982. Efficient dynamic computer simulation of robotic mechanisms. *J. Dyn. Systems Meas. Control* 104:205–211.

Woo, M., Neider, J., and Davis, T. 1997. *OpenGL Programming Guide*, 2nd ed. Reading, MA: Addison-Wesley.

Zeghloul, S., Blanchard, B., and Ayrault, M. 1997. SMAR: A robot modeling and simulation system. *Robotica* 15(1):63–73.