Automated Synthesis of Numerical Programs for Simulation of Rigid Mechanical Systems in Physics-Based Animation

Thomas Ellman ellman@cs.vassar.edu
Ryan Deak rydeak@vassar.edu
Jason Fotinatos jafotinatos@vassar.edu

Department of Computer Science
Vassar College
Poughkeepsie, NY 12601

Abstract

Physics-based animation programs are important in a variety of contexts, including science, engineering, education and entertainment among others. Manual construction of such programs is expensive, time-consuming and prone to error. We have developed a system for automatically synthesizing physics-based animation programs for a significant class of problems: constrained systems of rigid bodies, subject to driving and dissipative forces, under the control of an interactive user. Our system includes a graphical interface for specifying a physical scenario, including objects, geometry and coordinate systems, along with a symbolic interface for specifying dynamical variables, forces and constraints operating in the scenario. The entities defined in the graphical interface serve as the underlying vocabulary for specifications entered in the symbolic interface. Our system partitions the constraints and dynamical variables into classes and assigns each class to be implemented in a different component of a general simulation program scheme. It generates a numerical C^{++} simulation program that drives a real-time animation of the specified scenario. Our system is implemented as a collection of rewrite rules in the Mathematica programming language. Our approach provides some of the benefits of formal deductive program synthesis, while keeping the computational costs of program synthesis more in line with conventional program generator technology. We have successfully tested our system on numerous examples.

Keywords: Specification, Synthesis, Numerical, Program.

1. Introduction

Physics-based animation programs are useful in a variety of contexts, including science, engineering, education and entertainment, among others. For example, in science, physics-based animation programs are used to investigate the behavior of dynamical systems. In engineering, they are used to help design vehicles, machinery and other mechanical devices. In education, they are used to teach basic principles of physics. Finally, in entertainment, physics-based animation programs are used in video games involving cars, planes, spaceships and other moving objects. Such programs are usually constructed by hand, in conventional programming languages, such as C^{++}, possibly augmented with a physics-based animation engine toolkit. Unfortunately, manual construction of physics-based animation programs is expensive, time-consuming and highly prone to error.

Our research is aimed at dealing with this problem by applying and extending techniques of Automated Software Engineering. We have developed a system for automatically synthesizing physics-based animation programs for a significant class of problems: constrained systems of rigid bodies, subject to driving and dissipative forces, under the control of an interactive user. This class includes a wide variety of physical systems, such as vehicles (e.g., cars, bicycles, sleds and planes) and articulated structures (e.g. robots and the human skeleton), among others. Our system includes a graphical interface (implemented in the MaxScript language of 3D Studio Max®) for specifying a physical scenario, including objects, geometry and coordinate systems, along with a symbolic interface (implemented in the Mathematica® programming language) for specifying dynamical variables, forces and constraints operating in the scenario. The entities defined in the graphical interface serve as the underlying vocabulary for specifications constructed in the symbolic interface. (See Figure 1.) We use a rewrite system to synthesize rigid-body animation programs. The system includes a knowledge base of geometry, kinematics, dynamics and numerical methods. It operates by partitioning problem constraints and dynamical variables into classes and assigning each class to be implemented in a different component of a general simulation program scheme. The system automatically generates a C^{++} program that simulates the behavior of the system as it interacts with a user, and supplies time-dependent parameters to a rendering engine that generates the animation in real time.

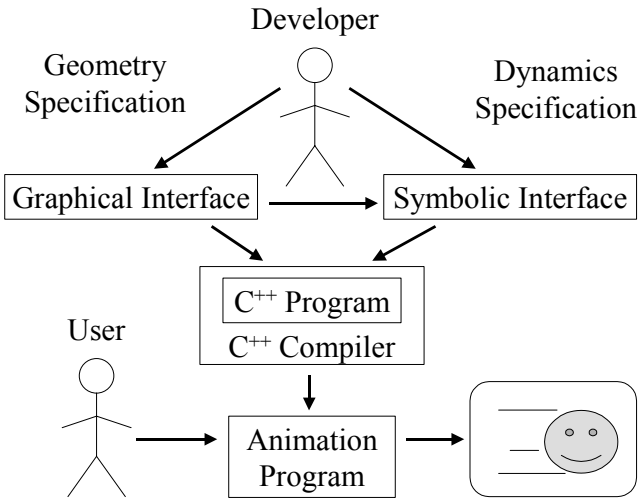

Figure 1. System Architecture

Our research may be seen as an effort to address a fundamental issue in the field of Automated Software Engineering. *How can a program synthesis system provide some of the benefits of formal deductive synthesis, while limiting the computational cost of program generation?* Deductive synthesis uses a theorem-prover to automatically generate a program satisfying a given specification [Manna and Waldinger, 1992]. Soundness of the prover guarantees correctness of the synthesized program. Completeness of the prover makes the method applicable, in principle, to a wide variety of programming problems. Specifications are expressed in logic and therefore have a declarative interpretation in terms of objects and relationships of the application domain. The theorem-prover draws upon a theory of the application domain expressed as a set of logical axioms. The declarative nature of the domain theory may promote transparency and maintainability of the program synthesis system. Unfortunately, theorem proving is a computationally expensive process. For this reason, deductive synthesis is not widely used in software engineering.

Our approach is to sacrifice generality and the formal guarantee of correctness in order to obtain a computationally feasible synthesis process. We focus on a class of problems that is broad enough to include a variety of interesting physical systems, and complex enough to present a challenging synthesis problem, yet narrow enough to allow the development and application of powerful, specialized synthesis techniques. Our system provides some of the features of a formal deductive method: The synthesis process is completely automatic. Program specifications have a declarative interpretation as statements about objects and relationships in the scenario being modeled. Rules of the system knowledge base have a declarative interpretation as general laws of physics and mathematics. We support these claims with examples and experimental results in this paper. We also conjecture that our declarative implementation promotes

transparency and maintainability of the program synthesis system; however, we have not run experiments to directly test these properties. Our approach does not provide the formal guarantee of correctness that comes with using a deductive method. On the other hand, the computational cost of program synthesis is considerably lower and more in line with conventional program generator technology.

This paper is an extended and expanded version of [Ellman et al., 2002]. In Section 2, we review the physics of rigid-body systems. A reader without a background in physics should read this section only to obtain a rough idea of the kinds of phenomena we handle in our system. In Section 3, we describe our graphical and symbolic languages for specifying rigid-body systems. In Section 4, we describe the overall architecture of rigid-body simulation programs. In Section 5, we describe our method of synthesizing programs that instantiate the architecture and satisfy the graphical and symbolic specifications. In Section 6, we describe the results we have obtained from experimenting with our program synthesis system. In Section 7, we discuss related research. In Section 8, we discuss our plans for future work. Finally, in Section 9, we summarize our contributions to the field of Automated Software Engineering.

2. Dynamics of Rigid-Body Systems

Systems of rigid bodies are the subject of a branch of physics known as "Analytical Dynamics" [Baruh, 1999]. In the formalism of Analytical Dynamics, constrained systems of rigid bodies are governed by the Euler-Lagrange equations. (See Figure 2.) These differential equations describe the changes that occur over time in the values of dynamical variables representing the state of a system of rigid bodies. The equations include expressions involving conservative forces (derived from a potential function, e.g., gravity), nonconservative forces (not derived from a potential function, e.g., frictional drag and engine thrust) as well as forces derived from constraints. Two types of constraints appear in the equations: Holonomic constraints depend only on the values of dynamical variables, but not on their derivatives. Nonholonomic constraints depend on both the values and the time derivatives of dynamical variables. Each (holonomic or nonholonomic) constraint is associated with a Lagrange multiplier (λ or μ) representing the force that maintains the constraint. The Euler-Lagrange equations may be instantiated in the context of a given rigid-body system by specifying the following things: a set of dynamical variables; the Lagrangian function (kinetic energy minus potential energy); holonomic constraints; nonholonomic constraints and nonconservative forces.

Lagrangian:	$L = T(\mathbf{q}, \dot{\mathbf{q}}) - P(\mathbf{q}, t)$
Kinetic Energy:	$T(\mathbf{q}, \dot{\mathbf{q}})$
Potential Energy:	$P(\mathbf{q}, t)$
Dynamical Variables:	$\mathbf{q} = (q_1, \ldots, q_n)^{\mathrm{T}}$
Holonomic Constraints:	$\mathbf{H}(\mathbf{q}, t) = \mathbf{0}$
Nonholonomic Constraints:	$\mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}, t) = \mathbf{0}$
Nonconservative Forces:	$\mathbf{F}(\mathbf{q}, \dot{\mathbf{q}}, t)$
Euler-Lagrange Equations:	$(\mathrm{i} = 1 \ldots \mathrm{n})$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} + \sum_{j=1}^{s} \lambda_j \frac{\partial H_j}{\partial q_i} + \sum_{k=1}^{t} \mu_k \frac{\partial N_j}{\partial \dot{q}_i} = F_i$$

Figure 2. Analytical Dynamics

Consider a three-car roller coaster moving on a circular track with hills and valleys. (See Figures 3 and 4.) The state of each car may be described by the following dynamical variables: the angle of revolution of the car around the center of the track; the altitude of the car above the ground; the pitch angle of the car, which varies as the car goes up and down hills; the angles of rotation of the front and rear axles; and the angles of rotation of the joint at the rear of the car, which control the orientation of the link attaching it to the following car. The (gravitational) potential energy of the car is a simple linear function of height. Treating each car as a point mass, the kinetic energy of each is $(1/2)mv^2$ where m is the mass of the car, and v is the linear speed of the car in the direction tangent to the car's location on the track. The nonconservative forces include a friction component, which causes a car to lose energy, and a thrust component (under user control), which causes a car to accelerate forward or backward. Each car's motion is governed by several constraints. A (holonomic) constraint asserts that the car's altitude varies as the track goes up and down to keep the car's center of gravity just above the track surface. Another (holonomic) constraint asserts that the car's pitch angle varies to keep the wheels in contact with the surface of the track. Several (nonholonomic) constraints assert that the axles rotate so that the relative motion between each tire and the track is zero at the point of contact, i.e., the car does not skid along the surface of the track. Additional (holonomic) constraints assert that each car is a fixed distance from the one in front or back of it. A final set of constraints assert that the link behind each car, except the last, is properly aimed at the joint at the front of the car behind it, so the links appear to connect the cars to each other.

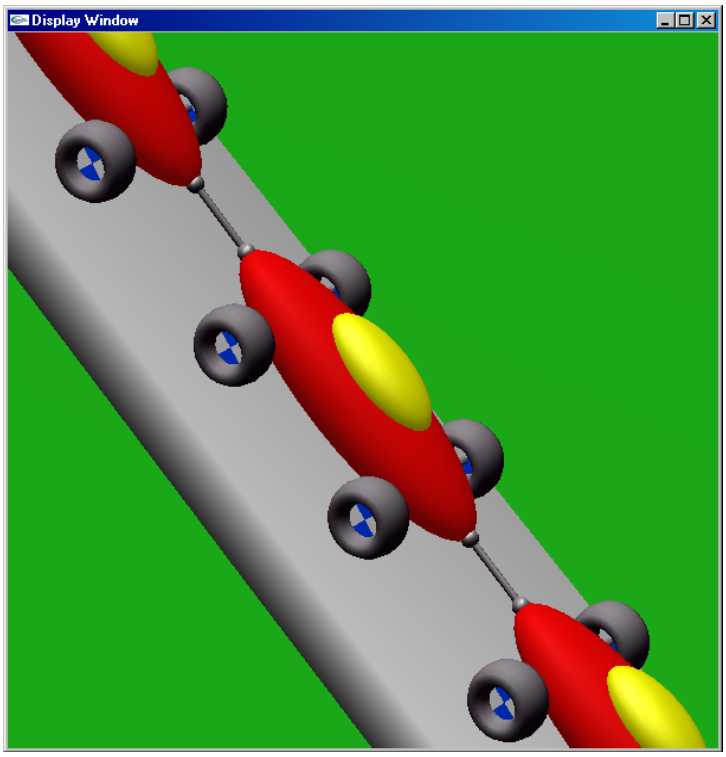

Figure 3. Roller-Coaster Cars

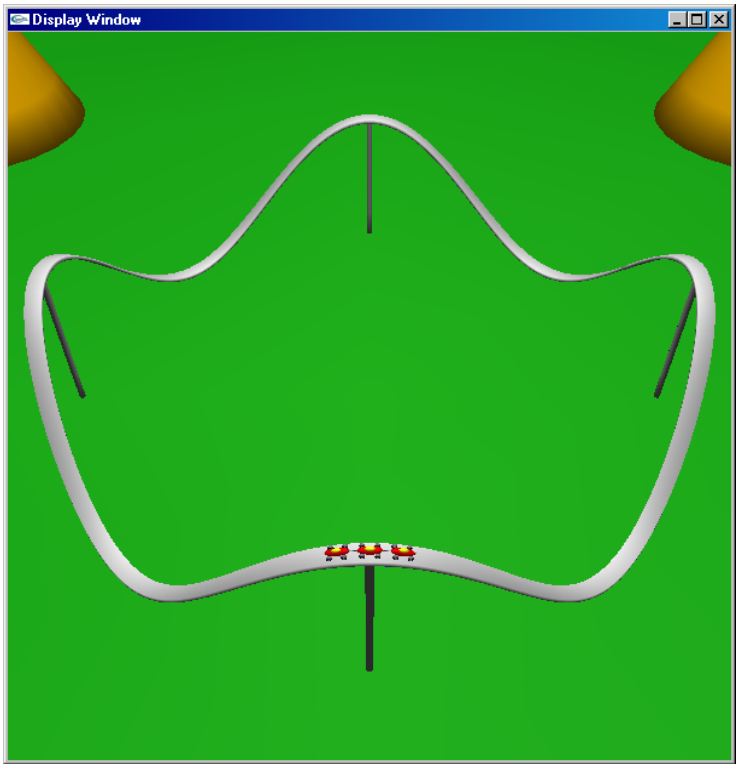

Figure 4. Roller-Coaster Track

Physics-based animation differs from other applications of Analytical Dynamics, such as engineering design. The difference lies in the fact that animation often includes *noncausal* objects, i.e., objects that contribute to visual realism, but which play no role in the dynamics of the remaining part of the system. A dynamical variable is considered "noncausal" if it does not appear in either the Lagrangian or the nonconservative forces. The behavior of the noncausal variables is determined entirely by the constraints in which they participate. In the roller-coaster example, the dynamical variables controlling the cars' pitch, the axles' rotation and the link joints' rotation, are noncausal.

3. Specification of Simulation Programs

3.1 Graphical Specification

In our system, a developer specifies an animation program through a combination of graphical and symbolic interfaces. (See Figure 1.) The graphical interface is implemented in MaxScript, the language of 3D Studio Max. The developer typically begins by defining a tree-structured hierarchy of coordinate systems. A coordinate system hierarchy for the roller-coaster animation is shown in Figure 5. The *Root* of the hierarchy is the fixed, global coordinate system for the entire scene. The *CarOriginI* (I=1,2,3) coordinate systems are children of the *Root* system. Each of these is used to describe the revolution of one of the cars around the center of the track, relative to the *Root* system. Each *CarI* coordinate system specifies the location of a car's center of mass, relative to the corresponding *CarOriginI* system. Each is defined as a child of the corresponding *CarOriginI* system. The *FrontAxleI* and *RearAxleI* coordinate systems locate the front and rear axles of each car. Each of these is a child of the corresponding *CarI* system. In addition, the *FrontContactI* and *RearContactI* coordinate systems specify points at which front and rear tires of each car contact the track. Each of these is also a child of the corresponding *CarI* system. Finally, each car has *FrontJointI* and *RearJointI* coordinate systems, the origins of which mark the points at which it may be linked to the cars in front or back of it. After defining a hierarchy of coordinate systems, the developer typically proceeds to define the visual aspects of the animation, including the geometry and light-reflecting properties of visible objects, as well as the locations of lights and cameras. The visible objects, lights and cameras are attached to the leaves of the coordinate system hierarchy.

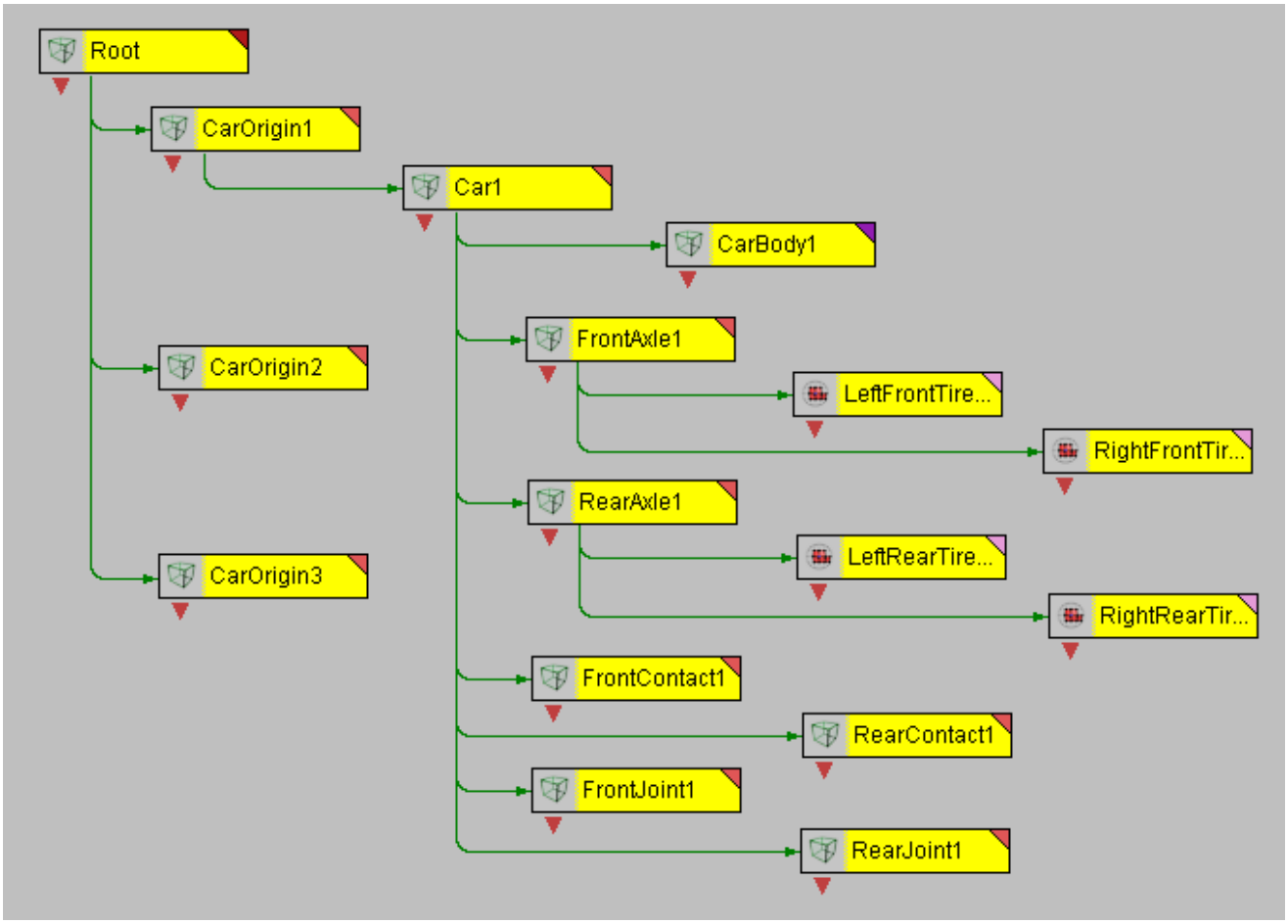

Figure 5. Roller-Coaster Coordinate System Hierarchy

The coordinate system hierarchy defines a vocabulary in terms of which the developer specifies the dynamical behavior of the system to be animated. Each coordinate system is defined in relation to its parent by a translation, a rotation and a scaling operation. The parameters of these transformations (translation vectors (Tx,Ty,Tz), rotation quaternions (Rw,Rx,Ry,Rz) and scaling factors (Sx,Sy,Sz)) may be dynamical variables in the dynamical system being specified by the developer. These transformation parameters may therefore appear in the developer's specification of the Lagrangian, nonconservative forces and constraints governing the behavior of the system. Furthermore, the developer may specify the system's constraints in terms of any of the coordinate systems in the hierarchy.

The coordinate system hierarchy also defines the manner in which dynamical variables influence the visual aspects of the scenario. In the final animation program, some of the parent-child transformation parameters will change over time, in response to numerical simulation of the dynamical system. As a result of these changes in

parameters, the coordinate systems may change their positions, orientations and/or scale measures. Since the visible objects, lights and cameras are attached to the leaves of the hierarchy, the positions, orientations and sizes of these objects may change as well, resulting in an animation of the physical scenario.

3.2 Symbolic Specification

The symbolic interface is implemented in the Mathematica programming language. It provides a declarative language in which the developer specifies the dynamics of the system to be animated. A symbolic specification of the roller-coaster system is shown in Figure 6. The dynamical variables appearing in the specification are illustrated in Figure 7. Selected primitives of the specification language are described in Figure 8. The example specification is divided into several sections: *Initialization, Dynamical Variables, Dynamics, Constraints* and *Definitions*.[1]

In the *Initialization* section, the developer specifies the input parameters to be accepted by the simulation program. These parameters allow the animation to be run multiple times, with different initial states, or different values of forces, masses or other numerical constants appearing in the specification. In the roller-coaster example, there are three input parameters: *InitialOmega* represents the initial velocity of the first car (as an angular velocity). *ThrustConstant* controls the sensitivity of the thrust on the first car to changes in the state of the *ControlUpDown* input device. *DragConstant* determines the relation between the drag on the first car and the car's velocity. This section also includes *InitializationConstraints*, i.e., equations that relate the input parameters to the initial values of the dynamical variables of the system, thereby giving semantics to the input parameters. In the roller-coaster specification, there is one initialization constraint. This constraint relates *InitialOmega* to the z-axis component of the angular velocity of the *CarOrigin1* coordinate system, taken at time zero.

---

[1] The specification uses the following Mathematica notation: The operator *[ ]* indicates function application. A vector is represented as *{x, y, z}*. The *x, y* and *z* components of vector *v* are *v[[1]], v[[2]]* and *v[[3]]*. The expression *u . v* represents the scalar product of two vectors. The expressions *m . v* and *v . m* respectively represent the product of a matrix and column vector and the product of a row vector and a matrix. Definitions are encoded as transformation rules in the form *Lhs -> Rhs*. Each rule describes how an expression matching *Lhs* is replaced with the instantiation of *Rhs*. A pattern variable in *Lhs* has an underscore at the end of its name. The expression *E /. R* indicates the application of rule or rule set *R* to expression *E*. The expression *D[e,v]* is the partial derivative of *e* with respect to *v*.

Initialization:
ParameterNames -> {InitalOmega, ThrustConstant, DragConstant}
ParameterValues -> {InitalOmega -> 5.0, ThrustConstant ->1.0, DragConstant -> 0.0001}
InitializationConstraints -> {LocAV[CarOrigin1][0][[3]] == InitialOmega*DegreesToRadians }

Dynamical Variables:
Rz[CarOrigin1], Tz[Car1], Rx[Car1], Rx[FrontAxle1], Rx[RearAxle1], Rx[RearJoint1], Rz[RearJoint1],
Rz[CarOrigin2], Tz[Car2], Rx[Car2], Rx[FrontAxle2], Rx[RearAxle2], Rx[RearJoint2], Rz[RearJoint2],
Rz[CarOrigin3], Tz[Car3], Rx[Car3], Rx[FrontAxle3], Rx[RearAxle3]

Dynamics:
Masses -> {Car1, Car2, Car3}, Mass[Car1] -> 1.0, Mass[Car2] -> 1.0, Mass[Car3] -> 1.0,
MassType[Car1] -> Point, MassType[Car2] -> Point, MassType[Car3] -> Point,
PEType -> Unit, PE[o_][t_] -> Mass[o]*g*AbsTrans[o][t][[3]],
ForceBearingObjects -> {Car1}, TorqueBearingObjects -> {},
Force[car_][t_] -> Thrust[car,t] + Drag[car,t]

Constraints:
(C1) ForAll[t,AbsTrans[Car1][t][[3]] == Altitude[Revolution[CarOrigin1, t]]],
(C2) ForAll[t,AbsTrans[Car2][t][[3]] == Altitude[Revolution[CarOrigin2, t]]],
(C3) ForAll[t,AbsTrans[Car3][t][[3]] == Altitude[Revolution[CarOrigin3, t]]],
(C4) ForAll[t, Tan[Pitch[Car1, t]] == Slope[Revolution[CarOrigin1, t]]],
(C5) ForAll[t, Tan[Pitch[Car2, t]] == Slope[Revolution[CarOrigin2, t]]],
(C6) ForAll[t, Tan[Pitch[Car3, t]] == Slope[Revolution[CarOrigin3, t]]],
(C7) ForAll[t, CPV[FrontAxle1,FrontContact1, Car1, t][[2]] == 0],
(C8) ForAll[t, CPV[FrontAxle2,FrontContact2, Car2, t][[2]] == 0],
(C9) ForAll[t, CPV[FrontAxle3,FrontContact3, Car3, t][[2]] == 0],
(C10) ForAll[t, CPV[RearAxle1,RearContact1, Car1, t][[2]] == 0],
(C11) ForAll[t, CPV[RearAxle2,RearContact2, Car2, t][[2]] == 0],
(C12) ForAll[t, CPV[RearAxle3,RearContact3, Car3, t][[2]] == 0],
(C13) ForAll[t, Separation[Car1, Car2, t] . Separation[Car1, Car2, t] == (CarLength+LinkLength)^2],
(C14) ForAll[t, Separation[Car2, Car3, t] . Separation[Car2, Car3, t] == (CarLength+LinkLength)^2],
(C15) ForAll[t,XfnP[AbsTrans[FrontJoint2][t],Root,RearJoint1][t][[1]] == 0],
(C16) ForAll[t,XfnP[AbsTrans[FrontJoint2][t],Root,RearJoint1][t][[3]] == 0],
(C17) ForAll[t,XfnP[AbsTrans[FrontJoint3][t],Root,RearJoint2][t][[1]] == 0],
(C18) ForAll[t,XfnP[AbsTrans[FrontJoint3][t],Root,RearJoint2][t][[3]] == 0]

Definitions:
Altitude[angle_] -> Amplitude*Cos[Frequency*angle]
Slope[angle_] -> (D[Altitude[a],a] /. a->angle) / Radius
Amplitude -> 12.5, Frequency -> 4, Radius -> 50.0, CarLength -> 3.0, LinkLength -> 1.0,
Revolution[carOrigin_,t_] -> EulerZ[carOrigin][t],
Pitch[car_,t_] -> EulerX[car][t],
Thrust[car_,t_] -> ThrustConstant*ControlUpDown*Heading[car,t]
Drag[car_,t_] -> -DragConstant*AbsLV [car][t]
Heading[car_,t_] -> XfnV[{0,1,0},car,Root][t]
Separation[carA_,carB_,t] -> AbsTrans[carA][t]-AbsTrans[carB][t]
CPV[axle_,contact_,car_,t_] -> LocLV[car][t]+Cross[RelAV[axle,car,t],RadiusVector[axle,contact,car,t]]
RadiusVector[axle_,contact_,car_,t_] -> RelTrans[contact,car][t] - RelTrans[axle,car][t]

Figure 6. Roller-Coaster Symbolic Specification

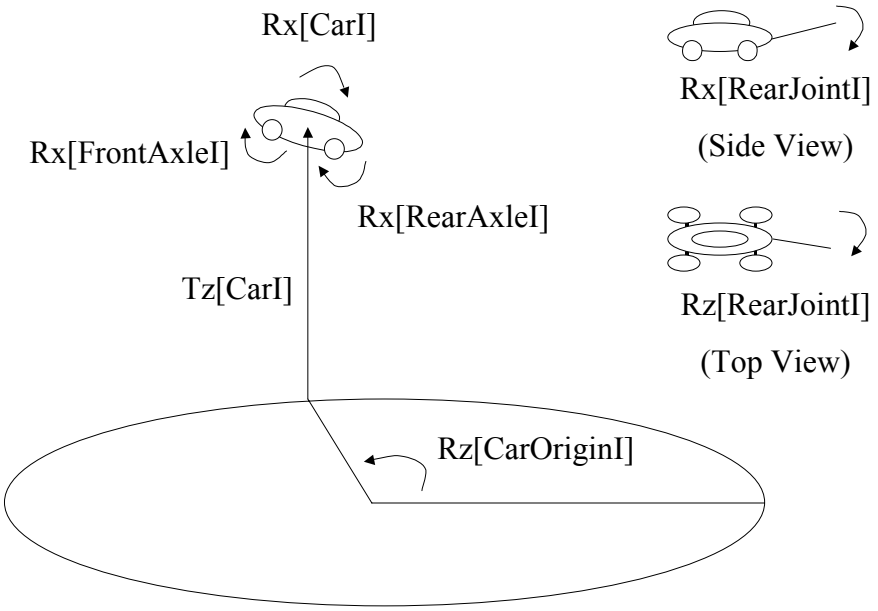

Figure 7. Roller-Coaster Dynamical Variables

In the *DynamicalVariables* section of the specification, the developer provides a list of the system's dynamical variables. Each of these variables represents a parameter to a transformation relating a particular coordinate system to its parent. (See Figure 7.) In the roller-coaster example, the specification includes 19 dynamical variables. The variable *Rz[CarOriginI]* (I=1,2,3) represents the rotation of the *CarOriginI* coordinate system (i.e., the angle of revolution of the car around the center of the track). The variable *Tz[CarI]* represents the translation of the *CarI* coordinate system (i.e., the vertical motion of the car up and down hills of the track). The variable *Rx[CarI]* represents the rotation of the *CarI* coordinate system (i.e., the pitch angle of the car). The variables *Rx[FrontAxleI]* and *Rx[RearAxleI]* represent rotations of the car's wheels. Finally, the variables *Rx[RearJointI]* and *Rz[RearJointI]* (I=1,2) represent the rotation of the joint at the back of *CarI,* which controls the orientation of the link attaching *CarI* to the following car.[2]

[2] Each rotation variable is a component of a unit quaternion (*Rw,Rx,Ry,Rz*) describing a rotation in three dimensions. Whenever the developer includes any of *Rx*, *Ry* or *Rz* in the specification, the system automatically includes *Rw* as well, which is required to maintain normalization of unit quaternions. The roller-coaster system actually has a total of 33 dynamical variables.

In the *Dynamics* section, the developer provides a specification of the Lagrangian and nonconservative forces. The developer does not need to explicitly specify the Lagrangian, since the general form of the Lagrangian (as the difference of kinetic and potential energy) is predefined in our system's knowledge base. Instead he/she specifies properties of the objects in the system, which are then used to determine the manner in which kinetic and potential energy are calculated. In particular, the developer provides a list of the mass-bearing objects along with the mass of each and an indication of whether it should be treated as a *Point* mass (with translational but not rotational kinetic energy) or an extended *Body* mass (with both translational and rotational kinetic energy). In the roller-coaster example, each *CarI* (I=1,2,3) is a mass-bearing object and is treated as a point mass. The developer also provides a specification of the system's potential energy function, along with an indication of whether it is a *Unit* potential (sum over individual objects' potentials) or a *Pair* potential (sum over pairs of objects' interaction potentials). The roller coaster has a *Unit* potential in which the contribution of each car is a simple linear function of the car's altitude. Finally, in order to specify the nonconservative forces, the developer provides lists of the force-bearing and torque-bearing objects, and specifies the forces and torques on each of them. The system uses this information to determine the nonconservative force associated with each dynamical variable. In the roller-coaster example, the first car is the only force-bearing object, and there are no torque-bearing objects. The force on the car is the sum of the frictional drag and the thrust controlled by the interactive user.

- **Tx[o][t], Ty[o][t], Tz[o][t]:** The translation of an object *o*, relative to its parent, at time *t*.
- **Rw[o][t],Rx[o][t], Ry[o][t], Rz[o][t]:** The rotation of an object *o*, relative to its parent, at time *t*.
- **Sx[o][t], Sy[o][t], Sz[o][t]**: The scale of an object *o*, relative to its parent, at time *t*.
- **Masses:** A list of all the mass-bearing objects.
- **MassType[o]:** The mass type of object *o*: **Point** or **Body**.
- **PEType:** The type of potential energy function that governs the system: **Unit** or **Pair**.
- **Mass[o], InertiaTensor[o]:** The mass of an object *o* and its inertia tensor.
- **KE[t], KE[o][t]:** Total kinetic energy and kinetic energy of object *o*, at time *t*.
- **PE[t], PE[o][t], PE[a,b][t]:** Total potential energy, potential energy of object *o*, and interaction potential of objects *a* and *b*, at time *t*.
- **AbsTrans[o][t], AbsRot[o][t], AbsScale[o][t]:** The position, rotation or scale of an object *o*, relative to the root coordinate system, in terms of the root coordinate system, at time *t*.
- **RelTrans[a,b][t], RelRot[a,b][t], RelScale[a,b][t]:** The position, rotation or scale of object *a*, relative to object b's coordinate system, in terms of object *b*'s coordinate system, at time *t*.
- **AbsLV[o][t], AbsAV[o][t]:** The linear or angular velocity of object *o*, relative to the root coordinate system, in terms of the root coordinate system, at time *t*.
- **LocLV[o][t], LocAV[o][t]:** The linear or angular velocity of object *o*, relative to the root coordinate system, in terms of its own coordinate system, at time *t*.
- **RelLV[a,b,t]:** Linear velocity of object *a* relative to object *b*, in terms of object *b*'s coordinate system.
- **RelAV[a,b,t]:** Angular velocity of object *a* relative to object *b*, in terms of object *b*'s coordinate system.
- **EulerX[o][t], EulerY[o][t], EulerZ[o][t]:** The x, y or z Euler angle of the rotation of object *o* relative to its parent, at time *t*.
- **XfnP[p,f,g][t], XfnV[v,f,g][t]**, **XfnAV[w,f,g][t], XfnN[n,f,g][t]:** A position, linear velocity, angular velocity or normal vector transformed from coordinate system *f* to coordinate system *g*, at time *t*.
- **LocNormal[s,p]:** The unit vector normal to surface *s* at location *p* relative to the origin of surface *s*, in terms of the coordinate system of surface *s*.
- **AbsNormal[s,p][t]:** The unit vector normal to surface *s* at location *p* relative to the root coordinate system, in terms of the root coordinate system.
- **Contains[s,o][t]:** A predicate asserting that object *o*'s origin lies in surface *s*, at time *t*.

Figure 8. Selected Specification Language Primitives

The *Constraints* section is a set of temporally quantified equations involving dynamical variables (representing holonomic constraints) or involving both dynamical variables and their derivatives (representing nonholonomic constraints). The (holonomic) constraints C1, C2 and C3 assert that the altitude of each car is equal to the height of the track at the car's current angle of revolution around the center of the track. (Each car is on the track.) The (holonomic) constraints C4, C5 and C6 assert that the tangent of each car's pitch angle is equal to the slope of the track, at the car's current angle of revolution. (The wheels of each car are in contact with the surface of the track.) The (nonholonomic) constraints C7, C8, C9 assert that the velocities of each car's front wheel are zero at the points at which the wheels make contact with the track. The (nonholonomic) constraints C10, C11, C12 make the same assertion for the rear wheels. (The car is not skidding.) The (holonomic) constraints C13 and C14 assert that the distance between successive cars is equal to the car length plus the link length. (Each car is a fixed distance from the car in front or back of it.) Finally, the (holonomic)

constraints C15, C16, C17 and C18 govern the rotation of the rear link joint of each of the first two cars. They assert that each joint rotates to aim its link at the front joint of the next car. (The links appear to connect the cars to each other.)[3]

The primitives of our specification language are implemented in terms of rewrite rules in Mathematica.[4] Rules implementing selected primitives in our specification language are shown in Figure 9. First consider the rules implementing the *Dynamics* section of the specification: The Lagrangian is total kinetic energy minus total potential energy. Total kinetic energy is a sum of the kinetic energies of the masses, including translational energy for *Point* masses and both translational and rotational energy for *Body* masses. The translational kinetic energy of an object depends on its absolute linear velocity, i.e., the derivative of its absolute translation, which is computed by applying translation, rotation and scaling transformations along the path from the object to the root of the coordinate system hierarchy. Potential energy is either a sum over individual masses (*Unit* potential) or a sum over pairs of masses (*Pair* potential). An object's potential energy usually depends on its absolute translation. Thus both kinetic and potential energy depend on parameters (*Tx, Ty, Tz, Rw, Rx, Ry, Rz, Sx, Sy, Sz*) of transformations in the coordinate system hierarchy, or their derivatives, some of which may be dynamical variables.

Now consider the rules implementing the *Constraints* section of the specification. The first rule implements a predicate asserting that a point lies in an implicit surface, i.e., a surface defined by a function of the form *f(x,y,z)=0*. The second rule describes how to compute the unit vector normal to a surface at a point, by evaluating the gradient of the defining function *f*, normalizing the result, and transforming it into the *Root* coordinate system. A position vector *p* is transformed from coordinate system *f* to coordinate system *g* by finding the least common ancestor *a* of *f* and *g*; transforming *p* from *f* to *a*; and then transforming the result from *a* to *g*. These primitives can be used together to assert that one surface is tangent to another surface. Tangency is useful in defining constraints asserting that one object slides or rolls across another.

---

[3] Several additional constraints are needed in the roller-coaster specification. For each rotating coordinate system, the system automatically adds a constraint requiring the corresponding quaternion to be normalized. Thus the roller-coaster is governed by a total of 32 constraints.

[4] The rules use the following Mathematica notation: A rule of the form *Lhs /; Cond := Rhs* asserts that an expression matching *Lhs* and satisfying condition *Cond* can be replaced with the instantiation of *Rhs*. The expression *Sum[e[i],{i,N}]* is the sum of values of *e[i]* for (i=1,…,N). The expression *Dt[f[t],t]* is the total derivative of *f* with respect to *t*. The expression *Gradient[f[{x,y,z}],Cartesian[x,y,z]]* is the gradient of *f* in Cartesian coordinates.

Implementing the Dynamics:

```
Lagrangian[t_] := KE[t] - PE[t]
KE[t_] := Sum[KE[Masses[[m]]][t],{m, NumMasses}]
KE[o_][t_] := TKE[o][t] + RKE[o][t]
TKE[o_][t_] := (1/2) * Mass[o] * (AbsLV[o][t])^2
RKE[o_][t_] /; MassType[o] == Point := 0
RKE[o_][t_] /; MassType[o] == Body := (1/2)Mass[o] * LocAV[o][t] . InertiaTensor[o] . LocAV[o][t]
PE[t_] /; PEType == Unit := Sum[PE[Masses[[m]]][t],{m,NumMasses}]
PE[t_] /; PEType == Pair := Sum[PE[Masses[[m1]],Masses[[m2]]][t],{m1,NumMasses},{m2,NumMasses}]
AbsLV[o_][t_] := Dt[AbsTrans[o][t], t]
AbsTrans[o_][t_] := RelTrans[o, Root][t]
RelTrans[o_,o_][t_] := {0,0,0}
RelTrans[o1_,o2_][t_] /; o1=!=o2 := ApplyTrans[RelTrans[Parent[o1], o2][t],
                                                ApplyRot[RelRot[Parent[o1], o2][t],
                                                         ApplyScale[RelScale[Parent[o1], o2][t], Trans[o1][t]]]]

Trans[o_][t_] := {Tx[o][t], Ty[o][t], Tz[o][t]}
```

Implementing Constraints:

```
Contains[s_, o_][t_] := SurfFn[s][XfnP[AbsTrans[o][t],Root,s][t]] == 0
AbsNormal[s_][p_][t_] := XfnN[LocNormal[s][XfnP[p,Root,s][t]],s,Root][t]
LocNormal[s_][{a_, b_, c_}] := Normalize[Gradient[SurfFn[s][{x, y, z}], Cartesian[x, y, z]] /.{x->a, y->b, c->z}]
XfnP[p_,f_,g_][t_] := XfnPDown[XfnPUp[p,f,LCA[f,g]][t], LCA[f,g],g][t]
XfnPUp[p_,f_,lca_][t_] := ApplyTrans[RelTrans[f,lca][t],
                                     ApplyRot[RelRot[f,lca][t],
                                              ApplyScale[RelScale[f,lca][t],p]]]
XfnPDown[p_,lca_,g_][t_] := ApplyScale[RelScaleInv[g,lca][t],
                                       ApplyRot[RelRotInv[g,lca][t],
                                                ApplyTrans[RelTransInv[g,lca][t],p]]]
```

Figure 9. Rules Implementing Selected Specification Language Primitives

4. Rigid-Body Simulation Programs

Real-time physics-based animation programs typically operate by repeating the following steps: (1) Numerically simulate the behavior of the rigid-body system over a short period of time; (2) Render an image of the current state of the system. In order for this process to operate in real time, the simulation step must be fast enough to execute many times per second. This has important implications for program synthesis.

The general scheme of a program for simulating a constrained rigid-body system is shown in Figure 10. Definitions of the symbols and notation used in the program scheme are shown in Figure 11. The main program begins by initializing the positions and velocities of the dynamical variables. The main loop repeatedly calls a numerical integration routine to integrate a set of dynamical variables over a short time

interval. The integration routine solves a set of differential equations that are first-order in some variables and second-order in other variables. It takes a system derivative function as a parameter and calls this derivative function repeatedly during the numerical integration process. After each integration step, the main loop calls a stabilization routine that adjusts the values of some dynamical variables to maintain numerical stability. After stabilization, it updates the values of other dynamical variables by solving algebraic constraints.

In our simulation program scheme, the dynamical variables have been partitioned into three sets: Q_0, Q_1 and Q_2. Each variable set is handled in a different component of the simulation program scheme. Likewise, the holonomic constraints have been partitioned into three sets (H_0, H_1 and H_2), and the nonholonomic constraints have been partitioned into two sets (N_1 and N_2). Each constraint set is enforced in a different component of the simulation program scheme. Variables in Q_0 (zeroth-order variables) are updated at the end of each simulation step, by solving constraints in H_0. Variables in Q_1 and Q_2 are updated by numerical integration. Sets Q_1 (first-order variables) and Q_2 (second-order variables) differ in the way they appear in the integration process. The variables of integration include Q_1 and Q_2 along with the derivatives $D^1(Q_2)$ of variables in Q_2, but not the derivatives of variables in Q_1. The system derivative function takes the variables of integration (Q_1, Q_2 and $D^1(Q_2)$) as input and computes their derivatives ($D^1(Q_1)$, $D^1(Q_2)$ and $D^2(Q_2)$). The values of variables in $D^1(Q_1)$ (i.e., velocities of variables in Q_1) are computed by solving the constraints in $D^1(H_1)\cup N_1$ (i.e., first derivatives of constraints in H_1 along with constraints in N_1). The values of variables in $D^1(Q_2)$ (i.e., velocities of variables in Q_2) are obtained from the input to the system derivative function. The values of variables in $D^2(Q_2)$ (i.e., accelerations of variables in Q_2) are computed by solving the Euler-Lagrange equations and the constraints in $D^2(H_1\cup H_2)\cup D^1(N_1\cup N_2)$ (i.e., second derivatives of constraints in H_1 and H_2 along with first derivatives of constraints in N_1 and N_2) for values of all variables in $D^2(Q_1\cup Q_2)$ and the multipliers, and extracting the values of variables in $D^2(Q_2)$ from the result.

Given:

- Program parameters: $P = (p_1,\ldots,p_M)$
- Initial user-controlled parameters: $C = (c_1,\ldots,c_M)$
- Initial system configuration: $Q = (Q_0,Q_1,Q_2)$

Find:

- System trajectory $Q(t) = (Q_0(t),Q_1(t),Q_2(t))$ for all $t > 0$.
- Rendering images at times $i \bullet \Delta T$, for $i = 0,1,2,\ldots$ etc.

Initialize(P,Q_0,Q_1,Q_2):

a. Find a minimal norm change $(\Delta Q_0,\Delta Q_1,\Delta Q_2)$ to variables Q_0, Q_1 and Q_2 such that $(Q_0+\Delta Q_0, Q_1+\Delta Q_1, Q_2+\Delta Q_2)$ satisfies constraints $H_0 \cup H_1 \cup H_2$.
b. Solve constraints $I \cup D^1(H_S) \cup N_S$ for variables $D^1(Q_S)$ and extract $D^1(Q_2)$.
c. Return $(Q_0,Q_1,Q_2,D^1(Q_2))$.

Derivative$(P,C)(Q_1,Q_2, D^1(Q_2),t)$:

a. Solve constraints $D^1(H_1) \cup N_1$ for variables $D^1(Q_1)$.
b. Solve equations EL and constraints $D^2(H_1 \cup H_2) \cup D^1(N_1 \cup N_2)$ for variables $D^2(Q_1 \cup Q_2)$, multipliers $\{\lambda_i \mid i=1\ldots|H_1 \cup H_2|\}$ and multipliers $\{\mu_i \mid i=1\ldots|N_1 \cup N_2|\}$ and extract $D^2(Q_2)$.
c. Return $(D^1(Q_1),D^1(Q_2),D^2(Q_2))$.

Stabilize$(P,Q_1,Q_2, D^1(Q_2),t)$:

a. Find a minimal norm change $(\Delta Q_1, \Delta Q_2, \Delta D^1(Q_2))$ to variables Q_1, Q_2 and $D^1(Q_2)$ such that $(Q_1+\Delta Q_1, Q_2+\Delta Q_2, D^1(Q_2)+\Delta D^1(Q_2))$ satisfies constraints $H_1 \cup H_2 \cup D^1(H_2) \cup N_2$.
b. Return $(Q_1+\Delta Q_1, Q_2+\Delta Q_2, D^1(Q_2)+\Delta D^1(Q_2))$.

Step$(P,C,Q_1,Q_2,D^1(Q_2),t,\Delta T)$:

a. $(Q_1,Q_2,D^1(Q_2)) = \text{Integral}(\text{Derivative}(P,C),(Q_1,Q_2,D^1(Q_2)),t,t+\Delta T)$.
b. $(Q_1,Q_2,D^1(Q_2)) = \text{Stabilize}(P,Q_1,Q_2, D^1(Q_2),t+\Delta T)$.
c. Solve constraints H_0 for variables Q_0.
d. Return $(Q_0,Q_1,Q_2,D^1(Q_2))$.

Main(P,C,Q_0,Q_1,Q_2):

1. $(Q_0,Q_1,Q_2,D^1(Q_2)) = \text{Initialize}(P,Q_0,Q_1,Q_2)$.
2. $\text{Render}(Q_0,Q_1,Q_2)$.
3. Let $t = 0$.
4. Repeat:
    a. Let $C = \text{InputDeviceState}()$.
    b. $(Q_0,Q_1,Q_2, D^1(Q_2)) = \text{Step}(P,C,Q_1,Q_2,D^1(Q_2),t,\Delta T)$.
    c. Let $t = t + \Delta T$.
    d. $\text{Render}(Q_0,Q_1,Q_2)$.

Figure 10. Simulation Program Scheme

P	Program parameters.
I	Initialization constraints.
Q_S	Variables used in initial velocity solution.
H_S	Holonomic constraints used in initial velocity solution.
N_S	Nonholonomic constraints used in initial velocity solution.
Q_0, Q_1, Q_2	Partitioning of dynamical variables into sets used in different program components.
H_0, H_1, H_2	Partitioning of holonomic constraints into sets used in different program components.
N_1, N_2	Partitioning of nonholonomic constraints into sets used in different program components.
$Q_i(t)$	Set of values of all variables in set Q_i at time t.
$D^i(Q)$	Set consisting of the i^{th} derivative of each variable in variable set Q.
$D^i(C)$	Set consisting of the i^{th} derivative $D^i(e_1)=D^i(e_2)$ of each constraint $e_1=e_2$ in constraint set C.
EL(q,L,F,H,N)	Euler-Lagrange equation for variable q based on Lagrangian L, nonconservative forces F, holonomic constraints H and nonholonomic constraints N.
EL	Set of all Euler-Lagrange equations for a given system.

Figure 11. Definitions for Program Scheme and Classification Algorithm

Our simulation programs incorporate several routines from the Numerical Recipes library [Press et al., 1986], including an adaptive Runge-Kutta routine for integration of differential equations; an LU decomposition routine for solving systems of linear algebraic equations, and a Newton-Raphson routine for solving systems of nonlinear algebraic equations. The Newton-Raphson routine has been modified to handle under-constrained systems of equations as well as fully constrained systems. It operates by computing a pseudo-inverse to the equation system Jacobian, rather than a true inverse, when the system is under-constrained. Our approach to simulation of rigid-body systems is based on numerical techniques described in [Haug, 1989]; however, we use a Runge-Kutta method, rather than a predictor-corrector method, for carrying out the main integration step. We also use an explicit stabilization step incorporating the modified Newton-Raphson routine to maintain numerical stability, instead of the dynamical variable partitioning technique described in [Haug, 1989].

The performance of the simulation program depends on the manner in which the dynamical variables and constraints are partitioned into the sets described above. One consideration involves the computational cost of enforcing constraints in parts (a) and (b) of the Derivative function and part (c) of the Step function. Each constraint may be enforced in terms of its zeroth, first or second derivative. Constraints usually become larger and more expensive to evaluate each time they are differentiated. This argues for placing holonomic constraints in set H_0 (where they are enforced without being differentiated at all) in preference to H_1 (where they are differentiated once) and in H_1 in preference to H_2 (where they are differentiated twice). It also argues for placing nonholonomic constraints in set N_1 in preference to N_2. A second consideration involves the cost of the Stabilize function. This function is implemented using a variation on the Newton-Raphson algorithm to find changes to the values of the variables in sets Q_1, Q_2 and $D^1(Q_2)$ that satisfy the constraints in sets H_1, H_2,

$D^1(H_2)$ and N_2. The cost of the algorithm is dominated by a step that constructs the Jacobian of the constraints with respect to the solution variables and then finds the pseudo-inverse of the Jacobian. Constructing the Jacobian takes time $O(mn)$ where $m = |H_1|+2|H_2|+|N_2|$ and $n = |Q_1|+2|Q_2|$. Inverting the Jacobian takes $O(n^3)$ operations. This also argues for classifying the constraints as described above, and argues for placing variables in set Q_0 in preference to Q_1, and in set Q_1 in preference to Q_2.

5. Synthesis of Simulation Programs

Our program synthesis procedure is divided into three stages. The first stage takes the program specification as input and generates a functional program as its output. The main task of this stage is to instantiate a parameterized program scheme corresponding to the generic simulation and animation program outlined above. It does so by classifying the dynamical variables and constraints into sets that correspond to components of the program scheme and selecting numerical methods for solving each set. The second stage takes the initial functional program as input and generates an optimized functional program as its output. The main task of this stage is to improve the performance of the initial functional program. It does so by decomposing numerical program components and optimizing the flow of data to avoid unnecessary computation. Finally, the third stage takes the optimized functional program as input and generates a C^{++} program as output. The main tasks of this stage are to define C^{++} function object classes implementing functional parameters to higher-order numerical procedures, and to generate C^{++} functions that implement the optimized functional program.

5.1 Classification of Variables and Constraints

The classification of variables and constraints can be formulated as a combinatorial optimization problem. The problem inputs include the dynamical variables Q, initialization constraints I, holonomic constraints H, and nonholonomic constraints N, along with the Lagrangian L and nonconservative forces F. A solution is a description of the variable and constraint sets (Q_S, H_S, N_S, Q_0, Q_1, Q_2, H_0, H_1, H_2, N_1, N_2 and EL) that appear in the generic simulation algorithm. A solution is feasible if it allows the equation-solving operations appearing in the simulation algorithm to be carried out. (See Figure 10.) Thus constraints H_0 must be solvable for variables Q_0. (See line (c) of the Step function.). Constraints $D^1(H_1)\cup N_1$ must be solvable for variables $D^1(Q_1)$. (See line (a) of the Derivative function). Equations EL and constraints $D^2(H_1\cup H_2)\cup D^1(N_1\cup N_2)$ must be solvable for variables $D^2(Q_1\cup Q_2)$ and the multipliers. (See line (b) of the Derivative function). In addition, $Q_0\cup D^1(Q_0)$ must not contain variables referenced in L, F, $H_1\cup H_2$, $N_1\cup N_2$, because these expressions are referenced in the

Euler-Lagrange equations EL that are solved in the system derivative function, where variables $Q_0 \cup D^1(Q_0)$ are not available. Likewise, $D^1(Q_1)$ must not contain variables referenced in $D^1(H_2) \cup N_2$, because these expressions are referenced by the stabilization function, where variables $D^1(Q_1)$ are not available. Finally, the initialization variables Q_S must include the second-order dynamical variables Q_2, whose velocities are needed to initialize the simulation; and the initialization constraints $I \cup D^1(H_S) \cup N_S$ must be solvable for the initialization variables $D^1(Q_S)$. Subject to these conditions, the solution should maximize the numbers of constraints placed in H_0 rather than H_1 or H_2; H_1 rather than H_2; and N_1 rather than N_2.

An exhaustive search for the optimal classification appears to be computationally infeasible, except for small problem instances. For this reason, our classification algorithm uses a greedy method to find an approximation to the optimal solution. (See Figure 12.) It begins by finding Q_0 and H_0 such that Q_0 is as large as possible and constraints H_0 can be solved for variables Q_0. Then it examines the remaining variables and constraints to find Q_1, H_1 and N_1 such that Q_1 is as large as possible and constraints $D^1(H_1) \cup N_1$ can be solved for variables Q_1. Finally, it forms Q_2, H_2 and N_2 from the remaining variables and constraints. The algorithm also finds variable and constraint sets (Q_S, H_S and N_S) sufficient for initializing the simulation, and constructs the Euler-Lagrange equations EL based on the variable and constraint classification. The algorithm can fail to find an admissible classification in either of two possible ways. One such possibility arises if no suitable initialization parameters (Q_S, H_S and N_S) can be found. Another possibility arises if the Euler-Lagrange equations cannot be solved for the accelerations of the second-order dynamical variables. In either case, the problem specifications are incomplete, or outside the standard Euler-Lagrange formalism. (For example, the developer may have neglected to provide a constraint governing the behavior of a noncausal variable; or he/she may have specified a variable to be noncausal, when it really should be treated as a causal variable.) In such cases, the system exits with an error message describing the problem. The algorithm may fail to find the optimal classification of variables and constraints. In such cases, the resulting program may not perform as well as a program based on the optimal classification.

Given:

- Initialization constraints I.
- Dynamical variables Q.
- Lagrangian L.
- Nonconservative forces F.
- Holonomic constraints H.
- Nonholonomic constraints N.

Find:

- Partitioning of dynamical variables Q into sets Q_0, Q_1 and Q_2.
- Partitioning of holonomic constraints H into sets H_0, H_1 and H_2.
- Partitioning of nonholonomic constraints N into sets N_1 and N_2.
- Initialization variables Q_S and constraints H_S and N_S.
- Euler-Lagrange equations EL.

1. **Identify the zeroth-order solution variables and constraints:**
   Find $Q_0 \subseteq Q$ and $H_0 \subseteq H$ such that Q_0 is as large as possible; H_0 can be solved for variables Q_0 in terms of variables $Q\text{-}Q_0$; and L, F and $(H\text{-}H_0)\cup N$ are free of variables $Q_0 \cup D^1(Q_0)$.
2. **Identify the first-order solution variables and constraints:**
   Find $Q_1 \subseteq Q\text{-}Q_0$, $H_1 \subseteq H\text{-}H_0$ and $N_1 \subseteq N$ such that Q_1 is as large as possible; $D^1(H_1)\cup N_1$ can be solved for variables $D^1(Q_1)$ in terms of variables $(Q\text{-}Q_0)\cup D^1(Q\text{-}Q_0\text{-}Q_1)$; and $D^1(H\text{-}H_0\text{-}H_1)\cup(N\text{-}N_1)$ is free of variables $D^1(Q_1)$.
3. **Identify the second-order solution variables and constraints:**
   Let $Q_2 = Q\text{-}Q_0\text{-}Q_1$. Let $H_2 = H\text{-}H_0\text{-}H_1$. Let $N_2 = N\text{-}N_1$.
4. **Identify variables and constraints for initialization:**
   a. Determine whether there exist $Q_S \subseteq Q$, $H_S \subseteq H$ and $N_S \subseteq N$ such that $Q_2 \subseteq Q_S$ and $I\cup D^1(H_S)\cup N_S$ can be solved for variables $D^1(Q_S)$ in terms of program parameters P.
   b. If not, exit with an error message stating that the specifications are incomplete or outside the standard Euler-Lagrange formalism.
   c. Otherwise, let Q_S, H_S and N_S be the smallest such sets.
5. **Generate and validate the Euler-Lagrange equations:**
   a. $EL = \{EL(q, L, F, H_1\cup H_2, N_1\cup N_2) \mid q \in Q_1\cup Q_2\}$.
   b. Determine whether equations EL and constraints $D^2(H_1\cup H_2)\cup D^1(N_1\cup N_2)$ can be solved for variables $D^2(Q_1\cup Q_2)$, multipliers $\{\lambda_i \mid i=1\ldots|H_1\cup H_2|\}$ and multipliers $\{\mu_i \mid i=1\ldots|N_1\cup N_2|\}$.
   c. If not, exit with an error message stating that the specifications are incomplete or outside the standard Euler-Lagrange formalism.
6. **Return** Q_S, H_S, N_S, Q_0, Q_1, Q_2, H_0, H_1, H_2, N_1, N_2 and EL.

Figure 12. Variable and Constraint Classification Algorithm

The behavior of the variable and constraint classification algorithm can be illustrated by considering the roller-coaster example. In the initial formulation, the roller-coaster specification has 33 dynamical variables and 32 constraints. These variables and constraints are classified in the following way: In step (1) the link-joint rotation variables (*Rw[RearJoint1], Rx[RearJoint1], Rz[RearJoint1], Rw[RearJoint2], Rx[RearJoint2], Rz[RearJoint2]*) are placed in Q_0, since they can be computed from the link-aiming constraints (C15, C16, C17, C18) along with their quaternion normalization constraints. (The link aiming and quaternion normalization constraints are placed in H_0.) In step (2) the cars' pitch rotation variables (*Rw[Car1], Rx[Car1], Rw[Car2], Rx[Car2], Rw[Car3], Rx[Car3]*) are placed in Q_1, since their derivatives can be computed from the derivatives of the car pitch constraints (C4, C5, C6), along with their quaternion normalization constraints. (The car pitch and quaternion normalization constraints are placed in H_1.) In addition, the cars' axles' rotation variables (*Rw[FrontAxle1], Rx[FrontAxle1], Rw[RearAxle1], Rx[RearAxle1], Rw[FrontAxle2], Rx[FrontAxle2], Rw[RearAxle2], Rx[RearAxle2], Rw[FrontAxle3], Rx[FrontAxle3], Rw[RearAxle3], Rx[RearAxle3]*) are also placed in Q_1, since they can be computed from the no-skid constraints (C7, C8, C9, C10, C11, C12), along with their quaternion normalization constraints. (The no-skid constraints are placed in N_1 and the quaternion normalization constraints are placed in H_1.) In step (3), the cars' origins' rotation variables (*Rw[CarOrigin1], Rz[CarOrigin1], Rw[CarOrigin2], Rz[CarOrigin2], Rw[CarOrigin3], Rz[CarOrigin3]*) and the cars' vertical translation variables (*Tz[Car1], Tz[Car2], Tz[Car3]*) are placed in Q_2. (The inter-car distance constraints (C13, C14), car altitude constraints (C1, C2, C3) and the remaining three quaternion normalization constraints are placed in H_2.)

A key subtask appears in steps (1) and (2) of the variable and constraint classification algorithm: Given a set V of variables and a set C of constraints, find the largest $V' \subseteq V$ and $C' \subseteq C$ such that constraints C' can be solved for variables V'. An exhaustive search would be computationally expensive. We use two ideas to make the search tractable. First we partition the variables V and constraints C into components that can be considered separately, by doing depth-first searches on graphs describing interdependence of variables and interdependence of constraints, resulting in partitions $V=\{V_i|i=1\ldots n\}$ and $C=\{C_i|i=1\ldots n\}$. Next we use another greedy method to find solvable subsets V_i' and C_i' of each pair of components V_i and C_i. For this purpose, we set an a priori upper bound *k* on the size of any system of simultaneous equations we will consider at once.[5] We find sets V_i' and C_i' by repeatedly extracting subsets V_{ij} and C_{ij} from V_i and C_i such that $|V_{ij}|=|C_{ij}|\leq k$ and C_{ij} can be solved for V_{ij}. After extracting *m* such subsets and reaching a point at which no more solvable

[5] In all of our experiments, we set k=7 so that a single system of equations could in principle be solved for all the dynamical variables representing the position and orientation of a single rigid body. In hindsight, we observed that the largest analytically solved system had 6 equations and 6 unknowns. Therefore k=6 would have produced the same classifications in all cases. In many cases, k=4, k=3 or k=2 would have been sufficient to obtain the same classification as when k=6.

subsets can be extracted, we let $V_i' = \cup_j V_{ij}$ (j=1…m) and let $C_i' = \cup_j C_{ij}$ (j=1…m). In effect, we are requiring that the equation system determined by V_i' and C_i' be serially decomposable into components of size *k* or less. Finally, we let $V' = \cup_i V_i'$ (i=1…n) and $C' = \cup_i C_i'$ (i=1…n).

Another key subtask appears in steps (1), (2), (4) and (5) of the classification algorithm. Given a set C of constraints and a set V of variables, determine whether C can be solved for V. The problem is difficult in general for two reasons: The constraints C may be nonlinear in the solution variables V. Furthermore, even if the constraints are linear, their coefficients may be nonlinear functions of other dynamical variables and their derivatives. We take a conservative approach to the problem, based on examining the determinant D of the Jacobian of the constraints C with respect to the variables V. In steps (1) and (2) we are trying to improve the efficiency of the program. In these steps our priority is to avoid generating unsolvable problems, so we apply criteria that avoid false positives: We consider the constraints to be solvable only if D is a nonzero constant, or if all the variables in V are noncausal and C includes all the constraints referencing variables in V. (In this latter case, the constraints must be solvable, or else the specifications are incomplete.) In steps (4) and (5) we are judging the adequacy of the developer's specifications. In these steps our priority is to avoid rejecting valid specifications, so we apply criteria that avoid false negatives: We consider the constraints to be solvable as long as D is not identically zero.

The complexity of our variable and constraint classification algorithm can be analyzed as follows. Each time the algorithm searches for sets V_{ij} and C_{ij} it must examine at most $O(v^k c^k)$ pairs of sets, where v=|Q| is the number of dynamical variables, c=|H|+|N| is the number of constraints, and k is our upper bound on the number of constraints to be considered simultaneously during the classification search process. A complete run of the classification algorithm can extract at most Min[$v/k, c/k$] such sets. Evaluation of each system's Jacobian determinant takes at most $O(k^3)$ operations. The overall complexity is therefore $O(k^2\ \mathrm{Min}[v,c]\ v^k c^k)$. This bound is exponential in k, the largest number of simultaneous constraints; however, it is polynomial in the numbers v and c of variables and constraints.

5.2 Instantiation of the Program Scheme

The internal representation of our parameterized simulation program scheme is shown in Figure 13. The program scheme is written in a higher-order functional language. The primitive operations of this language are described in Figure 14. Instances of this program scheme are designed to interface with our main simulation and animation driving program, through the *programParameters* (values that change from one run of the program to another), *controlParameters* (values that change dynamically in response to user inputs during a

run of the program), *initialPositions* (initial values of the dynamical variables), *stateVariables* (current values of all dynamical variables and velocities of second-order dynamical variables) and the current *time*.

```
Initialize -> Function[{programParameters,initialPositions},
                  Let[{{constrainedPositions, NonLinearSolution[InitializationResidualFunction[programParameters],
                                                                initialPositions]}},
                      Concatenate[constrainedPositions,
                                  InitialVelocities[programParameters,constrainedPositions]]]]


Step -> Function[{programParameters,controlParameters,stateVariables,time,deltaTime},
                Let[{{newTime, time+deltaTime},
                     {integrationVariables, Stabilize[programParameters,
                                                      Integrate[DerivativeFunction[programParameters, controlParameters],
                                                                Slice[stateVariables,IntegrationVariablesRange],
                                                                time,
                                                                newTime],
                                                      newTime]}},
                      Concatenate[ZerothOrderVariablesPositions[programParameters,
                                                                integrationVariables,
                                                                Slice[stateVariables,ZerothOrderVariablesRange],
                                                                time],
                                  integrationVariables]]]


DerivativeFunction  -> Function[{programParameters,controlParameters},
                           Function[{integrationVariables,time},
                               Let[{{velocities, FirstOrderVariablesVelocities[programParameters,
                                                                               integrationVariables,
                                                                               time]}},
                                   Concatenate[velocities,
                                               Slice[integrationVariables,
                                                     SecondOrderVariablesVelocitiesRange],
                                               SecondOrderVariablesAccelerations[programParameters,
                                                                                 controlParameters,
                                                                                 integrationVariables,
                                                                                 velocities,
                                                                                 time]]]]]


Stabilize -> Function[{programParameters,integrationVariables,time},
                NonLinearSystem[StabilizationResidualFunction[programParameters,time],
                                integrationVariables]]
```

Figure 13. Internal Representation of Simulation Program Scheme

- **Integral[Derivative,InitialState,TStart,TEnd]**: The result of integrating the *Derivative* function starting in *InitialState* from time *TStart* to time *TEnd*.
- **LinearSystem[Matrix,Vector]**: The solution to the system of linear equations defined by *Matrix* and *Vector*.
- **NonLinearSystem[Residual,Seed]**: The result of using the Newton-Raphson algorithm to solve the system of nonlinear equations defined by the *Residual* function, starting at *Seed*.
- **Function[{p1,…,pN},Body]**: A function of N formal parameters (*p1*,…,*pN*) that returns *Body*.
- **Let[{{v1, e1},…,{vN,eN}}, Body]**: An expression that evaluates and returns *Body* in an environment in which local variables (*v1*,…,*vN*) have been initialized to (*e1*,…,*eN*).
- **Element[Vector,N]**: The *N*th element of *Vector*.
- **Slice[Vector,{Low,High}]**: The portion of *Vector* running from *Low* to *High*.
- **Shred[Vector,{Index1,…,IndexN}]**: A vector formed by selecting elements of *Vector* at positions specified by *Index1*,…,*IndexN*.
- **Concatenate[Vector1,Vector2]**: The result of concatenating *Vector1* and *Vector2*.
- **Shuffle[Vector1,Vector2,{Index1,…,IndexN}]**: The result of inserting elements of *Vector2* into *Vector1* at locations specified by *Index1*,…,*IndexN*.
- Arithmetic and trigonometric scalar expressions.

Figure 14. Primitives of the Internal Functional Language

The major parameters needed to instantiate the program scheme are indicated in boldface type. The variable and constraint classification algorithm provides the information needed to synthesize them. Most of these parameters are functions that solve sets of equations (or constraints) for the values, first derivatives, or second derivatives of dynamical variables. **InitialVelocities** solves constraints $I \cup D^1(H_S) \cup N_S$ for initial velocities $D^1(Q_S)$ and extracts $D^1(Q_2)$ from the solution. **ZerothOrderVariablesPositions** solves constraints H_0 for variables Q_0. **FirstOrderVariablesVelocities** solves constraints $D^1(H_1) \cup N_1$ for variables $D^1(Q_1)$. **SecondOrderVariablesAccelerations** solves equations EL and constraints $D^2(H_1 \cup H_2) \cup D^1(N_1 \cup N_2)$ for variables $D^2(Q_1 \cup Q_2)$ and the multipliers and extracts $D^2(Q_2)$ from the solution. Two of the program scheme parameters construct residual functions that are used in solving under-constrained systems of equations. **InitializationResidualFunction** is used in the *Initialize* function to enforce constraints on the initial positions of the dynamical variables. It constructs a function that computes the error in the constraints $H_0 \cup H_1 \cup H_2$, i.e., the holonomic constraints. **StabilizationResidualFunction** is used in the *Stabilize* function to enforce constraints on the integration variables. It constructs a function that computes the error in the constraints $H_1 \cup H_2 \cup D^1(H_2) \cup N_2$ that require stabilization.

Functional code implementing each program scheme parameter is generated in the following way: First a set of equations or residual expressions is associated with the program scheme parameter, based on the results of the variable and constraint classification algorithm. Next the equations or residual expressions are lambda-abstracted, so that they may be expressed in terms of inputs to the function being synthesized. (Two of the

program scheme parameters are lambda-abstracted twice, since they return functional values.) If the program scheme parameter solves a set of equations, the equations are classified as linear or nonlinear, by generating the Jacobian of the equation system. If solution variables appear in the Jacobian, the equations are nonlinear. Otherwise they are linear. Then an appropriate solution method is selected. Linear systems are solved analytically (at synthesis time), if they are smaller than a preset bound. Otherwise they are solved numerically (at run time) by LU decomposition. Nonlinear systems are always solved numerically (at run time), using the Newton-Raphson algorithm. We do not attempt to find analytic solutions to nonlinear systems of equations. Once a solution method is selected, it remains only to generate an expression with the appropriate function call, using the corresponding primitive operation of our functional language. (See Figure 14.)

In the roller-coaster example, the program scheme parameters operate in the following way: **InitializationResidualFunction** constructs a function that computes the error in 26 holonomic constraints as a function of 33 position variables. **InitialVelocities** solves a linear system of 9 equations and unknowns for the initial velocities of the cars' altitude and revolution variables. **ZerothOrderVariablesPositions** solves a nonlinear system of 6 equations and unknowns for the rear joint rotations of the first two cars. **FirstOrderVariablesVelocities** solves a linear system of 18 equations and unknowns for the velocities of the cars' pitch and axle rotation variables. **SecondOrderVariablesAccelerations** solves a linear system of 53 equations and unknowns for the accelerations of the cars' altitude and revolution variables. **StabilizationResidualFunction** constructs a function that computes the error in 28 constraints as a function of 36 integration variables.

5.3 Optimization of the Functional Program

The initial functional program can often be optimized by decomposing numerical solutions into components that can be solved sequentially or independently of each other. One case occurs when the dynamical variables can be partitioned into two sets that do not interact with each other. In such cases a single numerical integration of the entire system can be decomposed into two independent integrations of smaller sets of variables. Decomposition will improve performance if the integration is performed using an implicit method (by decreasing the size of the nonlinear algebraic system to be solved on each time step) or an adaptive method (by allowing the component integrations to refine the numerical step size at different points on the time line). Another opportunity for decomposition occurs when the variables and constraints of a nonlinear system of equations can be partitioned into sets such that no equation references variables in two different variable sets, and no variable is referenced by equations in two different equation sets. In such cases, each pair of sets of variables and equations can be solved independently of the others. Decomposition improves performance by

decreasing the size of the largest Jacobian matrix to be computed and inverted on each iteration of the Newton-Raphson algorithm. Finally, another opportunity for decomposition occurs when the variables and equations of a linear system can be ordered to put the matrix in block triangular form, so that the blocks can be solved sequentially or independently. Decomposition improves performance since it decreases the size of the largest component to be solved. Decomposition may also improve performance if one or more of the component systems are small enough to be solved analytically.

Our system implements these decompositions by applying program transformation rules to the initial functional program. The transformations that decompose calls to the *Integral* and *NonLinearSystem* functions are quite similar to each other. In each case, we begin by analyzing the input/output dependencies of the function representing the system derivative or nonlinear equation residual. We use the results of the dependency analysis to partition the input and output elements into independent components, by doing depth-first searches on graphs describing the interdependence of inputs and the interdependence of outputs. Finally, we define a separate *Integral* or *NonLinearSystem* function call for each pair of input and output graph components. Our procedure for decomposing the *LinearSystem* function is based on the standard method for constructing the block triangular form of a matrix, using a combination of algorithms for maximal matching and strongly connected components of graphs defined in terms of the matrix of the linear system [Pissanetsky, 1984].

The efficiency of the initial functional program is further improved using transformations that optimize the flow of data. Two types of improvements are often available. To begin with, the program may include computations that simply are not needed to carry out the simulation. For example, this happens if a linear system of size n is being solved only to obtain values of $m<n$ of the variables, and the system has been decomposed so that some or all of the unneeded variables are in components separate from the needed ones. We remove unneeded computations using transformation rules that implement a kind of dependency tracking. A key part of the process involves transposing vector access functions (*Element, Slice, Shred*) and vector constructor functions (*Concatenate, Shuffle*). For example, suppose a function definition contains vector expressions x and y that result from decomposing a system of equations into two components, and that the expression *Concatenate[x,y]* represents the recomposed solution to the original equation system. Suppose further that only the solution variables in x, or only the variables in y, are needed in the subsequent computation. In this case, the function definition might contain expressions of the form *Element[Concatenate[x,y],i]* to extract the required solution variables. This expression can be simplified to *Element[x,i]* (if $i \leq$*Length[x]*) or *Element[y,i-Length[x]]* (if $i>$*Length[x]*), because the lengths of all vectors are known at this time. In either case, one of the arguments of *Concatenate* is eliminated from the computation.

Analogous transformations apply when solutions are recomposed using the *Shuffle* vector constructor, and when the required solution variables are extracted using the *Slice* or *Shred* access functions.

Additional improvements are possible if the functional program includes multiple occurrences of identical complex arithmetic expressions. For example, repeated expressions are often generated by our rules that define transformations between coordinate systems. Repeated expressions also result from symbolic differentiation of the Lagrangian and constraints. Repeated evaluation of complex expressions can be a major source of inefficiency in the resulting program. For this reason we apply a set of program transformation rules that identify and collect repeated nontrivial expressions and arrange for them to be evaluated just once and stored in local variables.

The initial functional roller-coaster program is optimized in the following way: The *Step* function includes a call to the *Integral* function. This computation is not decomposable, because all of the roller coaster's integration variables interact with each other, either directly or indirectly. The *Initialize* function includes a call to the *NonLinearSystem* function, passing it a function constructed by **InitializationResidualFunction**. Prior to decomposition, this nonlinear system involves 26 equations and 33 unknowns. After decomposition, the system consists of 7 components, the largest of which involves 20 equations and 21 unknowns. The linear system in **InitialVelocities** is decomposed into two components, each of size 3 (each of which is solved numerically by LU decomposition) and two components of sizes 2 and 1 (each of which is solved analytically). The nonlinear system in **ZerothOrderVariablesPositions** is decomposed into two components, each of size 3, (each of which is solved numerically using the Newton-Raphson algorithm). The linear system in **FirstOrderVariablesVelocities** is decomposed into 9 components, each of size 2 (each of which is solved analytically). The initial version of **SecondOrderVariablesAccelerations** solves a linear system of 53 equations and unknowns. After decomposition, the system consists of 10 components, the largest of which involves 17 variables and unknowns. Subsequent dataflow analysis determines that only one of these components (the largest) is needed to compute the accelerations of the second-order dynamical variables. The dataflow analysis therefore eliminates these components from the function definition. (The remaining component is solved numerically by LU decomposition.) The *Stabilize* function includes a call to the *NonLinearSystem* function, passing it a function constructed by **StabilizationResidualFunction**. Prior to decomposition, this nonlinear system involves 28 equations and 36 unknowns. After decomposition, the system consists of 7 components, the largest of which involves 22 equations and 24 unknowns. When all of these optimizations are turned off, the system generates a roller-coaster program that runs at about 8 frames per second, i.e., too slowly for real time animation. When all of the optimizations are used, the generated program

runs at about 30 frames per second, i.e., just fast enough for real time animation. At this point, the computational cost of animation is dominated by the rendering step. Simulation is no longer the bottleneck.

5.4 Translation of the Functional Program into C++

Translation of the functional program into C++ is carried out in two steps. The first step is to translate the functional program into a hierarchical expression representing the abstract syntax of the C++ program. This step is implemented by rewrite rules that conduct a depth-first traversal of the functional program. The rules operate by matching the principal constructs of the functional language; generating the corresponding C++ function calls; and recursively generating C++ expressions for the functions' arguments. These rules also generate appropriate function and variable declarations and storage allocation statements. Once the abstract C++ expression has been generated, additional translation rules construct a concrete C++ program by instantiating parameterized string templates.

Expressions that reference higher-order functions, such as *Integral* and *NonLinearSystem,* require special treatment. The key idea is to translate the functional arguments of these procedures into C++ function objects. Each time the translation rules encounter an invocation of a higher-order function, they define a C++ class to implement the functional argument of the higher-order function. They also define data members to hold values of the free variables of the functional argument, along with a constructor that initializes the data members. The body of the functional argument is translated into the implementation of the function call operator for the class. The call to the higher-order function is translated into an expression that constructs an instance of the function object class and passes it to the appropriate higher-order numerical routine.

5.5 Specification Errors and Run-Time Exceptions

Programs synthesized by our system may fail to run properly if the original specification contains errors. For example, one common error occurs when the developer specifies a set of constraints that are not independent of each other, usually because a dependent constraint needs to be replaced with an independent constraint that was mistakenly omitted. Specification errors can cause our system to generate singular (unsolvable) sets of equations and incorporate them into numerical routines in the synthesized program. Our system attempts to identify singular systems of equations at program synthesis time, by examining the symbolic determinant of each linear system, or the symbolic Jacobian determinant of each nonlinear system. Unfortunately, our program is unable to detect singularities when the determinant depends on dynamic variables, and is zero for

some values of the variables and nonzero for others. The result may be a numerical exception at run time. When this happens, the run-time system exits after reporting the name of the numerical routine in which the error occurred. It appears possible to generate exception-handling code to tell the user what part of his/her specification contains the error; however, we have not attempted to develop such a facility in our work to date. Numerical exceptions may also occur even when the specifications are correct, if the developer specifies a "stiff" dynamical system. Our numerical integration routine (Runge-Kutta) is not able to handle stiff systems and exits with an error message when it encounters one.

5.6 Implementation of the Program Synthesis System

Our program synthesis system is implemented as rewrite system in the Mathematica programming language. The implementation consists entirely of declarative rewrite rules. Some rules implement our symbolic specification language. Others implement our criteria and methods for classification of variables and constraints. Still others instantiate the simulation program scheme and optimize the functional program. Some of the rules invoke Mathematica's predefined functions for symbolic differentiation, computation of determinants, analytic solution of linear algebraic equations, and simplification of algebraic expressions. None of the rules invokes the imperative features of Mathematica. The entire system includes roughly 350 rules comprising about 2300 lines of Mathematica code.

The rewrite system is deterministic. Mathematica includes no explicit search or backtracking mechanism. Once a rule is applied, its effects are never undone. Nevertheless, in some parts of our system, we implement a search algorithm in terms of deterministically applied rewrite rules. Some of these search algorithms occur in the context of our variable and constraint classification algorithm, at points where we search for solvable sets of variables and constraints. (See Figure 12.) Others occur in the context of our methods for optimizing the synthesized program, where we conduct depth-first search on dependency graphs in attempting to decompose integrals and systems of equations.

We conjecture that our declarative implementation promotes transparency and maintainability of the program synthesis system. In particular, the rules implementing the Dynamics and Constraints portions of program specifications could probably be modified without interacting adversely with the remaining portions of the system. (See Figure 9.) For example, the rules implementing kinetic and potential energy functions might be modified to handle other types of rigid bodies and potential fields. Likewise, the rules implementing surface containment and surface normal functions might be modified to handle other types of surfaces, such as

Nonuniform Rational B-Splines (NURBS). Nevertheless, we have not attempted to run experiments designed to test the transparency and maintainability of our program synthesis system.

6. Experimental Results

Our system has been successfully tested on over sixteen qualitatively distinct example problems. A summary of these results is shown in Figure 15. In each of these example programs, the developer carried out the following steps: (1) Enter the graphical and symbolic components of the specification; (2) Execute the program synthesis algorithm; (3) Compile the generated C^{++} code; (4) Execute the resulting animation program. Program synthesis can be done with or without decomposition and dataflow optimization. When these two optimizations are turned off, program synthesis time ranges from about 3 seconds on the simplest problems to 1 minute and 45 seconds on the most complicated. When the optimizations are turned on, program synthesis time ranges from about 3 seconds on the simplest problems to 22 minutes on the most complicated. (These timing results were obtained running the system on a PC workstation using a Pentium IV processor running at 2.0 GHz.) A typical developer would probably use the system with optimization turned off during the development phase, as he/she repeatedly generates a program, observes its behavior, and modifies it according to his/her purpose. On the other hand, the developer would probably want to carry out full optimization when producing a final product that must run as fast as possible in real time. Our timing results are consistent with this mode of operation.

One might ask how the time and effort required to construct an animation program using our system compares to the conventional approach of coding directly in C^{++}, perhaps utilizing numerical libraries or a physics-based animation toolkit. Precise comparisons are not available in the absence of experiments with real human programmers. Nevertheless, a qualitative idea of the difference may be obtained by comparing the size of the specification provided to our system to the size of the C^{++} program it generates. Consider the pendulum, and the three-car roller coaster, i.e., the simplest and most complex examples in Figure 15. The pendulum specification comprises about 10 lines of Mathematica rules and expressions. The generated program has 158 lines of code (in a C^{++} file of 6.32 KB), i.e., a code to specification ratio of about 16. The roller-coaster specification comprises about 40 lines of Mathematica rules and expressions. The generated program has 3477 lines of code (in a C^{++} file of 578KB), i.e., a code to specification ratio of about 89. (The sizes of the C^{++} programs include only the code actually generated by our system. They do not include the numerical libraries or the rendering component of the animation program.) We do not claim that these ratios can be used reliably to predict the time that would be saved by a software developer using our system. Nevertheless, they do provide some evidence that significant savings in time and effort would result.

- **Pendulum**: Demonstrates basic system operation. (2 Variables; 1 Constraint.)
- **Double Pendulum**: One pendulum hangs off another. Demonstrates coordinate system hierarchy. (4 Variables; 2 Constraints.)
- **Three Body Planetary System**: Demonstrates use of a Newtonian gravitational potential. (6 Variables; 0 Constraints.)
- **Two Spring-Coupled Pendula**: Two pendula are linked by a spring. Demonstrates use of a potential describing Hooke's law. Also demonstrates decomposition of systems of linear algebraic equations. (8 Variables; 6 Constraints.)
- **Two Rigidly Linked Pendula**: Two pendula are linked by a rigid rod. Demonstrates handling of constraint systems forming a graph, not a tree. (8 Variables; 7 Constraints.)
- **Pendula on Spinning Wheel**: Four pendula are attached to wheel spinning with time-varying angular velocity. Demonstrates use of a time-dependent holonomic constraint. (10 Variables; 6 Constraints.)
- **Nested Rolling Toruses**: One torus rolls along the inner circumference of a second torus, which rolls along the inner circumference of a third torus. Demonstrates handling of nonholonomic constraints. (8 Variables; 6 Constraints.)
- **Weighted Ball Rolling on Plane**: Ball with off-center weight rolls and spins erratically across a plane. Demonstrates rotation in three dimensions. (6 Variables; 3 Constraints.)
- **Two Weighted Balls Rolling on Plane**: Two non-interacting copies of the system described above. Demonstrates decomposition of integration and systems of nonlinear algebraic equations. (12 Variables; 6 Constraints.)
- **Torus Rolling on Plane**: Tilted torus rolls and spins erratically on plane. Demonstrates use of a dummy noncausal object to track a point of contact between two surfaces. (11 Variables; 8 Constraints.)
- **Ellipsoid Rolling on a Plane**: An ellipsoid rolls and spins erratically on a plane. Demonstrates use of a dummy noncausal object to track a point of contact between two surfaces. (9 Variables; 6 Constraints.)
- **Ball Rolling Inside Torus**: A ball rolls on the interior surface of a torus. Demonstrates rolling contact between two curved implicit surfaces. (11 Variables; 8 Constraints.)
- **Ball Rolling Inside Rotating Ellipsoidal Drum**: A ball rolls on the interior surface of a rotating ellipsoidal drum. Demonstrates rolling contact between two moving implicit surfaces. (13 Variables; 10 Constraints.)
- **Roller Coaster (One Car):** A simpler version of the example described in this paper. Demonstrates our variable and constraint classification algorithm. (9 Variables; 8 Constraints.)
- **Sliding Coaster (Five Cars):** Similar to the example discussed in this paper, but with more cars and no wheels. (25 Variables; 24 Constraints).
- **Roller Coaster (Three Cars):** The example described in this paper. Demonstrates a variety of features of our program synthesis system. (33 Variables; 32 Constraints.)

Figure 15. Summary of Experimental Results

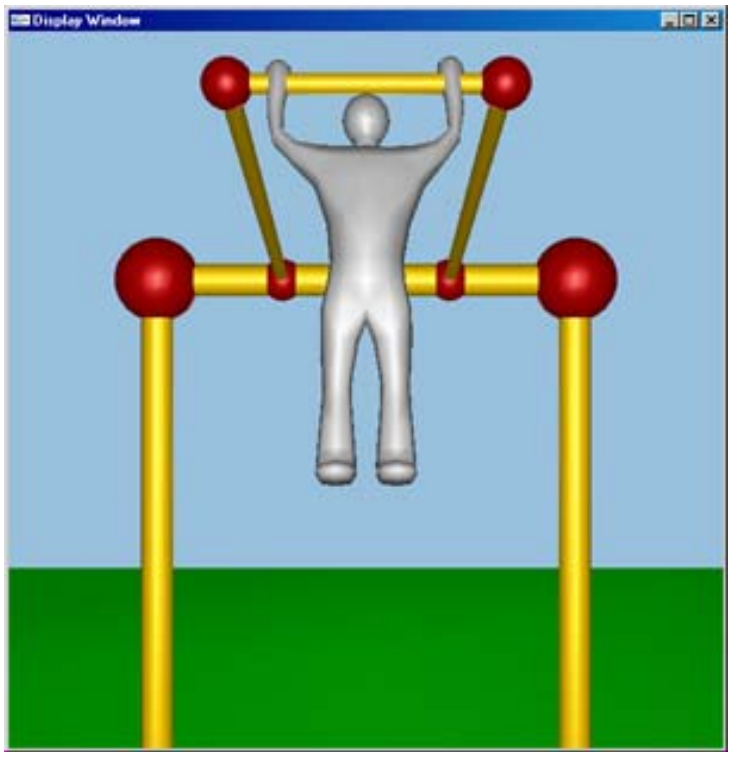

Figure 16. Acrobat on Trapeze

Figure 17. Dancing Snowman

The reader may come away with the impression that the examples described in Figure 15 are mere exercises in basic mechanics. This is partly true; however, our most dramatic programs result from exploiting the manner in which the dynamical variables are linked to the visual aspects of a physical scenario. Recall that the developer may attach arbitrary visual objects (surfaces, lights and cameras) to the leaves of the coordinate system hierarchy defined in the graphical interface. As the simulation unfolds over time, the positions, sizes and orientations of these objects may change as well. The user may view the scene from the point of view of any of the cameras. Some of the resulting effects are illustrated in Figures 16 and 17. With the right choice of geometry, lights and cameras, the double pendulum becomes an "Acrobat on Trapeze", and the ball with off-center weight rolling erratically on a plane becomes a "Dancing Snowman".

7. Related Work

The first author's previous work developed deductive methods of synthesizing numerical simulation programs for specifications composed of algebraic and differential equations, with a focus on engineering design applications [Ellman and Murata, 1998]. In that work, the algebraic and differential equations were fairly simple. The complexity of program synthesis resulted from the variety of logical forms of specifications and the corresponding variety of program architectures for combining numerical codes for integration of differential equations and finding of roots of algebraic equations. In the present work, focusing on synthesis of simulators for rigid-body systems, the architecture for combining numerical codes is relatively fixed. On the other hand, the algebraic and differential equations themselves are much more complex. Furthermore, the synthesis process requires the use of specialized symbolic computation techniques for solving algebraic equations, differentiating constraints and simplifying expressions, which are more efficiently carried out by specialized algorithms, as implemented in a system like Mathematica, rather than a general theorem-proving mechanism. Finally, in the present work, we have addressed the problem of finding an efficient implementation, rather than a merely correct implementation. Standard methods of deductive synthesis are not capable of distinguishing between two correct implementations of differing efficiency. These differences motivated our effort to develop the specialized methods of manipulating the equations and assigning them to components of the simulator architecture, as described above.

The AutoBayes and AutoFilter systems overcome limitations of deductive synthesis using a strategy similar to ours, i.e., they sacrifice generality and guaranteed correctness to obtain a tractable synthesis problem. AutoBayes generates data analysis programs from declarative descriptions of problem variables and probability distributions [Gray et al., 2003], [Fischer and Schumann, 2003]. It uses schema-guided deductive synthesis, augmented by symbolic-algebraic computation techniques. AutoFilter synthesizes programs for state

estimation problems [Rosu and Whittle, 2002]. It generates programs by recursive instantiation of program schemata. AutoBayes and AutoFilter differ from our system in the number and variety of program schemata and the instantiation mechanism they use in the synthesis process. They select from a variety of program schemata. Our system always instantiates the same program scheme. They use pattern-matching and constraint-checking to find appropriate instantiation parameters. Our system uses a specialized algorithm (the variable and constraint classification procedure) to construct parameters instantiating the simulation program scheme. AutoBayes and AutoFilter also construct proofs that certify key properties of synthesized programs [Schumann et al., 2003].

An alternative approach to overcoming limitations of deductive synthesis is found in the Amphion and Meta-Amphion systems. Amphion is a deductive system that synthesizes programs utilizing libraries of astronomical software [Lowry et al., 1994]. Meta-Amphion is a program synthesis system that generates other, domain-specific, program synthesis systems [Lowry and Van Baalen, 1997]. The domain specific synthesis systems incorporate specialized decision procedures into formal deductive methods of program synthesis [Roach and Van Baalen, 2002]. In our work, specialized symbolic algebra procedures are an important component of our system's capabilities. It would be interesting to investigate whether a system like Meta-Amphion could synthesize a domain-specific program synthesis system for constructing rigid-body simulation programs, such as the one we have presented here.

A number of other investigators have also developed automated program synthesis techniques for scientific and numerical computation. The SciNapse, Agnes and Ctadel systems each synthesize programs for solving certain types of partial differential equations (PDEs). SciNapse constructs finite element codes, using a knowledge base of transformation rules implemented in Mathematica [Kant, 1993], [Akers et al., 1998]. Agnes constructs numerical codes by matching input equations to templates, in order to choose an appropriate solution method [Kowalski and Peskin, 1990]. Ctadel generates code that runs on sequential, vector and shared virtual and distributed memory architectures [Van Engelen et al., 1997]. These systems handle specifications involving *partial* differential equations (involving more than one independent variable) with no constraints. In contrast, our system handles specifications involving ordinary differential equations (involving a single independent variable) as well as constraints.

Methods of automating the synthesis of planning and scheduling programs are reported in [Srivastava and Kambhampati, 1998] and [Blaine et al., 1998]. These techniques operate in part by assigning each problem constraint to be enforced in an appropriate part of the program being generated. This is similar to our approach of classifying dynamical variables and constraints and assigning each class to be implemented in a different

program component. Nevertheless, despite this surface similarity, the applications are so different (symbolic search versus numerical simulation) that the respective program synthesis techniques do not appear to be transferable from either type of application to the other.

Simulink® is a commercial system for modeling, simulating, and analyzing dynamical systems. It presents a user with an interface through which he/she constructs a dataflow block diagram describing a dynamical system, and translates the diagram into an executable MATLAB® program. The system does not appear to be capable of handling constrained systems of rigid bodies, which are governed by combinations of differential and algebraic equations, except perhaps by allowing the user to call upon the general programming facilities of MATLAB. Havok® is a commercial physics-based animation toolkit. It provides a limited set of tools for constructing programs that simulate constrained systems of rigid bodies, along with specialized facilities for generating programs to simulate motor vehicles and humanoid characters. It's rigid-body facilities support only "point-to-point" constraints, a specialized type of holonomic constraint. It does not appear to support systems governed by arbitrary holonomic or non-holonomic constraints or arbitrary force and potential functions.

8. Plans for Future Work

We are currently extending our work in several ways. One extension is aimed at handling rigid-body systems in which the constraints change over time, in response to collisions, user commands or other events. Systems of this type can be modeled as hybrid automata, i.e., combinations differential equations and finite state machines [Van Der Schaft and Schumacher, 2000]. A key problem involves specifying and synthesizing transitions in which the system switches from one mode of operation to another and values of dynamical variables change instantaneously. Our approach is to define a family of parameterized specification schemata, and to associate each specification scheme with a specialized program synthesis procedure [Ellman, 2003]. We also plan to extend this work to handle systems whose behavior can be specified in terms of optimal control strategies based on the Pontryagin maximum principle [Hartl, et al., 1995]. In the presence of unilateral state constraints, optimal controls often exhibit a mode switching behavior that can be modeled by a hybrid automaton. Finally, we plan extend our system to handle articulated figures engaged in walking, running, jumping and similar actions. Systems of this sort have many more dynamical variables than the ones we have investigated to date. It would therefore be worthwhile to investigate problems of scale that may arise in specifying and synthesizing programs that animate the behavior of such complex articulated figures.

9. Summary of Contributions

Our research is a contribution to the field of Automated Software Engineering in several different respects. To begin with, real-time 3D animation is becoming a progressively more important part of the software industry. It is now commonly used in computer and video games, as well as in many educational, scientific and engineering applications. As network and processor speeds rise, we expect real-time 3D animation to commonly appear in other contexts, such as web pages and user interfaces to application programs. We have automated the synthesis of an important class of real-time 3D animation programs, i.e., those involving constrained systems of rigid bodies. Our research is therefore a contribution toward automating a portion of the software engineering problem, the importance of which will grow in the coming years. In addition, our classification-based approach to program synthesis may be applicable to other kinds of software, in which equations and constraints are assigned to be handled in various components of a relatively fixed program architecture. For example, we believe it would be useful in the context of engineering design problems that use a software architecture combining numerical simulation and optimization codes, as described in [Ellman et al., 1998]. Finally, our approach illustrates a methodology that exploits the tradeoff between the power and the generality of a program synthesis system. Our system is a compromise between formal deductive methods of program synthesis and conventional program generators. We have sacrificed the generality and correctness guarantee of deductive synthesis. In return, we have obtained a system in which the computational costs of synthesis are more in line with conventional program generators. Furthermore, our system uses a declarative specification language and a declarative knowledge base, which may provide more maintainability and transparency than are typically seen in conventional program generators. We expect that approaches to program synthesis manifesting this sort of compromise would be useful in other areas of Automated Software Engineering as well.

Acknowledgements

The research reported in this paper was supported by Vassar College through a faculty research grant and the Undergraduate Research Summer Institute. The anonymous referees of this paper provided thoughtful comments that the authors found quite helpful.

References

[Akers et al., 1998] R. Akers, P. Baffes, E. Kant, C. Randall, S. Steinberg and R. Young, “Automatic Synthesis of Numerical Codes for Solving Partial Differential Equations”, Mathematics and Computers in Simulation, 45, 1998.

[Baruh, 1999] H. Baruh, "Analytical Dynamics", WCB/McGraw-Hill, 1999.

[Blaine et al., 1998] L. Blaine, L. Gilham, J. Liu, D. Smith, and S. Westfold, "Planware -- Domain-Specific Synthesis of High-Performance Schedulers", Proceedings of the Thirteenth Automated Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[Ellman and Murata, 1998] T. Ellman and T. Murata, "Deductive Synthesis of Numerical Simulation Programs from Networks of Algebraic and Ordinary Differential Equations", Automated Software Engineering, 5, 3, 1998.

[Ellman et al., 1998] T. Ellman, J. Keane, A. Banerjee and G. Armhold, "A Transformation System for Interactive Reformulation of Design Optimization Strategies", Research in Engineering Design, 10, 1, 1998.

[Ellman et al, 2002] T. Ellman, R. Deak and J. Fotinatos, “Knowledge-Based Synthesis of Numerical Simulation Programs for Rigid-Body Systems in Physics-Based Animation”, Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, Edinburgh, UK.

[Ellman, 2003] T. Ellman, “Specification and Synthesis of Hybrid Automata for Physics-Based Animation”, Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003, Montreal, Canada. In Press.

[Fischer and Schumann, 2003] B. Fischer and J. Schumann, "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models", Journal of Functional Programming, 2003, in press.

[Gray et al., 2003] A. Gray, B. Fischer, J. Schumann and W. Buntine, "Deriving Statistical Algorithms Automatically: The EM Family and Beyond", Proceedings of the Conference on Neural Information Processing Systems (NIPS2002), 2003. In Press.

[Hartl, et al., 1995] "A Survey of the Maximum Principles for Optimal Control Problems with State Constraints", SIAM Review 37, 2, 1995.

[Haug, 1989] E. Haug, "Computer-Aided Kinematics and Dynamics of Mechanical Systems", Allyn & Bacon, Boston, MA, 1989.

[Kant, 1993] E. Kant, "Synthesis of Mathematical Modeling Software", IEEE Software, 10, 3, 1993.

[Keller et al., 1994] R. Keller, M. Rimon, and A. Das, "A Knowledge-Based Prototyping Environment for Construction of Scientific Modeling Software", Automated Software Engineering, 1, 1, 1994.

[Kowalski and Peskin, 1990] A. Kowalski and R. Peskin, "Anatomy of AGNES: An Automatic Generator of Numerical Equation Solutions", in E. N. Houstis, Ed., Intelligent Mathematical Software Systems, Elsevier Science Publishers, New York, NY, 1990.

[Lowry et al., 1994] M. Lowry, A. Philpot, T. Pressberger, and I. Underwood, "A Formal Approach to Domain-Oriented Software Design Environments", Proceedings of the Ninth Knowledge-Based Software Engineering Conference, Monterey, CA, 1994

[Lowry and Van Baalen, 1997] M. Lowry and J. Van Baalen, "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems", Automated Software Engineering, 4, 2, 1997.

[Manna and Waldinger, 1992] Z. Manna and R. Waldinger, “Fundamentals of Deductive Program Synthesis”, IEEE Transactions on Software Engineering, August, 1992.

[Pissanetsky, 1984] S. Pissanetsky, "Sparse Matrix Technology", Academic Press, 1984.

[Press et al., 1986] W. Press, W. Vetterling, S. Teukolsky, and B. Flannery, "Numerical Recipes", Cambridge University Press, New York, NY, 1986.

[Roach and Van Baalen, 2002] S. Roach and J. Van Baalen, "Experience Report on Automated Procedure Construction for Deductive Synthesis", Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, Edinburgh, UK.

[Rosu and Whittle, 2002] G. Rosu and J. Whittle, “Towards Certifying Domain Specific Properties of Synthesized Code”, Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, Edinburgh, UK.

[Schumann et al., 2003] J. Schumann, B. Fischer, M. Whalen and J. Whittle, “Certification Support for Automatically Generated Programs”, Proceedings of the 36th Hawaii International Conference on System Sciences, 2003.

[Srivastava and Kambhampati, 1998] B. Srivastava and S. Kambhampati, "Synthesizing Customized Planners from Specifications", Journal of Artificial Intelligence Research, 8, 1998.

[Van Engelen et al., 1997] R. Van Engelen, L. Wolters, and G. Cats, "Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling", IEEE Computational Science & Engineering, 4, 3, 1997.

[Van Der Schaft and Schumacher, 2000], A. Van Der Schaft and H. Schumacher, “An Introduction to Hybrid Dynamical Systems”, Lecture Notes in Control and Information Sciences, 251, Springer-Verlag, 2000.