Mathematical Equations as Executable Models of Mechanical Systems[*]

Yun Zhu[†] , Edwin Westbrook, Jun Inoue,
Alexandre Chapoutot[‡] , Cherif Salama,
Marisa Peralta, Travis Martin,
Walid Taha, Marcia O'Malley,
Robert Cartwright
Rice University
taha@rice.edu

Aaron Ames, Raktim Bhattacharya
Texas A&M
aames@tamu.edu,
raktim@aero.tamu.edu

ABSTRACT

Cyber-physical systems comprise digital components that directly interact with a physical environment. Specifying the behavior *desired* of such systems requires analytical modeling of physical phenomena. Similarly, testing them requires simulation of continuous systems. While numerous tools support later stages of developing simulation codes, there is still a large gap between analytical modeling and building running simulators. This gap significantly impedes the ability of scientists and engineers to develop novel cyber-physical systems.

We propose bridging this gap by automating the mapping from analytical models to simulation codes. Focusing on mechanical systems as an important class of physical systems, we study the form of analytical models that arise in this domain, along with the process by which domain experts map them to executable codes. We show that the key steps needed to automate this mapping are 1) a light-weight analysis to partially direct equations, 2) a binding-time analysis, and 3) symbolic differentiation. In addition to producing a prototype modeling environment, we highlight some limitations in the state of the art in tool support of simulation, and suggest ways in which some of these limitations could be overcome.

Categories and Subject Descriptors

D.1.2 [**Software**]: Programming Techniques—*Automatic Programming*; J.2 [**Computer Applications**]: Physical Sciences and Engineering

[*]Funding for this work was provided by generous support from Schlumberger, LogicBlox, and the National Science Foundation (NSF) Awards #0747431, #0439017, and #0720857.

[†]Now at Schlumberger

[‡]While visiting from Université Pierre et Marie Curie

1. INTRODUCTION

Systems where digital components are intimately coupled with their physical environment — often called cyber-physical systems (CPS) – pose a challenge when we wish to specify their desired behavior. For purely digital computing systems, the programming language or hardware description language itself can suffice. Often, the implementation language itself can express the desired behavior using *assertions.* Such assertions can then be used as dynamic checks, used by test-case generation tools, or used by formal verification tools. In contrast, for CPS applications we typically want to specify the desired behavior for a computational component in terms of its action on the physical environment. For example, we may wish that for a particular robot controller the torque on one of the robot's joints never exceed a certain value. However, part of specifying "the robot's joint" is to specify "the robot". Rigorously specifying the latter, and in particular, its continuous mechanics, is most naturally and accurately expressed using the mathematical formalisms of analytical dynamics (c.f. [1]). Specifying a robot's dynamics is neither naturally nor accurately specified in the discrete-step language used to write the controller.

It is tempting, given that simulation of continuous systems is a well-established discipline [8], to assume that there is a tool for directly expressing the analytical dynamics of a physical system, and which can be automatically used to simulate the system. This does not seem to be the case. This absence is surprising, given the abundance of tools that seem relevant to this process, and suggests a need for closer scrutiny of the process for going from models to simulation codes.

1.1 The Model/Simulator Gap

As a starting point for exploring the gap between models and simulation codes, we focus on mechanical systems. While mechanics is only one feature of a physical environment, it is expressive enough to capture important continuous aspects of vehicles, robots, linear circuits, molecular structures, nano structures, and others. At the same time, even though mechanical systems give rise only to ordinary differential equations, any challenges that this domain reveals will also be present with more sophisticated domains such as fluid dynamics, which give rise to partial differential equations.

Without computational tools, mechanical and control engineers transform models of mechanical systems to simulation codes through a series of laborious, manual mathematical manipulations, which result in mathematical equations in a "constructive enough form" that they are easy to implement as a simulation program.

When we consider mechanical tools, we find that there is a wide range of varieties with different focus and design philosophies. The sheer diversity of such tools poses a practical problem for all practitioners working on CPS innovations. More fundamentally, there is a number of reasons why most tools available today do not address the model/simulator gap directly.

One reason is that many widely used tools do not attempt to capture models directly. For example, scripting or programming languages such as MATLAB [29], R [26], or Biopython [4] focus on providing a more convenient *programming* language for a broad class of scientific problems. They help programmers (or scientists with programming expertise and inclination) develop simulation codes, and are not really intended to capture *models* as they are most naturally expressed by domain experts. With such programming languages, assuming that underlying analytical models assumptions were ever explicitly expressed somewhere in the code, they typically get buried and mangled with numerous implementation concerns.

Another reason is that tools often are incapable of capturing analytical models naturally. For example, with the above tools (and their graphical extensions such as Simulink), even if the user is careful enough to keep the code aligned with some underlying analytical model, the nature of these programming languages limits them to expressing so-called *causal models*, which are models that are not expressed as equations, but rather, in a more restrictive, directed form.

Tools that focus on high-performance typically require special expertise to be used effectively, and are generally designed to be used as a back-end for higher-level tools. Examples of such tools include solvers such as LINPACK [9] and PETSc [3].

Tools that facilitate capturing physical design often hide the underlying analytical models. For example, AutoCAD [2], Pro/ENGINEER [25], and SolidWorks [28] all provide excellent graphical interfaces for capturing geometric design. In many cases, such tools also provide a simulation capability, and lend themselves naturally for helping the user visual the results of a simulation (at least in the same geometric terms as entry was done). However, the fact that the underlying analytical models (as well as the treatment of various modalities that may arise in a simulation) are generally hidden, limit the fidelity of their simulations. In this respect, these tools are not unlike real-time physics simulation engines [5], which are very useful for graphics, animation, and gaming applications, but do not presume to be faithful to an explicit analytical model.

Like programming languages, symbolic algebra tools are often too generic. While symbolic algebra tools such as Mathematica [32] and Maple [21] can, in principle, assist in manipulating analytical models and mapping them to code, in practice, they are not always well suited for this task. While these systems can provide a powerful tool for assisting in analytical modeling, they often suffer from subtle but highly significant problems. Guiding such tools to perform the particular desired transformations can be challenging, and their effective use can require intimate familiarity with the tool, its built-in routines and libraries, as well as using care to avoid divergent rewriting and exponential increase in the size of the model. Even simple examples of these difficulties can be hard to get unstuck from. A specific example of such difficulty arose in our work. As the size of the physical system being modeled increases, symbolic computing systems can *both* take exponentially large time to compute a symbolic result *and* produce a result that is unnecessarily exponentially large. We encountered such a problem in the context of modeling a novel bipedal natural gate robot. Trying to symbolically differentiate an 8 by 8 Lagrangian coefficient using one of the leading symbolic solvers, the result was a file that was 13MB large. The level of redundancy in the generated term made it impractical as a basis for a simulation code. More significantly, the underlying combinatoric explosion made modeling more sophisticated systems impossible until this problem was addressed. The problems could be avoided if symbolic algebra tools incorporate any of decades worth of improvements on symbolic differentiation which generally go under the name of *automatic differentiation* (c.f. [17]).

Thus, even for the simplest physical domains, multiple tools are needed to go from analytical modeling to simulation. For example, often symbolic computing tools are used to assist in the analytical modeling of mechanical systems, and the resulting computations need to be mapped to the particular platform that will be used for carrying out the resulting computation. Carrying out such mapping by hand is error prone. Using an automatic translation would be preferable, but this requires either finding a translator or writing one. Often, because of the numerous semantic subtleties between languages and the differences in the libraries used in each language, attaining an operational and a correct translation itself becomes a significant investment. Worse, in many cases, the internal languages for the most successful tools are not readily available. Even when standards exist for interchange of information, such as is the case for CAD tools, practical experience shows that correct data interchange between such tools is still difficult. This is not too surprising, because the semantics of the mediating formats are typically defined only by the implementations, which themselves were organically grown without a blueprint for integration.

1.2 Contributions

To bridge the gap between analytical models and running simulation codes, we show the feasibility of using a subset of the mathematics used for analytical modeling as a domain-specific language (DSL) for building simulation codes. In doing so we converge on a design similar to a promising class of modeling tools, namely equation-oriented languages. The leading example of such languages is Modelica (c.f. [12]). Although Modelica is a highly-developed, industrial strength language, the limited expressivity of its core language can be noticed when we attempt to analytically model even small mechanical systems. Two examples of such limitations are binding-time separation and support for partial derivatives.

The first half of the paper introduces the syntax of a core analytical modeling language. Basic and fairly self-evident

```
(* pendulum-with-controller.acumen *)

discrete efrp (* A simple E-FRP controller *)
 reads     theta;
 writes    F;
 observes  event clock rate t = 0.1;
 begin
  last = init 0 in { clock => theta later};
  F = init 0 in
      { clock =>
        if abs(theta) < pi/100 then 0
        else if theta > last then -1.0
        else 1.0 };
 end

continuous (* Pendulum physics *)
 m = 5.0; g = 9.81; l = 3;
 I = m * l^2;
 F*l*cos(theta) - m*g*l*sin(theta) = I*theta'';

 boundary conditions
  theta with theta(0)= 0.1, theta'(0) = 0;
```

Figure 1: Acumen Model of a Controlled Pendulum

features of an analytical modeling language include equations, point-free (or implicit time) notation, as well as time derivatives (Section 2). More sophisticated – and in some cases less-self evident – features needed to model even basic mechanical systems include partial derivates, families of equations, aggregates (such as sequences and matrices), and recursion (Section 3). The need for each of these features as well as the complexities that some of them introduce are explained.

The second half of the paper shows how the language defined in the first half is mapped to executable code (Section 4). Our study reveals that it is possible to provide a practical mapping for a highly expressive language using only 1) a light-weight analysis to partially direct equations, 2) a binding-time analysis, and 3) symbolic differentiation. By implementing this mapping, we are able to accurately explain why current symbolic algebra tools are not ideally suited to assist in building simulation codes for novel robotic systems (Section 5). In particular, we find the naive implementations of symbolic differentiation in mainstream symbolic algebra tools such as Mathematica and Maple to be a significant performance bottleneck for mapping larger analytical models to executable codes.

2. A SIMPLE MODEL

As noted in the introduction, the motivation for this work stems from interest in CPS systems. Therefore, while our focus is primarily on analytical models for continuous systems, it is important also to explain how such models are integrated with models for the digital components. To this end, this section uses a simple example to illustrate how both discrete and continuous models are integrated, as well as the basics of analytical modeling of continuous systems. To help ground these concepts, we will also explain how our implementation (called Acumen) supports both types of modeling and their integration.

2.1 Discrete Modeling Language

Components described in Acumen are assumed to exist in a time-varying universe, where there is a global notion of real-valued time. Within this universe, both discrete and continuous components can be defined.

Figure 1 presents the Acumen ASCII source code for a discrete controller attached to a continuous model of the simple pendulum shown in Figure 2. The `discrete` section in Figure 1 introduces an event-driven component. The qualifier `efrp` specifies that what follows is a description of a controller described in the E-FRP formalism [20, 31]. The key features of E-FRP are providing support for specifying event-driven reactive systems, interfaces across which values are read from or written to the environment, and periodic or observation-driven events. Acumen and its semantics are not tied to this particular formalism. Rather, any formalism that can satisfy the synchrony hypothesis can be used for specifying discrete computations.

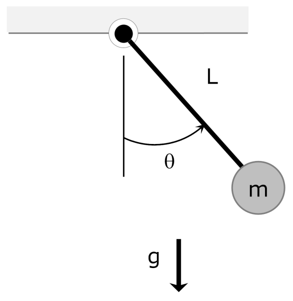


Figure 2: Pendulum

The controller described in the figure reads the quantized value of `theta`, and upon completing its computation, writes a quantized value to `F` and `last`. Outside the component these variables have continuous values, but they can only be read and written to in a quantized form. Within a discrete component, by default, quantized values are assumed to be represented as standard floating-point numbers. The `observes` clause defines the event `clock` that is automatically generated at a constant rate of `0.1` time steps. This event will be used to trigger activations of this agent. The rest of the section specifies the values written for `F` and `last` as two event-driven equations. The equations in this example are simple because only one event (`clock`) is used as a possible trigger. In general, any number of events can be used. The first equation declares `last` to be initially `0`, and then to carry the value of `theta` from then onwards. The annotation `later` means that the other equation always reads the old value of `last` when it is triggered. This way, `last` is used to always carry the last value of `theta`. The next equation defines a simple discrete controller that applies a lateral force of `1` on the pendulum in a direction opposing motion. A threshold of `pi/100` degrees is used to inhibit any force to the pendulum when it is almost vertical.

2.2 Continuous Modeling Language

The `continuous` section in Figure 1 describes the continuous environment through as a series of equalities that constrain a set of real-valued, time-varying variables, and where the notion of time itself is real-valued. In this example, our physical environment is a simple pendulum, and the description simply introduces some constants and the rotational analog of Newton's second law of motion ($\sum f = ma$) for a pendulum. Acumen's equations are not directed. This means that writing the equation as $a = F/m$ would have been equivalent. Thus, the constraints are relational or *acausal*. Equations can refer to derivatives of variables. For example, `theta''` refers to the second derivative of the variable `theta`. Finally, the `boundary conditions` subsection allows us to provide the information necessary to solve the system of equations.

2.3 Other Features of the Implementation

In addition to allowing the user to describe both continuous and discrete components, Acumen also provides a module system that allows components to be packaged in a reusable fashion. Module instantiation is a fairly straight forward process, and is orthogonal to the issues addressed in this paper.

Acumen supports two methods for executing models. For purely discrete (E-FRP) components, Acumen provides support for real-time execution [20, 31]. For hybrid models, Acumen supports discretized simulation of the whole system by compiling the continuous models using the methods that we present in the rest of this paper. For this method of execution, the user declares simulation parameters as part of the source program in a `simulation` section. The user can specify what values to observe during a simulation and how they should be printed in a `discrete log` section.

Acumen uses an ASCII-based syntax designed to be as close as possible to mathematical notation. However, because it is often more convenient to read the code type set in the same way that mathematical formulae appear in scientific publications and texts, Acumen also provides a pretty printer that provides this functionality. This pretty printer produces a LaTeX file typesetting the model after each invocation of the Acumen compiler. The generated LaTeX file contains a type set version of the source program as well as the intermediate forms that arise during the compilation process. Several examples of the output of this pretty printer will be presented in this paper. Only minor manual modifications were needed to adapt these outputs to fit the needs of the exposition.

3. MORE SOPHISTICATED MODELS

Mechanical engineers and researchers working on novel cyber-physical systems use a wide range of analytical modeling methods [1]. The most basic method for modeling mechanical systems is to write a set of equations based on Newton's laws of motion. A somewhat more sophisticated method, often considered more elegant and more systematic, is to write the so called Lagrange equations. Because this approach is centered around formalizing kinetic and potential energies, it can also have the advantage of being useful for specifying and verifying stability properties of a given system. Yet another method can be used once the Lagrange equations have been attained, which is to write the so called Hamilton's equations. For a variety of technical reasons, researchers working on novel robotic systems tend to make extensive use of both Lagrangian and Hamiltonian methods.

As noted earlier, in current practice, going from an analytical model for a novel mechanical system to simulation code is generally done by hand. The process typically includes steps such as:

- Mapping generic problem solving techniques like Lagrangian or Hamiltonian modeling to specific instances
- Eliminating partial derivatives, often using symbolic differentiation
- Algebraic manipulation to solve for unknown variables, including Gaussian elimination
- Discretization

There are steps that engineering students learn as part of their training, and are generally able to perform quite skillfully for small, model problems. But for virtually any realistic problem, the size of the systems involved makes these steps tedious and error prone. These difficulties are particularly problematic when the system being developed is novel. In such situations, the structure of the system is constantly changing and the expected behavior is not yet well understood.

Acumen's continuous modeling language is designed to address this need, and has been developed as a collaboration between computer scientists and scientists actively developing novel cyber-physical systems. In the rest of this section we introduce the key mathematical concepts supported by Acumen and illustrate their role in modeling mechanical systems.

3.1 Partial Derivatives

The three basic methods for modeling mechanical systems described above can be illustrated by using them to model a simple pendulum. Figure 3(a) presents the automatically pretty-printed version of the Acumen code we presented in the previous section. As noted earlier, the main equation in this model simply uses Newton's second law of motion.

Figure 3(b) presents a model that uses Lagrange's equation. When using this method, one specifies the kinetic energy T and the potential energy V in the system. Then, the Lagrangian L is always taken to be $L = T - V$. For a system that has only one state variable such as θ in this system, the Lagrange equation is simply the final equation in that display. Shortly, a more sophisticated example will illustrate what is done when there are multiple state variables, and how Acumen supports modeling such systems. Nevertheless, this simple example is sufficient to illustrate the utility of partial derivatives (such as $\partial L / \partial \dot{\theta}$ in this example) in applying systematic methods such as Lagrangian modeling. Figure 3(c) similarly illustrates the need for partial derivatives to model systems using Hamilton's equations.

3.2 Families of Equations

Once the system being described has more than one state variable, modeling using Lagrange or Hamilton equations employs *families of equations*, which are written as one equation but really represent a collection of different equations derived by instantiating certain indices and performing some affine (or "small") computations. For example, Figure 3(d) provides a model for the system shown in Figure 4. It consists of a pendulum hanging from a mass, and where the

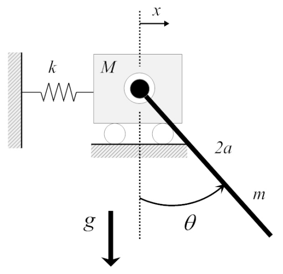


Figure 4: A Pendulum-Spring-Mass system

$$m = 5 \quad g = 9.81 \quad \ell = 3 \quad I = m\ell^2$$
$$F\ell\cos(\theta) - mg\ell\sin(\theta) = I\ddot{\theta}$$

(a) Newtonian Formulation of a Pendulum

$$m = 5 \quad g = 9.81 \quad \ell = 3 \quad I = m\ell^2$$
$$T = \frac{1}{2}I\dot{\theta}^2 \quad V = mg\ell\left(1 - \cos(\theta)\right)$$
$$L = T - V \qquad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = 0$$

(b) Lagrangian Formulation of a Pendulum

$$\mathrm{m} = 5 \quad \mathrm{g} = \frac{981}{100} \quad \ell = 3$$
$$\mathrm{H} = \frac{p^2}{2\ell\mathrm{m}} - \mathrm{g}\ell\mathrm{m}\cos(\theta) \quad \dot{\theta} = \frac{\partial H}{\partial p} \quad \dot{p} = -\frac{\partial H}{\partial \theta}$$

(c) Hamiltonian Formulation of a Pendulum

$$q = [x, \theta] \quad a = 1 \quad m = 2 \quad M = 5$$
$$g = \frac{49}{5} \quad k = 2 \quad I = \frac{4}{3}ma^2$$
$$T = \frac{1}{2}(M + m)\dot{x}^2 + ma\dot{x}\dot{\theta}\cos(\theta) + \frac{2}{3}ma^2\dot{\theta}^2$$
$$V = \frac{1}{2}kx^2 + mga\left(1 - \cos(\theta)\right) \quad L = T - V$$
$$\forall i \in \dim(q) \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0$$

(d) Pendulum/Mass

$$q = [\theta_{ns}, \theta_s] \quad d = \begin{bmatrix}\dot{\theta}_{ns}\\ \dot{\theta}_s\end{bmatrix} \quad m = 1 \quad a = 2$$
$$b = 3 \quad \ell = a + b \quad m_H = 10 \quad g = 9.81$$
$$M = \begin{bmatrix} mb^2 & -m\ell b\cos(\theta_s - \theta_{ns}) \\ -m\ell b\cos(\theta_s - \theta_{ns}) & (m_H + m)\,\ell^2 + ma^2 \end{bmatrix}$$
$$T = \frac{1}{2}\left(d^{\mathrm{T}}Md\right)_{(0,0)} \quad V = m_H g\ell\cos(\theta_s) + mga\cos(\theta_s)$$
$$+\, mg\left(\ell\cos(\theta_s) - b\cos(\theta_{ns})\right) \quad L = T - V \ \ldots$$

(e) 2D Bipedal Robot

$$\mathrm{P}(\ell, m) = \begin{cases} 1 & \text{if } \ell = m \text{ and } \ell = 0 \\ (1 - 2m)\,\mathrm{P}(m-1, m-1) & \text{if } \ell = m \\ (1 + 2m)\,z\,\mathrm{P}(m, m) & \text{if } \ell = m + 1 \\ \frac{(2\ell-1)z\,\mathrm{P}(\ell-1,m) - (\ell+m-1)\,\mathrm{P}(\ell-2,m)}{\ell-m} & \text{otherwise} \end{cases}$$
$$\mathrm{S}(m) = \begin{cases} 0 & \text{if } m = 0 \\ x\,\mathrm{C}(m-1) - y\,\mathrm{S}(m-1) & \text{otherwise} \end{cases}$$
$$\mathrm{C}(m) = \begin{cases} 1 & \text{if } \quad m = 0 \\ x\,\mathrm{S}(m-1) - y\,\mathrm{C}(m-1) & \text{otherwise} \end{cases}$$
$$\mathrm{fact}(n) = \begin{cases} 1 & \text{if } \quad n = 0 \\ n\,\mathrm{fact}(n-1) & \text{otherwise} \end{cases}$$
$$\mathrm{N}(\ell, m) = \begin{cases} (2\ell + 1/4\pi)^{1/2} & \text{if } \quad m = 0 \\ \left((2\ell + 1/2\pi)\,\frac{\mathrm{fact}(\ell-m)}{\mathrm{fact}(\ell+m)}\right)^{1/2} & \text{otherwise} \end{cases}$$
$$\mathrm{Y}(\ell, m) = \begin{cases} \mathrm{N}(\ell, -m)\,\mathrm{P}(\ell, -m)\,\mathrm{S}(-m) & \text{if } \quad m < 0 \\ \mathrm{N}(\ell, m)\,\mathrm{P}(\ell, m)\,\mathrm{C}(m) & \text{otherwise} \end{cases}$$
$$m = 5 \quad V = \mathrm{Y}(3, 2) \quad q = [x, y, z]$$
$$T = \left(\frac{1}{2}\right) m\left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2\right) \quad L = T - V \ \ldots$$

(f) Particle in a Field Defined by Spherical Harmonics

Figure 3: Automatically Typeset Examples of Continuous Systems Described in Acumen

mass is attached via a spring to a wall. Because this example has two degrees of freedom, x and θ, the example introduces a tuple of state variables q. The equations covering the dynamics of the system are then expressed by the family of equations that appears at the end of the examples. In Figure 3(d), the $\forall$ quantifier is used to introduce the index variable for a family of equations. Such quantifiers can be nested as needed. In the ASCII-based syntax, the keyword `foreach` represents this quantifier. The intent is to express as concisely and as naturally as possibly what would typically be expressed in a mechanics textbook or research paper as:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0 \text{ where } \begin{matrix} q = (x, \theta) \\ i \in \{1, 2\} \end{matrix}$$

For someone not familiar with this notation its exact interpretation can be hard to discern. In particular, if it is read compositionally, one must interpret the negated term as the derivative of L with respect to the real-number value of the ith element of the tuple q. Strictly speaking, there may well be a mathematically valid compositional interpretation for this notation along those lines. What is generally meant, however, is a more syntactic interpretation that is more akin to macro-expansion than to a compositional interpretation. In particular, what is meant is that the *name* contained in the ith element of the tuple is looked up. In other words, this family of equation literally represents the following two equations:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0 \quad \text{and} \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = 0$$

It is useful to note here that mechanical engineers do not seem to write such expressions when a complex computation is needed to look up the name. Rather, computations of this form are usually fairly simple, and one can expect that a simple analysis can be used to ensure that these types of computations are well-formed. In fact, one can argue that it would be ill-advised to use a general purpose algebraic solver to perform such computations. In particular, if there is a simple typographic error in such a formulation, a general purpose engine can spend a lot of time trying to perform an unnecessary computation.

3.3 Aggregates

Not surprisingly, once we start modeling larger systems, there is a need for aggregate types. Acumen supports basic aggregate types, namely, tuples, vectors, and matrices, as well as the standard operations on these constructs. Figure 3(e) shows how these three notions arise naturally in modeling the simple compass-like biped robot [16] shown in Figure 5. In the ASCII-based syntax, tuples are written with regular parentheses, and vectors and matrices with square brackets. While the treatment of aggregates in Acumen is mostly straightforward, their implication in descriptions of families equations (for example, q in the example above is a tuple) does require some care. We return to this point in Section 4.2.

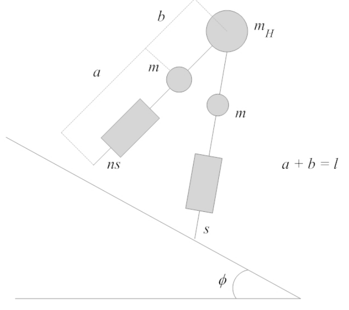


Figure 5: Bi-Ped Robot

3.4 Recursive Functions and Conditionals

Auxiliary functions are often needed to specify non-trivial mechanical systems. Furthermore, relatively sophisticated machinery, such as recursion, is generally needed to define such functions. Recursive functions are needed to write the dynamic equations that result when a series of masses are connected together with springs. While we can do this example with Acumen's built-in summation function, slightly more sophisticated (but still very common structures) such as robots with a tree-like topology (bipeds, humanoids, etc), are more naturally modeled using recursive functions that traverse the specification of the topology.

To showcase an example further a field from the rigid-body mechanics domain that we have focused on so far, and that requires several recursive auxiliary function definitions, Figure 3(f) presents a complete specification of a spherical harmonics problem in Acumen. The example uses Acumen's support for recursive functions that return both real and integer values, as well as the use of conditional expressions. Conditionals are needed, of course, to ensure that the recursive definitions have base cases. In the ASCII-based syntax, conditionals are written simply as `if-then-else` expressions.

4. EXECUTING AN ANALYTICAL MODEL

We now turn to the question of how such analytical models can be turned into executable code. To address this question in the most useful way, we chose to balance two goals. On the one hand, we are interested in automating as much as possible of the derivation and coding steps that engineers carry out by hand. On the other hand, we want to ensure that the tools that we and others develop for this purpose comprise primarily a small number of transformations that are fast, transparent, and predictable. In this section we explain three steps that are key to achieving this goal. These steps are 1) a light-weight analysis called Defined Variable

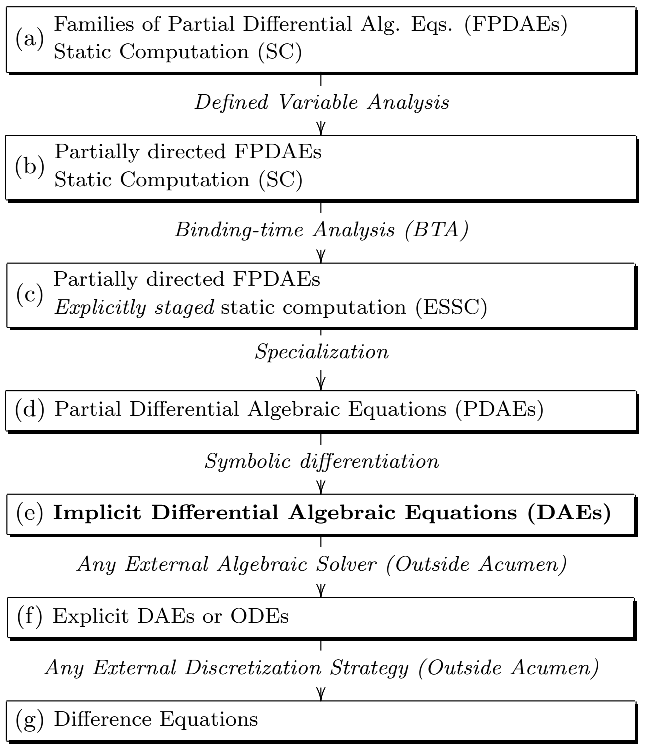

(a) Families of Partial Differential Alg. Eqs. (FPDAEs)
Static Computation (SC)
Defined Variable Analysis
(b) Partially directed FPDAEs
Static Computation (SC)
Binding-time Analysis (BTA)
(c) Partially directed FPDAEs
Explicitly staged static computation (ESSC)
Specialization
(d) Partial Differential Algebraic Equations (PDAEs)
Symbolic differentiation
(e) **Implicit Differential Algebraic Equations (DAEs)**
Any External Algebraic Solver (Outside Acumen)
(f) Explicit DAEs or ODEs
Any External Discretization Strategy (Outside Acumen)
(g) Difference Equations

Figure 6: Mapping Analytical Models to Code

Analysis that allows us to partially direct equations in the source program, 2) a Binding-time Analysis (BTA) with an associated specialization step, and 3) symbolic differentiation.

In addition to explaining these three important steps, we will also briefly describe two additional steps that are implemented in Acumen but that are *not* technically novel. These two steps are there only for practical and pedagogic reasons. Figure 6 gives an overview of all of these transformations. The transformation below the Implicit Differential Algebraic Equations (DAEs) are strictly not necessary, because fairly generic solvers exist for implicit DAEs.

4.1 Defined Variable Analysis

For the most part, most equation-solving is done outside Acumen by standard tools. However, in order to support expression families, partial derivatives, and recursive equations, a light-weight analysis is needed to determine how information flows across some of these equalities. This analysis is called Defined Variable Analysis, and works as follows. First, it is only a best-effort analysis, and does not attempt to direct all equations. If it fails to discover the direction of flow for a particular variable and it turns out that this information is needed by the two ensuing steps, the problem will be reported to the user.

An equation can be trivially directed if one side is a variable and the other side is a known value. Known values include constants, time, variables provided by a discrete controller or an external input, variables for which boundary conditions are provided, and any expression defined by a function applied to known values. Finally, a value is known when it is an explicitly provided tuple or vector of known

$$q = [x, \theta] \quad a = 1 \quad m = 2 \quad M = 5 \quad g = 9.8 \quad k = 2$$
$$I = \frac{4}{3}ma^2 \quad T = \frac{1}{2}(M+m)\dot{x}^2 + ma\dot{x}\dot{\theta}\cos(\theta) + \frac{2}{3}ma^2\dot{\theta}^2$$
$$V = \frac{1}{2}kx^2 + mga(1-\cos(\theta)) \quad L = T - V$$
$$\forall i \in \dim(q) \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0$$

(a) Acumen Source for Pendulum/Mass Example

Defined: $q := [x, \theta] \quad a := 1 \ \ldots \quad I := \frac{4}{3}ma^2 \ \ldots$

Computed: $\forall i \in \dim(q) \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0$

(b) After Defined Variable Analysis

Defined: $q := [\ x\ ,\ \theta\ ] \quad a := 1 \ \ldots \quad I := \frac{4}{3}ma^2 \ \ldots$

Computed: $\forall i \in \dim(q) \quad \frac{d}{dt}\left(\frac{\partial L}{\partial\ \dot{q}_i}\right) - \frac{\partial L}{\partial\ q_i} = 0$

(c) After Binding-Time Analysis (BTA)

Defined: $I := \frac{8}{3} \ \ldots$

Computed: $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0 \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = 0$

(d) After Specialization

Defined: $\mathsf{A} := \cos(\theta) \quad \mathsf{B} := \sin(\theta)$

Computed: $\left[\begin{array}{rcl} 2\mathsf{A}\ddot{\theta} - 2\mathsf{B}\dot{\theta}^2 + 7\ddot{x} + 2x & = & 0 \\ \frac{98}{5}\mathsf{B} + 2\mathsf{A}\ddot{x} + \frac{8}{3}\ddot{\theta} & = & 0 \end{array}\right]$

(e) After Symbolic Differentiation (Implicit DAEs)

$$\mathsf{A} := \cos(\theta) \quad \mathsf{B} := \sin(\theta) \quad \ddot{\theta} := \frac{4\mathsf{A}x - 4\mathsf{A}\mathsf{B}\dot{\theta} - \frac{686}{5}\mathsf{B}}{\frac{56}{3} - 4\mathsf{A}^2}$$
$$\ddot{x} := \frac{2}{7}\mathsf{B}\dot{\theta}^2 - \frac{2}{7}\mathsf{A}\ddot{\theta} - \frac{2}{7}x$$

(f) Example Explicit DAE form

$$\mathsf{A}^+ := \cos(\theta) \quad \mathsf{B}^+ := \sin(\theta) \quad x^+ := x + \dot{x}\Delta t$$
$$\dot{x}^+ := \dot{x} + \ddot{x}\Delta t \quad \theta^+ := \theta + \dot{\theta}\Delta t \quad \dot{\theta}^+ := \dot{\theta} + \ddot{\theta}\Delta t$$
$$\ddot{\theta}^+ := \frac{4\mathsf{A}x - 4\mathsf{A}\mathsf{B}\dot{\theta} - \frac{686}{5}\mathsf{B}}{\frac{56}{3} - 4\mathsf{A}^2} \quad \ddot{x}^+ := \frac{2}{7}\mathsf{B}\dot{\theta}^2 - \frac{2}{7}\mathsf{A}\ddot{\theta} - \frac{2}{7}x$$

(g) Example Difference Equation form

Figure 7: Compiling the Pendulum/Mass Example

size. The state variable q in our running example in Figure 7 is considered known because of this rule. An equation can also be directed when it contains exactly one unknown variable, and there is a series of simply algebraic transformations on that equation that can produce a definition for that variable in terms of the rest of the variables appearing in the equation. Any equations that do not have an obvious directionality are left as equations to be solved at the end of the transformation process.

The output of this analysis for for the Pendulum/Mass example is presented in Figure 7(b). Here, directed equalities are marked by using := instead of =.

4.2 Binding Time Analysis (BTA)

After determining which equation can be easily directed, we need to check that there is sufficient information to 1) expand all expression families, 2) evaluate all recursive functions, and 3) eliminate all partial derivatives. If any information is missing, an error is reported to the user. It turns out that this check is essentially a binding-time analysis (BTA) [19]. BTA is the analysis performed in an offline partial evaluation system to determine, given some early or "static" inputs to a program, which of the program's computation can be done at an early stage. Technically, Acumen's BTA is a hybrid between a standard BTA used in a partial evaluator and a type checker for a two-level language [23]. In particular, whereas a standard BTA makes a best-effort attempt to perform as much of a program's computations as early as possible, Acumen's BTA needs to produce a user-level error when certain goals are not met. Specifically, the source-level model must provide enough information for the three kinds of static computations listed above to be performed during the process of deriving executable code. Otherwise, the user needs to be notified that this information is missing, so that they can fix the model. For example, in a model that uses the Lagrange equation, if the user writes $\partial L/\partial q_i$, the BTA would produce an error if any information is missing that prevents Acumen from determining what the components of the state vector q are.

A successful BTA annotates the model with instructions for performing certain parts of the computation early and to keep other parts of the model computations for later processing. This annotation process is sometimes called *staging* as it separates the model into two stages. The annotated model for the pendulum/spring example is illustrated by Figure 7(c). In this illustration, computations that will remain for further processing are shaded in gray, whereas computations that should be performed immediately in the next transformation step appear with a white background.

The annotations on this example serve to illustrate a number of features of Acumen's BTA. The value of a is marked as available because it is a known value. The analysis does a little bit more work to determine that the whole definition for I is a known value. In particular, it uses the fact that the values for both m and a are known in the source. An even more interesting case is the value for q, where the tuple constructor itself is marked known but its elements are not. In the partial evaluation literature, this type of value is referred to as a *partially static data structure* [6]. Acumen's BTA supports such values to allow us to lookup the names of the individual state variables even though their values might not be known. This is enabled by the fact that the size of the tuple q as well as the names of its contents are known in the model, even though the values of these variables are not known (and in fact we will need to solve for the values of these variables during the simulation).

While the implementation details for performing BTA for the three types of computations listed at the start of this section are somewhat involved, there is an elegant semantic explanation for these details, and which is reflected in the underlying types. In particular, essentially all values belonging to the set of integers $\mathbb{N}$ or rationals $\mathbb{Q}$ are treated as static, and essentially all values belonging to the set of reals $\mathbb{R}$ are viewed as dynamic. As such, functions of type $\mathbb{N} \to \mathbb{R}$ or even of type $\mathbb{N} \times \mathbb{R} \to \mathbb{R}$ have statically known inputs, and are therefore amenable to specialization.

4.3 Specialization

As noted earlier, BTA is essentially a scheduling analysis. In the partial evaluation literature, the step which performs the work that a BTA schedules is called *specialization*. Acumen's specialization step has two non-standard features. First, because we are dealing with mathematical terms where functions do not have side effects, it is safe to memoize all functions. Second, to avoid the possibility of code duplication, all terms are named and the name is used instead of the value.

The result of specializing our running example is presented in Figure 7(d). Specialization is very similar to standard program execution, where we are allowed to have variables in place of some otherwise proper values. For example, the instantiation of expression families is essentially a type of iteration. Computing the value of I is simple rational arithmetic. Replacing q_i by x and θ is essentially just tuple (or "array") lookup. Note, however, that x is really still just an unknown value during specialization, and will carry different values as the actual simulation is being performed.

4.4 Symbolic Differentiation

Symbolic differentiation plays an important role in mapping analytical models to executable simulation code. First, it is needed to normalize applications of the differentiation operator to terms that consist of anything other than variables. Second, for a large class of physical modeling problems, eliminating partial derivatives is an important step in deriving executable code for solving the problem. In general, a wide range of techniques may be needed to eliminate partial derivatives, but symbolic differentiation generally plays an important role. For models arising from rigid body mechanics problems, symbolic differential on terms available in the model is in fact sufficient.

In the course of developing Acumen we were surprised to find out that mainstream symbolic manipulation tools do not seem to provide scalable support for symbolic differentiation. In particular, mainstream systems seem to inline all definitions before performing differentiation and not make an effort to avoid generating terms containing duplicate subexpressions. This poses a serious scalability problem, because many models that can be compactly specified (such as the spherical harmonics problem presented earlier) produce huge terms if they are fully inlined. This is particularly surprising because more efficient implementation techniques for performing symbolic differentiation, which are generally refered to as *automatic differentiation*, have been known for over 20 years (c.f. [17]).

To circumvent this problem in a clear and self-contained manner, we implemented a specialized symbolic differentiation procedure for Acumen, which computes derivatives of mathematical definitions without fully inlining. We do not view this as an innovation to automatic differentiation, but rather, a simple way of showing how the performance of naive symbolic differentiation can be significantly improved. In programming languages terms, inlining is avoided by extending a basic symbolic differentiation algorithm to support local definitions ("let expressions") and hash-consing the construction of new expressions. The result of using this strategy to eliminate the partial derivatives in our running example is presented in Figure 7(e). Note that the variables A and B name two computations that would otherwise each appear duplicated twice in the equations that follow. Due to the behavior of the chain rule, this duplication behavior would get compounded as terms grow larger, and can make symbolic differentiation prohibitively costly. While significantly more sophisticated implementations of symbolic differentiation exist (c.f. [17] for a recent account), Section 5 shows that even the simple representational improvement used in Acumen leads to dramatic improvement over the performance of symbolic differentiation in Mathematica and Maple.

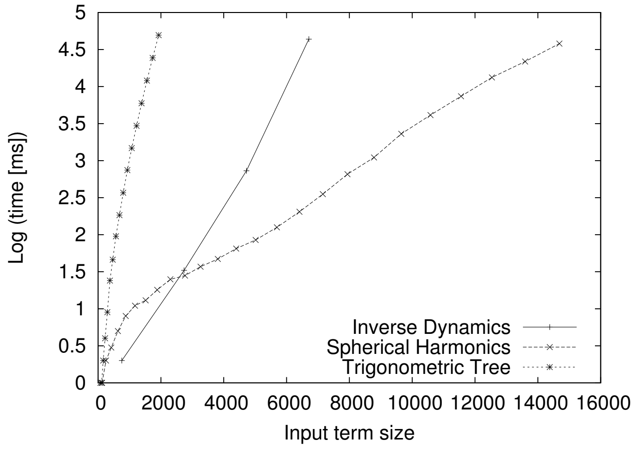
4.5 Algebraic Solving and Discretization

The algebraic solver used in Acumen is an extension of the Defined Variable Analysis described above. The main extension in this analysis is that it supports solving multiple equations simultaneously, which is achieved using an analog of Gaussian elimination. For our running example, the result of this step is presented in Figure 7(f). The basic idea of this type of "analytic solving" is to put the equations into a form where there is always a variable on the left-hand side, and all variables occurring on the right-hand side are somehow considered "known". We omit the details of this process because it is standard, and elegant descriptions can be found elsewhere (c.f. Chapter 7 in [8]). It is also important to note that this is neither the only way nor the most general way for solving DAEs. Nevertheless we include it in the implementation of Acumen because it allows it to be a useful tool without requiring the installation of stand-alone solvers, and because it is useful to show that there is at least one method for a non-trivial class of DAEs.

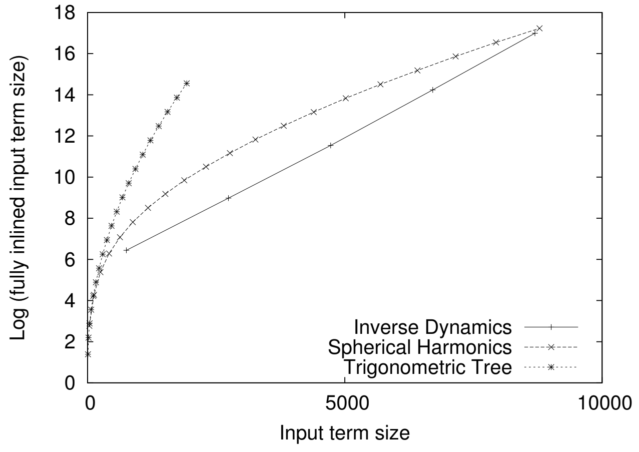
Even when differential equations have been put into a "solved" or, more accurately *explicit* form, a final question that must be answered is how we will step through these equations using discrete, finite, steps in time. The choice of this strategy is called discretization, and often corresponds to mapping the differential equations into *difference equations*. While a wide range of techniques also exists for realizing this step, and a few are implemented in Acumen, we only give one example of what such equations may look like. The equations presented in Figure 7(g) are virtually identical syntactically to those from the previous phase. The two main differences are 1) to distinguish the values after the discrete time step with superscript of $+$, and 2) to add additional equations that explicitly perform the approximate integration needed to compute lower-degree derivatives from higher degree ones. To do the latter, of course, an explicit time-step parameter Δt is introduced.

5. SYMBOLIC DIFFERENTIATION

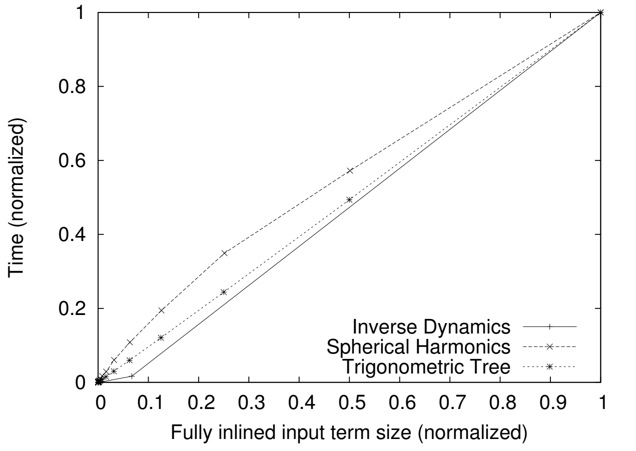
To evaluate the performance of Acumen's differentiation



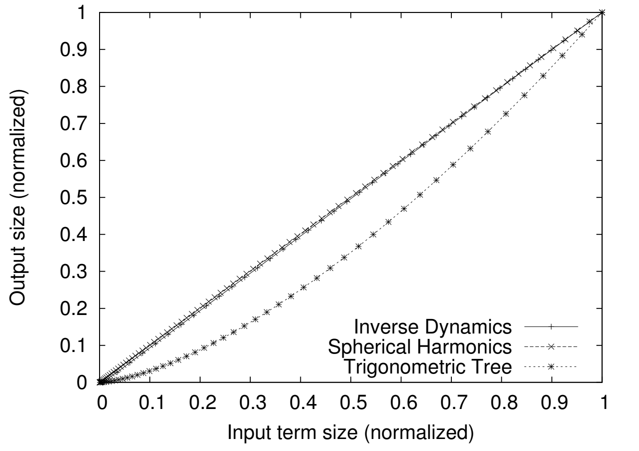
(a) Mathematica is exponential in input term size.



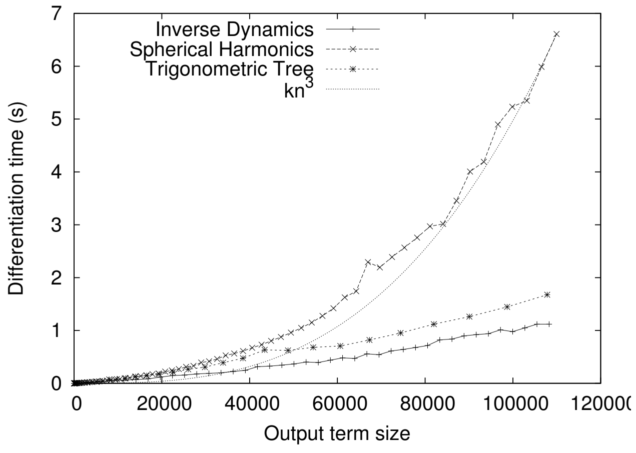
(b) Fully inlined output terms are exponentially bigger.



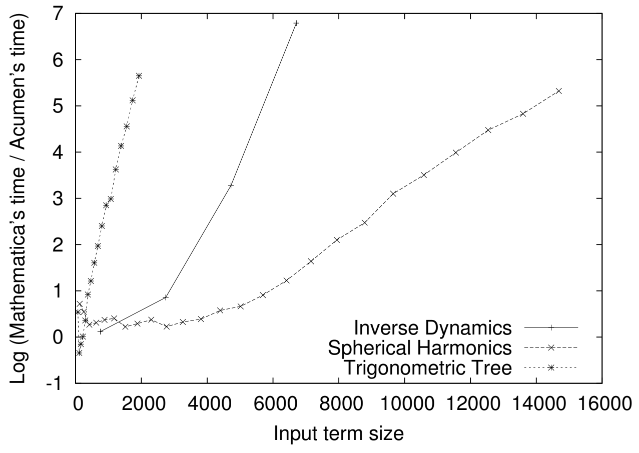
(c) Mathematica is linear in fully inlined input term size.



(d) Acumen derivatives are polynomial in input size.



(e) Acumen differentiation time is polynomial in output size.



(f) Acumen is exponentially faster than Mathematica.

Figure 8: Differentiation in Acumen vs. Mathematica.

procedure, we compare it with symbolic differentiation in Mathematica and Maple. Three benchmark models are used: An inverse dynamics model, a spherical harmonics model, and a trigonometric tree model. Timings are taken on a quad-core Pentium D 3.20 GHz with 3.9 GB RAM running Red Hat Linux 4.1.2-44, Linux kernel 2.6.18-128.el5PAE compiled with GCC 4.1.2. Acumen is compiled with the native code compiler of OCaml 3.11.1.

Figure 8 presents results that help compare and relate the performance of Acumen with Mathematica. Figure 8(a) shows Mathematica's differentiation time against the size of the term being differentiation. Because log-scale is used for the time dimension, and because the curves seem to converge to an upward linear pattern, it appears that Mathematica's differentiation time is exponential in term size. Our hypothesis is that this is because Mathematica fully inlines auxiliary definitions into the main program. As Figure 8(b) suggests, fully inlining auxiliary definitions leads to exponentially larger terms, and Figure 8(c) suggests that Mathematica's differentiation time is roughly linear in the size of fully inlined terms. Both these observations seem to support our hypothesis. In contrast to Mathematica's performance, Figure 8(d) shows that Acumen's procedure produces derivatives that are polynomial in the input size rather than exponential. This validates that Acumen's procedure avoids unnecessary inlining. Figure 8(e) shows that Acumen's differentiation time appears to be polynomial in the size of the output terms. Figure 8(f) shows that Acumen's differentiation procedure is faster than Mathematica, and that Acumen's advantage seems to grow exponentially with input size.

For Maple, the situation is similar to that with Mathematica, with the exception of a more accentuated initial dip in Acumen's performance for some benchmarks. Maple does provide an automatic differentiation routine that should be much more efficient. However, at this time we have not determined whether it can be used in all situations where the symbolic differentiation algorithm can be used.

6. RELATED WORK

Our work is inspired in part by Functional Reactive Programming (FRP) [10, 11, 18, 30] and by Functional Hybrid Modeling (FHM) [24, 13, 14, 27, 15]. FRP introduced the idea of combining continuous behaviors and discrete events for modeling reactive systems, and FHM brings this idea significantly closer to the natural way in which analytical models seem to be written. For example, whereas FRP is a causal modeling language, FHM is an acausal, higher-order modeling language. Compared to FHM, Acumen is unique in providing a clear binding time separation in the source language, as well as in its support for partial derivatives. However, FHM also supports features that Acumen does not support, including structural dynamism, where the connectivity of entities changes dynamically during simulation time. At a more technical level, Acumen is a closed DSL, in that it is not embedded in a host language. In contract, FHM is so far primarily implemented as an embedded language in Haskell. Embedding seems to greatly simply experimentation with different implementation strategies and different possible interpretations of various language constructs. Closing the language helps us keep a clear distinction between syntax and semantics, and provides us with fine-grained tools for controlling both as we work to understand how domain experts write and reason about analytical models.

Carette et al.'s observations about model manipulation [7] greatly parallel ours, in that they recognize that the highest level formal artifact that relates to building simulation codes is the mathematical model. However, the paper has a different target audience than ours. In particular, the paper promotes a strategy where the domain-expert is intimately involved in specifying explicit transformation steps that gradually take the model closer to the simulation code. Our approach is aimed instead at one formalism for specifying the model, and a fixed strategy for mapping this formalism to executable code. As such, we view their exploration of what can be done with automatic code generation as insightful first steps towards what has been demonstrated by our work. On the other hand, by allowing human intervention in the transformation process, they can guide the transformation of a larger class of models to executable code. As such, we foresee an elegant synergy between our approach and theirs, where we approximate the set of models that can be mapped automatically from the inside out (from smaller sets to larger), and they explore its approximation from the outside in.

Our paper focuses on the language for modeling continuous behaviors. Mosterman [22] provides an extensive discussion of the issues that arise when continuous behaviors and discrete events are combined.

7. CONCLUSIONS

In this paper we have shown how analytical models of a particular class of physical systems, namely, mechanical systems, can be automatically mapped to executable simulation codes. This mapping has the potential to significantly reduce the cost and time needed to develop simulation codes for novel cyber-physical systems. Our work sheds light on the fact that there are still significant opportunities for improving tool support for the process of developing novel cyber-physical systems.

8. ACKNOWLEDGMENTS

We are deeply indebted to Jacques Carette for guiding us in the process of implementing symbolic differentiation, and discussing the literature on symbolic differentiation and automatic differentiation. We are also very thankful to Brian Guenter for insights discussions about symbolic differentiation, and for providing us with the Trigonometric Tree benchmark. We are also thankful to Kristoffer Lidström, Zain-ul-Abdin, and Yan Wang for their excellent feedback on an early release of Acumen.

9. REFERENCES

[1] M. D. Ardema. *Analytical Dynamics: Theory and Applications*. Kluwer Academic/Plenum, New York, NY, 2005.

[2] Autodesk. *AutoCAD User's Guide*, 2009. [`http://docs.autodesk.com/ACD/2010/ENU/AutoCAD2010UserDocumentation/`; accessed 17-October-2009].

[3] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *PETSc Users Manual*, 2008. [`http://www.mcs.anl.gov/petsc/petsc-as/`

`snapshots/petsc-current/docs/`; accessed 17-October-2009].

[4] Biopython. *Biopython Documentation*, 2009. [`http://www.biopython.org/wiki/Documentation`; accessed 17-October-2009].

[5] A. Boeing and T. Bräunl. Evaluation of real-time physics simulation systems. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 281–288, New York, NY, USA, 2007. ACM.

[6] A. Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, 1992.

[7] J. Carette, S. Smith, J. McCutchan, C. K. Anand, and A. Korobkine. Case studies in model manipulation for scientific computing. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *AISC/MKM/Calculemus*, volume 5144 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 2008.

[8] F. E. Cellier and K. Ernesto. *Continuous System Simulation*. Springer, New York, NY, 2005.

[9] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[10] C. Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, Oct. 1997.

[11] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[12] P. A. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. IEEE Press, New York, NY, 2004.

[13] G. Giorgidze and H. Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Implementation and Application of Functional Languages: 20th International Workshop (IFL'08), Hatfield, UK September 2008, Revised Selected Papers*. Springer-Verlag, 2009.

[14] G. Giorgidze and H. Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference*, Linköping Electronic Conference Proceedings, pages 208–218, Como, Italy, Sept. 2009. Linköping University Electronic Press.

[15] G. Giorgidze and H. Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10)*, pages 19–34, Madrid, Spain, Jan. 2010.

[16] A. Goswami, B. Thuilot, B. Espiau, and L. G. Gravir. A study of the passive gait of a compass-like biped robot: Symmetry and chaos. *International Journal of Robotics Research*, 17:1282–1301, 1998.

[17] B. Guenter. Efficient symbolic differentiation for graphics applications. *ACM Trans. Graph.*, 26(3):108, 2007.

[18] P. Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[19] N. D. Jones, P. Sestoft, and H. Sondergraard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[20] R. Kaiabachev, W. Taha, and A. Zhu. E-frp with priorities. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 221–230, New York, NY, USA, 2007. ACM.

[21] Maplesoft, a division of Waterloo Maple Inc. *Maple User Manual*, 2009. [`http://www.maplesoft.com/documentation_center`; accessed 3-January-2010].

[22] P. J. Mosterman. Hybrid dynamic systems: Modeling and execution. In P. A. Fishwick, editor, *Handbook of Dynamic System Modeling*. CRC Press, 2007.

[23] F. Nielson and H. R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.

[24] H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, Jan. 2003. Springer-Verlag.

[25] PTC. Pro/ENGINEER, 2009. [`http://www.ptc.com/products/proengineer/`; accessed 17-October-2009].

[26] The R Development Core Team. *The R Manuals*, 2009. [`http://cran.r-project.org/manuals.html`; accessed 17-October-2009].

[27] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *Proceedings of the Fourteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 23–34, Edinburgh, Scotland, Sept. 2009. ACM Press.

[28] SolidWorks Corp. SolidWorks, 2009. [`http://www.solidworks.com/`; accessed 17-October-2009].

[29] The MathWorks, Inc. *MATLAB Documentation*, 2009. [`http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html`; accessed 17-October-2009].

[30] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *the Symposium on Programming Language Design and Implementation (PLDI '00)*. ACM, 2000.

[31] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages*. ACM, Jan 2002.

[32] Wolfram Research, Inc. *Mathematica Documentation*, 2009. [`http://reference.wolfram.com/mathematica/guide/Mathematica.html`; accessed 3-January-2010].