Deductive Synthesis of Numerical Simulation Programs from Networks of Algebraic and Ordinary Differential Equations

Thomas Ellman
Department of Computer Science
Rutgers University
New Brunswick, NJ 08855
ellman@cs.rutgers.edu

Takahiro Murata
Department of Computer Science
Rutgers University
New Brunswick, NJ 08855
murata@cs.rutgers.edu

Abstract

Computational science and engineering design can benefit from software tools that facilitate construction of programs for simulating physical systems. Our research adapts the methodology of deductive program synthesis to the problem of synthesizing numerical simulation codes. We have focused on simulators that can be represented as second-order functional programs composed of numerical integration and root-extraction routines. Synthesis of second-order programs appears to present a problem for deductive systems that operate in first order logic. We present a recursive synthesis algorithm that overcomes this difficulty for a class of program synthesis problems. Our system has successfully constructed numerical simulators for computational design of jet-engine nozzles and sailing yachts, among others.

1 Introduction

Computational science and engineering design can benefit from software tools that facilitate construction of programs for simulating physical systems. Scientists and design engineers both face a recurring problem of constructing, testing and modifying numerical simulation models. Design engineers often don't know in advance what physical phenomena will dominate the design process. Scientists often don't know what processes must be modeled in order to investigate the particular issues they are studying. Both types of professional typically take an experimental approach to the problem. They develop one simulation program, examine its qualitative or quantitative behavior, and subsequently modify it many times before arriving at a simulator that is adequate to the task at hand. The process of coding and revising such simulators is extremely time-consuming because such models are almost always written in conventional programming languages, usually Fortran.

This unfortunate situation appears not to be inevitable. A wide variety of systems can be modeled as networks of algebraic and ordinary differential equations. Examples include an aircraft simulator for computational design of jet-engine nozzles; [Gelsey *et al.*, 1996] a sailing yacht simulator for computational design of hull geometries [Ellman *et al.*, 1993]; and a planetary scientist's model of the atmosphere of the moon Titan [Keller *et al.*, 1994], among others. Simulators for such systems can be constructed as configurations of numerical integration and root extraction routines. Libraries of such generic numerical tools are now widely available [Press *et al.*, 1986]. They are under-utilized because of the difficulty of bridging the gap between equations - the language with which scientists and engineers are familiar - and the facilities provided by numerical tool libraries. Tools that bridge this gap promise to dramatically increase the productivity of working scientists and engineers.

Our research addresses the problem of synthesizing numerical simulation codes from networks of algebraic and ordinary differential equations. We are focusing on systems in which the algebraic and differential equations can be implicit (i.e., neither provides closed form formulas for desired quantities) and for which simulation can be carried out using combinations of numerical integration and root-extraction. An example of such a system is described informally in Figure 1. This example is a simplified version of an aircraft simulation problem used in research on design of jet-engine nozzles [Gelsey *et al.*, 1996]. The required simulator takes as input the size n of the nozzle, the empty mass m_e of the aircraft, and the velocity v and duration d of a flight. The simulator must determine the fully fueled mass m_f of the aircraft that will enable it to fly the prescribed mission - landing with empty fuel tanks. The instantaneous fuel mass consumption $\dot{m}(t)$ is described

Inputs:

- Size of the jet engine nozzle: n.
- Empty mass of the aircraft: m_e.
- Velocity of the aircraft: v.
- Duration of the flight: d.

Outputs:

- Fully fueled mass of the aircraft: m_f.

Specification:

$$
\begin{array}{rcl}
m(0) & = & m_e \\
m(d) & = & m_f \\
(\forall t) \;\; a_v(m(t), \alpha(t), v) & = & 0 \\
(\forall t) \;\; a_h(m(t), \dot{m}(t), \alpha(t), n) & = & 0
\end{array}
$$

Figure 1: Informal Specification of Simplified Aircraft Simulation Problem

implicitly, along with angle of attack $\alpha(t)$, by two simultaneous equations. A simulator for this problem can be constructed using a Runge-Kutta code to integrate $\dot{m}(t)$ backwards in time, starting with the empty mass of the aircraft, and ending with the mass of the fully fueled aircraft. The function to be integrated, $\dot{m}(t)$, can be computed from the instantaneous mass $m(t)$ by using a Newton-Raphson root-extraction code to solve the simultaneous equations at each point in time during the flight of the aircraft. Construction of such a simulator requires recognizing that the desired output m_f can be computed by integration; recognizing that the derivative $\dot{m}(t)$ can be computed by root-extraction; and formulating functional parameters to the integration and root-extraction routines that depend appropriately on the overall problem inputs, n, m_e, v and d.

The system we describe in this paper is one part of a larger system called the "DA-MSA" (Design Associate and Modeling/Simulation Associate). The architecture of the program synthesis components of the DA-MSA is shown in Figure 2. A human engineer uses a graphical interface to build a constraint network. The network represents a system of algebraic and ordinary differential equations. It also describes the inputs and outputs of a class of design simulation problems. In the first stage of operation the system uses deductive synthesis methods to construct an initial, executable, program for simulating the behavior of a design. The initial simulation program is represented internally as a LISP function. The function may include second-order operations such as integration and root extraction, i.e., operations that require functional arguments. The simulation program is presented visually to the user as a data-flow graph. The deductive methods used in this first stage of our system will be described in this paper. In the second stage of operation, the simulator program is combined with a numerical optimization routine. The user then draws upon a catalog of transformations to convert the initial simulation/optimization program into a more robust one. The catalog includes transforms for solving equations symbolically, selecting numerical methods or tolerance parameters, approximating functions and reformulating design parameter spaces, among others. The transformation system used in this stage was previously reported in [Ellman *et al.*, 1995]. At present, both the deductive synthesis system and the transformation system are implemented. The graphical interface for interacting with the transformation system is also implemented; however, the graphical interface for constructing the initial constraint network is not yet implemented. Thus all the specifications used in experiments reported in this paper were constructed by hand.

2 Approach

Our approach is based loosely on *deductive program synthesis* [Manna and Waldinger, 1992]. A specification in our system defines a relation $S(x, y)$ between a program input x and a desired output y. A program to satisfy the specification is synthesized by proving that for every input x there exists some output y satisfying $S(x, y)$. The proof is constructive in the sense that it actually generates a term $t(x)$ such that $(\forall x)S(x, t(x))$. The proof is based on axioms about properties of predefined primitive operations that are referenced in the term $t(x)$. The actual synthesized program has the form: $\lambda(x).t(x)$.

A problem arises immediately in attempting to use deductive methods to synthesize simulation programs built out of integrators and root extractors. Since integration and root extraction are second order operations, verification of such programs would appear to require a second order logic, i.e., one that can quantify over functions. Methods of reasoning about equality would appear to be necessary in order to handle algebraic and differential equations in a general manner. Unfortunately, a higher order equational logic would saddle us with a cumbersome formalism and a compu-

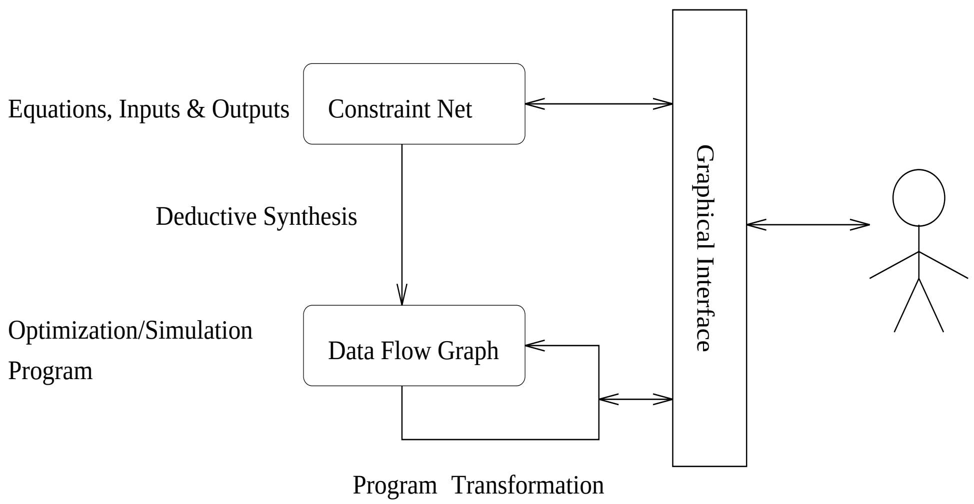

Figure 2: Program Synthesis in the DA-MSA

tationally expensive inference process.

In order to avoid such complex machinery, we have attempted to implement our system in first-order horn clause logic without equality. A key feature of our specification language makes this possible. We have reified the relationship between input and output quantities computed by functions, i.e., assertions in the database explicitly relate named quantities to specifications of functions for computing them. This allows us to axiomatize primitive functions for integration and root extraction with horn rules that reference the specification of named quantities. Such axioms lead to recursive invocation of the deductive synthesis process and thereby generate second order programs.

2.1 Specification Language

Predicates used in our specification language are described in Figure 3 along with an informal description of their semantics. The predicate *spec* is used to define a relation between inputs and outputs of a function. A *spec* clause asserts that the quantities on the list *outputs* are properly related to the quantities on the list *inputs* provided the clauses on *givens* entail the clauses on *proves*. A single such clause may specify several functions - one function for each output quantity. The predicate *apply* is used to state equations in the specification. An *apply* clause asserts an equality between a *term* and an application of a *function* to *terms*. The predicate *derivative* is used to indicate that one function is the derivative of another. A *derivative* clause asserts that the function named by *derivative* computes the derivative of the function named by *integral*. The *derivative* function accepts the quantities on *variables* and the independent variable *time* as inputs. Finally, the predicate *program* is used to associate functions with their implementations. A *program* clause appearing in a specification asserts that the named *function* is computed by a predefined *lambda* expression.

A specification for the simplified aircraft problem is shown in Figure 4. This specification describes a function that computes the *full-mass* of the aircraft when given the *empty-mass* of the aircraft, the size of the *nozzle*, the flight *velocity* and the flight *duration* as inputs. The "prove" list asserts that the value of the *mass* function at time *duration* is *full-mass*. The "givens" list asserts that the value of the *mass* function at time zero is *empty-mass*. The givens also assert that the derivative of the *mass* function is computed from *mass* and *time* by the *mdot* function. The function *mdot* is specified in another clause. This clause describes functions to compute the instantaneous fuel-mass consumption rate *mdot*, and the instantaneous *angle* of attack, when given the instantaneous *mass*, the *nozzle* size and the flight *velocity* as inputs. Its "prove" list asserts that the *vertical* and *horizontal* acceleration are zero at each point in time. Since its

- `spec(outputs,inputs,proves,givens)`: The *outputs* are properly related to the *inputs* provided the clauses on *givens* entail the clauses on *proves.*
- `apply(function,arity,args,terms,term)`: When the *function* of *arity* arguments is applied to *terms*, by binding the arguments on *args* the result is *term.*
- `derivative(integral,variables,function)`: The function *derivative* gives the derivative of the function *integral* when applied to the quantities on *variables* and the independent variable (time).
- `program(function,lambda)`: A pre-defined primitive program for *function* is *lambda.*

Figure 3: Predicates used in Specification Language

"givens" list is empty, these conditions must be satisfied unconditionally. Finally, the specification includes two *program* clauses providing implementations of pre-defined functions for computing vertical and horizontal acceleration.

2.2 Implementation Language

Our implementation language is described in Figure 5. It includes two second-order functions *integral* and *root*, and one first-order function *extract*. The *integral* function accepts as input a list of lambda expressions, an initial state, a start time and an end time. It returns the state that results from integrating the dynamic system represented by the lambda expressions from the start time to the end time. The *root* function accepts as input a list of lambda expressions. It returns a list of values that represents the simultaneous root of the functions defined by the lambda expressions. The *integral* and *root* functions are implemented in a run-time system. The implementation is composed of C routines taken from [Press *et al.*, 1986]. The C routines are wrapped by LISP functions providing the interfaces described above [Keane, 1996]. The *integral* function is computed using a fourth-order Runge-Kutta method. The *root* function is computed using the Newton-Raphson method. An example (in PROLOG syntax) of a program in our implementation language is shown in Figure 8. This program is a solution to the simple aircraft simulation problem specified in Figure 4. A translation of the same program into LISP is shown in Figure 9.

- `root(lambdas)`: Given a list of n lambda expressions, defining functions $f_i(x_1,\ldots,x_n)$ for $i = 1\ldots n$, return a list of length n representing a solution to the simultaneous equations $f_i(x_1,\ldots,x_n) = 0$ for $i = 1\ldots n$.
- `integral(lambdas,state,start,end)`: Given a list of n lambda expressions, defining functions $f_i(t,x_1,\ldots,x_n)$ for $i = 1\ldots n$, representing the derivative of an n-dimensional dynamic system; an initial state $(x_1^0,\ldots,x_n^0)$; a start time t_0; and an end time t_e; return a list $(x_1^e,\ldots,x_n^e)$ representing the state of the system at time t_e.
- `extract(n,list)`: Return the *nth* member of *list.*

Figure 5: Functions used in Implementation Language

2.3 Synthesis Rules

Our knowledge base consists of three groups of rules. One group serves to axiomatize properties of primitive functions in our implementation language. For example, we have rules that axiomatize second order functions such as *integral* and *root.* Another group of rules defines simple transformations on functions and lambda expressions. Examples of such transformations include lambda abstraction (adding dummy arguments), currying (binding arguments) and permutation (reordering arguments). A third group of rules define the recursive structure of our synthesis algorithm. All of the rules are implemented in Quintus PROLOG. The overall synthesis process is controlled by a meta-interpreter we have written in PROLOG.

Examples of rules that axiomatize primitive, second-order functions are shown in Figure 6. (These rules are simplified versions of actual rules in our system.) The first rule states that result of applying a function f to $root([\lambda])$ is zero, provided that λ is a program implementing function f. Thus, the *root* function finds a root. The second rule asserts conditions under which the result of applying a function f to a value t_f is equal to the value obtained by integrating $\dot{f}$ from t_0 to t_f starting in state x_0. In particular, $f(t_f)$ will be $integral([\lambda], x_0, t_0, t_f)$ provided $f(t_0)$ is x_0, $\dot{f}$ is the derivative of f, and $\dot{f}$ is implemented by λ. Thus, the *integral* function integrates. These two rules describe the behavior of *root* and *integral* on one-dimensional problems. Generalizations of these rules describe the behavior of *root* and *integral* on multi-dimensional problems.

The overall behavior of our synthesis algorithm is

```
spec([full-mass],                                   %Outputs.
     [empty-mass,nozzle,velocity,duration],         %Inputs.
     [apply(mass,1,[1],[duration],full-mass)],      %Proves.
     [apply(mass,1,[1],[0],empty-mass),             %Givens.
      derivative(mass,[mass],mdot)])

spec([mdot,angle],
     [mass,nozzle,velocity],
     [apply(vertical,3,[1,2,3],[mass,angle,velocity],0),
      apply(horizontal,4,[1,2,3,4],[mass,mdot,angle,nozzle],0)],
     []
     )

program(vertical,lambda([MASS,ANGLE,VELOCITY],vac(MASS,ANGLE,VELOCITY))).

program(horizontal,lambda([MASS,MDOT,ANGLE,NOZZLE],hac(MASS,MDOT,ANGLE,NOZZLE))).
```

Figure 4: Formal Specification of Simplified Aircraft Simulation Problem

```
apply(F,1,[1],[extract(1,root([LAMBDA]))],0) :- program(F,LAMBDA).

apply(F,1,[1],[TF],extract(1,integral([LAMBDA],[X0],T0,TF))) :- apply(F,1,[1],[T0],X0),
                                                                derivative(F,[F],DF),
                                                                program(DF,LAMBDA).
```

Figure 6: Rules to Axiomatize Root-Extraction and Integration

described by the rule shown in Figure 7. (This rule is a simplified version of an actual rule from our system.) It describes a procedure for processing a goal of the form *program(X, Y)* to find a lambda expression that computes an output quantity: First find a specification for the output quantity and identify the "proves". Next rewrite the proves by replacing each occurrence of the output quantity with a new variable "NEWVAR". Then prove that every condition on the proves follows from the givens (when combined with the permanent parts of the database). The resulting binding for "NEWVAR" is the body of a lambda expression for computing the output quantity.

When this algorithm used to process the specification of the simple aircraft simulation problem shown in Figure 4, the result is the simulation program shown in Figure 8. The simulation program operates by integrating the fuel-mass consumption rate $\dot{m}(t)$ over the duration of the flight. The derivative $\dot{m}(t)$ is computed by using a numerical root extractor to find values of fuel mass consumption $\dot{m}(t)$ and angle of attack $\alpha(t)$ that make horizontal acceleration a_h and vertical acceleration a_v go to zero. In this solution, the simultaneous equations $a_h = 0$ and $a_v = 0$ are expressed as nested one-dimensional roots. Our algorithm also finds other solutions to this problem, including some in which the equations are solved as a two-dimensional root.

A LISP function implementing the simple aircraft simulator is shown in Figure 9. This program was generated by a back-end translator in our synthesis system. The back-end translates PROLOG terms into LISP expressions and rewrites some function calls into a more readable keyword-oriented syntax. It also introduces variables representing seeds needed by the numeric root extraction functions. This LISP function has a form that is amenable to transformation using the tools described in [Ellman *et al.*, 1995], e.g., transformations that use symbolic algebra to replace numeric root-extraction expressions with closed-form solutions.

Insight into the behavior of our program synthesis algorithm may be gained by examining the hierarchic structure of the simple aircraft simulation program. The simulator involves an integrator wrapped around a root extractor wrapped around another root extractor.

```
program(OUTPUTS,lambda(INPUTS,NEWVAR))
    :-  spec(OUTPUTS,INPUTS,PROVES,GIVENS),
        variablize(PROVES,OUTPUTS,NEWVAR,NEW-PROVES),
        prove(NEW-PROVES,GIVENS).
```

Figure 7: Top Level Program Synthesis Rule

```
lambda([empty-mass,nozzle,velocity,duration],
       [extract(1,
                integral([lambda([t,mass],
                                 [extract(1,
                                          root([lambda([mdot],
                                               hac(mass,
                                                   mdot,
                                                   extract(1,
                                                           root([lambda([angle],
                                                                        vac(mass,
                                                                            angle,
                                                                            velocity))])),
                                                   nozzle))]))])],
                         [empty-mass],
                         0,
                         duration))])
```

Figure 8: Synthesized Prolog Term Representing Simulation Program

These nested second-order functions result from the recursive structure of our synthesis algorithm. In the course of processing a goal of the form *program(X, Y)*, our algorithm may invoke goals on the "proves" of some specification. (See Figure 7). These goals may, in turn, be processed by the rules that axiomatize the *root* and *integral* functions. (See Figure 6). The rules that axiomatize integration and root extraction both have references to needed programs for named functions. They lead to recursive invocation of goals for the form *program(X, Y)*. Thus, in generating the simple aircraft simulator, our synthesis algorithm goes through two levels of recursion. One level constructs the outer integration. One level constructs the embedded root extraction. Despite the restrictions of first order horn logic, the recursive structure of our synthesis algorithm generates hierarchically nested second-order programs.

3 Results

We have tested our program synthesis system on two groups of realistic simulation problems, and one group of synthetic problems. The real problems were drawn from two testbed domains we have used in our computer-aided design research: (1) simulating the fuel consumption of a jet aircraft under various different jet engine nozzle designs; (2) simulating the motion of a racing sailing yacht under various hull designs. The synthetic problems included a ball thrown in a gravitational field, among others. Each problem domain is specified by a network of linked algebraic and differential equations. Within each domain, problems vary according to the quantities that are given as inputs and the quantities that are desired as outputs. In "final state" problems, the initial state s_i of the system is specified along with an initial time t_i and a final time t_f. The simulator must compute the final state s_f at time t_f. In "final time" problems, the initial state s_i and final state s_f are specified along with the initial time t_i. The simulator must find the final time t_f at which the system moves into state s_f. In "mixed initial/final value" problems, parts of the initial and final states s_i and s_f are specified along with the initial and final times t_i and t_f. The simulator must find the remaining parts of the initial and final states. Finally, in "boundary-value" problems, the system is given the initial and final times t_i and t_f along with equations

```
(defun full-mass  (empty-mass nozzle velocity duration)
  (list
   (integral-1d :function
                #'(lambda (time mass)
                    (root-1d :function
                             #'(lambda (mdot)
                                 (hac mass
                                      mdot
                                      (root-1d :function
                                               #'(lambda (angle)
                                                   (vac mass angle velocity))
                                               :seed *seed-1*)
                                      nozzle))
                             :seed *seed-2*))
                :initial-state empty-mass
                :start 0
                :end duration)))
```

Figure 9: Lisp Version of Simulation Program

putting constraints on the initial and final states. The system must find complete descriptions of the initial and final states s_i and s_f that satisfy the boundary conditions.

The appendix provides summary of the problems on which we have tested our system, along with an indication of whether our system was successful in constructing a program for that problem. Each of these problems was solved in less than a minute on a Sun Microsystems Sparc 10 workstation. Our system was successful in constructing programs for two aircraft simulation problems (a final state problem and a final time problem); three yacht simulation problems (a purely algebraic steady-state motion problem, a final state problem and a final time problem); and one synthetic, mixed initial/final value problem. Each synthesized program was translated into LISP by our back-end translator, and compiled and executed on test examples. Each program was verified to give qualitatively correct numerical results; however, detailed numerical validation was not attempted.

We had hoped our system would synthesize a "shooting method" [Press *et al.*, 1986] for solving arbitrary boundary value problems. A general description of such problems is shown in Figure 10. We tested this hypothesis on a synthetic boundary-value problem involving a two dimensional dynamic system. Unfortunately, our system was not able to construct a shooting method, or any simulation program, for this problem. In contrast, our system was successful in constructing a shooting method to handle mixed initial/final value problems. The two types of problem differ according to the manner in which boundary conditions are specified. In the mixed initial/final value problem, boundary conditions are given explicitly. In the general boundary value problem, the conditions are all given implicitly. Apparently our axiomatization of integration and root extraction is not sufficiently general to verify (and synthesize) shooting methods for boundary value problems with implicitly specified boundary conditions. We believe that this negative result may be consequence of our decision to remain within the framework of first-order horn logic. Should this be the case, we would be forced to consider various alternative methodologies. One possibility would be to continue with deductive synthesis, but to go beyond horn logic to first order logic with equality. Another possibility would be to utilize a higher order logic. Finally, entirely different approaches to program synthesis may be considered, for example, a methodology based on specification morphisms [Smith, 1993] or constructive type theory [Constable, 1986]. In any case, a more complete diagnosis of the difficulty awaits further investigation.

4 Related Work

Our research was originally motivated by difficulties encountered in attempting to apply the SIGMA scientific modeling system [Keller *et al.*, 1994] to the task of constructing numerical simulators for automated op-

Inputs: A point in time: t.
Output: The state $(x_1(t), \ldots, x_n(t))$ of the system at time t.
Specification:

$$
\begin{aligned}
b_1(x_1(t_i), \ldots, x_n(t_i)) &= 0 \\
&\cdots \\
b_k(x_1(t_i), \ldots, x_n(t_i)) &= 0 \\
b_{k+1}(x_1(t_f), \ldots, x_n(t_f)) &= 0 \\
&\cdots \\
b_n(x_1(t_f), \ldots, x_n(t_f)) &= 0 \\
(\forall t) \quad f_1(\dot{x}_1(t), \ldots, \dot{x}_n(t), x_1(t), \ldots, x_n(t), t) &= 0 \\
&\cdots \\
(\forall t) \quad f_n(\dot{x}_1(t), \ldots, \dot{x}_n(t), x_1(t), \ldots, x_n(t), t) &= 0
\end{aligned}
$$

Figure 10: General Boundary Value Problem

timization of designs of jet-engine nozzles and sailing yachts. SIGMA uses a graphical first-order data-flow language to represent scientific models. Unfortunately SIGMA is not capable of representing models based on simultaneous algebraic and differential equations in a general manner. We considered modifying the first-order data-flow language used in SIGMA to include second order operations like integration and root extraction. It quickly became clear that scientists and design engineers could not be expected to directly implement second order functional programs, even when provided with a graphical interface. Furthermore, the real cause of the problem lay in the SIGMA representation language itself. The data-flow language used to represent models in SIGMA includes elements of specification and elements of implementation. The solution was to split the representation into two parts. A network of algebraic and differential equations represents specifications. A second-order data-flow language represents executable models. Our deductive synthesis system provides a mapping between these two levels of representation.

The Amphion system [Lowry *et al.*, 1994] is similar to the system we have described here. Amphion takes as input a problem specification in the form of a network of relationships among quantities and objects to be modeled. The network is provided by the user via a graphical user interface. Amphion uses a deductive method to synthesize a numerical program that calls library subroutines to satisfy the specification. The example of Amphion was part of our motivation for using a deductive approach in our research. Nevertheless, Amphion is not immediately applicable to our class of problems because it does not provide facilities for reasoning about differential equations. Our research may be seen as an approach to extending a system like Amphion in order to handle such problems.

The Synapse system [Kant, 1993] constructs finite element codes for simulating systems governed by partial differential equations, a class of problems that is distinct from the one we have attacked. Commercial mathematical toolkits, such as Matlab [Matlab, 1992] provide tools for integrating dynamic systems; however, Matlab's tools require that the system derivatives be given explicitly in closed form. Thus Matlab's tools do not immediately apply to several of the problems we have considered here, in which derivatives must be calculated by numerically solving algebraic equations.

5 Summary

We have demonstrated that deductive methods can be successfully adapted to the problem of constructing numerical simulators from networks of algebraic and ordinary differential equations. Our experiments were specifically focused on systems in which the algebraic and differential equations are implicit, i.e., neither provides closed form formulas for the desired quantities. Simulators for such systems can represented as second-order functional programs that use numerical integration and root-extraction as primitive operations. Synthesis of such programs can be carried out using first-order deductive methods by arranging for inference rules to access the specifications of named functions and recursively invoke the deductive synthesis process. This approach was shown to be successful on a variety of realistic and synthetic problems. A possible limitation of the approach was indicated by its failure to handle general boundary value problems. Future research will attempt to determine whether this limitation is inherent in our approach or merely reflects a temporary inability to find a suitable axiomatization of our routines for integration and root extraction.

6 Appendix

Aircraft Final State Problem: (Successful)
Problem: Find the fully-fueled mass m of an aircraft that can fly a given mission using a given jet engine nozzle design. The mission is specified by the altitude h velocity v and duration t_e of the flight. The nozzle is specified by three parameters l_1, l_2, l_3.

The empty mass of the aircraft is given as m_e. The instantaneous fuel consumption is governed by three equations: Two equations $a_v(h, v, m, \alpha) = 0$ and $a_h(l_1, l_2, l_3, h, v, \dot{m}, \alpha, \beta) = 0$ specify that vertical and horizontal acceleration be zero. A third equation $v_t(l_1, h, v, \dot{m}, \alpha, \beta) = 1.0$ specifies that the fuel-air mixture be traveling at mach 1.0 at the throat of the nozzle. The system of three equations involves instantaneous control parameters: fuel mass flow $\dot{m}(t)$, angle of attack $\alpha(t)$, and flap angle $\beta(t)$. Predefined functions compute the functions a_h, a_v and v_t.

Solution Found: Use a numerical integrator to integrate $\dot{m}(t)$ backwards through the flight of the aircraft, starting with the empty-mass m_e of the aircraft, a time zero, and ending with the mass of the fully fueled aircraft at time t_e. The instantaneous change in mass is determined using numerical root extractor to solve three equations $a_h = 0$; $a_v = 0$; and $v_t = 1.0$ for the instantaneous control parameters: $\dot{m}(t)$; $\alpha(t)$; $\beta(t)$.

Aircraft Final Time Problem: (Successful)

Problem: Find length of time t_e that an aircraft with empty mass m_e can fly at a given altitude h and velocity v, using a nozzle specified by three parameters l_1, l_2, l_3, with a specified mass m_f of the fully fueled aircraft. The instantaneous fuel consumption is governed by three equations: Two equations $a_v(h, v, m, \alpha) = 0$ and $a_h(l_1, l_2, l_3, h, v, \dot{m}, \alpha, \beta) = 0$ specify that vertical and horizontal acceleration be zero. A third equation $v_t(l_1, h, v, \dot{m}, \alpha, \beta) = 1.0$ specifies that the fuel-air mixture be traveling at mach 1.0 at the throat of the nozzle. The system of three equations involves instantaneous control parameters: fuel mass flow $\dot{m}(t)$, angle of attack $\alpha(t)$, and flap angle $\beta(t)$. Predefined functions compute the functions a_h, a_v and v_t.

Solution Found: Use a numerical root extractor to solve the equation $m(t_e) - m_e = 0$ for t_e. The function $m(t)$ is computed using a numerical integrator to integrate $\dot{m}(t)$ forwards in time, starting with the fully-fueled m_f of the aircraft, and ending with the mass at time t. The instantaneous change in mass is determined using numerical root extractor to solve three equations $a_h = 0$; $a_v = 0$; and $v_t = 1.0$ for the instantaneous control parameters: $f(t)$; $\alpha(t)$; $\beta(t)$.

Yacht Steady State Velocity and Heel Problem: (Successful)

Problem: Find the velocity v and heel angle α at which a sailing yacht's motion will be in a steady state, given parameters $(d_1, ..., d_n)$ representing the design of the sailing yacht, along with the windspeed w and the heading angle β at which the yacht sails. Predefined functions compute instantaneous force f and torque τ as a function of $(d_1, ..., d_n)$, w and β. Motion is in steady state when net force f and net torque τ are zero.

Solution Found: Use a numerical root extractor to find values of velocity v and heel α that make net force and net torque simultaneously go to zero.

Yacht Time-Dependent Final State Problem: (Successful)

Problem: Find the longitudinal position $x(t_e)$ and heel angle $\alpha(t_e)$ of a sailing yacht at a specified point t_e in time, given the initial position and heel $(x(0), \alpha(0))$ of the yacht, their derivatives $(\dot{x}(0), \dot{\alpha}(0))$, and parameters $(d_1, ..., d_n)$, w and β representing the design and environment in which the yacht sails. Predefined functions compute instantaneous force f and torque τ as a function of $(d_1, ..., d_n)$, w and β. The yacht's motion is governed by longitudinal and angular versions of Newton's laws: $f = m\ddot{x}$, $\tau = i\ddot{\alpha}$.

Solution Found: Use a numerical integrator to integrate a four dimensional dynamic system starting at the initial state $(x(0), \alpha(0), \dot{x}(0), \dot{\alpha}(0))$ and initial time $t = 0$ and ending at time t_e. The instantaneous system derivative is given by four functions that use a root extractor to solve the Newton equations $f = m\ddot{x}$, $\tau = i\ddot{\alpha}$; and the definitional equations $\ddot{x} = d/dt(\dot{x})$ and $\ddot{\alpha} = d/dt(\dot{\alpha})$.

Yacht Time-Dependent Final Time Problem: (Successful)

Problem: Find the length of time t_e needed for a yacht to sail to a specified distance d, given the initial position and heel $(x(0), \alpha(0))$ of the yacht, their derivatives $(\dot{x}(0), \dot{\alpha}(0))$, and parameters $(d_1, ..., d_n)$, w and β representing the design and environment in which the yacht sails. Predefined functions compute instantaneous force f and torque τ as a function of $(d_1, ..., d_n)$, w and β. The yacht's motion is governed by longitudinal and angular versions of Newton's laws: $f = m\ddot{x}$, $\tau = i\ddot{\alpha}$.

Solution Found: Use a numerical root extractor to solve the equation $x(t_e) - d = 0$ for t_e. The function $x(t)$ is computed using a numerical integrator to integrate a four dimensional dynamic system starting at the initial state $(x(0), \alpha(0), \dot{x}(0), \dot{\alpha}(0))$ and initial time $t = 0$ and ending at time t. The instantaneous system derivative is given by four functions that use a root extractor to solve the Newton equations $f = m\ddot{x}$, $\tau = i\ddot{\alpha}$; and the definitional equations $\ddot{x} = d/dt(\dot{x})$ and $\ddot{\alpha} = d/dt(\dot{\alpha})$.

Synthetic Mixed Initial/Final Value Problem: (Successful)

Problem: Given an explicit, but partial description of the trajectory of a ball moving in a uniform gravitational field, (i.e., given initial position x_i, final velocity v_f, initial time t_i and final time t_i), find the initial velocity v_i and final position x_f, assuming the motion of the ball is governed by the differential equation: $\ddot{x}(t) = -g$.

Solution Found: Use a numerical root extractor to solve for the variables x_f and v_i in the equations $e_x(x_i, v_i, t_i, t_f, x_f) = 0$ and $e_v(x_i, v_i, t_i, t_f, v_f) = 0$, where e_x and e_v both operate by integrating $(x(t), v(t))$ from t_i to t_f starting at the state (x_i, v_i). The function e_x returns $x(t_f) - x_f$. The function e_v returns $v(t_f) - v_f$.

Synthetic Boundary-Value Problem: (Not Successful)

Problem: Given an implicit partial description of the trajectory of a ball moving in a uniform gravitational field, (i.e., given initial time t_i, final time t_i, and two implicit boundary conditions $b_i(x_i, v_i) = 0$ and $b_f(x_f, v_f) = 0$), find the initial position x_i, initial velocity v_i, final position x_f and final velocity v_f, assuming the motion of the ball is governed by the differential equation: $\ddot{x}(t) = -g$.

Solution Not Found: Use a numerical root extractor to solve for the x_i, v_i, x_f and v_f in the equations $b_i(x_i, v_i) = 0$; $b_f(x_f, v_f) = 0$; $e_x(x_i, v_i, t_i, t_f, x_f) = 0$; $e_v(x_i, v_i, t_i, t_f, v_f) = 0$, where e_x and e_v both operate by integrating $(x(t), v(t))$ from t_i to t_f starting at the state (x_i, v_i). The function e_x returns $x(t_f) - x_f$. The function e_v returns $v(t_f) - v_f$.

7 Acknowledgments

The research reported in this paper is supported by the National Aeronautics and Space Administration through NASA Grants NCC-2-802 and NAG2-817 and by the Hypercomputing and Design Project (HPCD), sponsored by the Advanced Research Projects Agency of the Department of Defense through contract ARPA-DABT 63-93-C-0064. It has benefited from discussions with Andrew Gelsey, John Keane, Rich Keller, Keith Miyake and Don Smith.

References

[Constable, 1986] R. L. et al. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[Ellman *et al.*, 1993] T. Ellman, J. Keane, and M. Schwabacher. Intelligent model selection for hill-climbing search in computer-aided design. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, D.C., 1993.

[Ellman *et al.*, 1995] T. Ellman, J. Keane, M. Schwabacher, and T. Murata. A transformation system for interactive reformulation of design optimization strategies. In *Proceedings of the Tenth Knowledge-Based Software Engineering Conference*, Boston, MA, 1995.

[Gelsey *et al.*, 1996] Andrew Gelsey, Don Smith, Mark Schwabacher, Khaled Rasheed, and Keith Miyake. A search space toolkit. *Decision Support Systems, special issue on Unification of Artificial Intelligence with Optimization*, 1996. To appear.

[Kant, 1993] E. Kant. Synthesis of mathematical modeling software. *IEEE Software*, pages 30–41, May 1993.

[Keane, 1996] J. Keane. Extensions to the franz, inc.'s allegro common lisp foreign function interface. Technical Report HPCD-TR-41, Department of Computer Science, Rutgers University, 1996.

[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*, 1:79–128, 1994.

[Lowry *et al.*, 1994] M. Lowry, A. Philpot, T. Pressberger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*, Monterey, CA, 1994.

[Manna and Waldinger, 1992] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–705, August 1992.

[Matlab, 1992] *MATLAB: High Performance Numerical Computation and Visualization Software.* The Math Works, Inc., Natick, MA, 1992.

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

[Smith, 1993] D. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 16, 1993.