Automatic Construction of Accurate Models of Physical Systems

Elizabeth Bradley*
University of Colorado
Department of Computer Science
Boulder, CO 80309-0430
lizb@cs.colorado.edu

Abstract

This paper describes some preliminary work on a program that builds ordinary differential equation (ODE) models of target systems from user-supplied hypotheses, observations, and specifications. Its implementation exploits symbolic and qualitative reasoning whenever possible, only resorting to low-level numeric methods if absolutely necessary. ODE theory and domain-specific rules are used to combine hypotheses, to check evolving models against the observations modulo the resolution inherent in the specifications, to remove unnecessary terms, and to synthesize new terms from scratch if need be. This tool has been designed to use sensors and actuators in an *input-output* approach to modeling real physical systems.

Introduction

One of the most powerful analysis tools in existence — and often one of the most difficult to create — is a good model. Expert model-builders typically construct hierarchies of successively subtler representations that capture the salient features of a physical system, each incorporating more physics than the last. At each level in the hierarchy, the modeler assesses what properties and perspectives are important and uses approximations and abstractions to focus the model accordingly. The subtlety of the reasoning skills involved in this process, together with the intricacy of the interplay between them, has led many of its practitioners to classify modeling as "intuitive" and "an art(Morrison 1991)." This paper presents some preliminary ideas about a computer program, currently under development, that is an attempt to automate a coherent and useful part of this art. Such a tool is of obvious practical importance in science and engineering: as a corroborator of existing models and designs, as a medium within which to instruct newcomers, and as an intelligent assistant, whose aid allows more time and creative thought to be devoted to other demanding tasks.

The modeling program described here works with ordinary differential equations (ODEs), linear or nonlinear, with multiple variables; it combines powerful mathematical formalisms with domain-specific notions — Kirchhoff's laws for electronic circuits, for instance, or force balances for mechanics — to allow the type of "custom-generated approximations"(Weld 1992) that are lacking in many existing automated modeling programs. The current incarnation of the program is written in Scheme and Maple; its implementation combines traditional numerical analysis methods, such as simulation and nonlinear regression, with symbolic computation, and will soon incorporate qualitative simulation(Kuipers 1986).

The program builds ODE models from three types of information: mathematical precepts that are true for all ODEs, such as the definition of an equilibrium point; rules that apply in individual domains, such as Kirchoff's voltage law for electronic circuits; and highly specific information about the individual target system, entered by a user. Hypotheses may conflict and need not be mutually exclusive, whereas ODE and domain rules are always held to be true. The ODE and domain rules are currently hard-coded in the program; we ultimately envision incorporating a few dozen of the former in total, plus half of a dozen of the latter for each supported domain, and allowing users to augment and modify both types. The specific information about the target system is presented to this modeling tool in a variety of forms and formats:

- the user's hypotheses about the physics involved
- observations, interpreted and described by the user, symbolically or graphically, in varying degrees of precision
- physical measurements made directly and automatically on the system

Because the physical interface will include both sensors *and actuators*, this tool will be able to take an *input-output* approach to modeling, another novel and powerful feature.

This research project has two goals; one is immediate and concrete, while the other is far-reaching and less well-defined. The first is to create a program that autonomously constructs mathematical models using the

*Supported by NSF NYI #CCR-9357740

same kinds of inputs that a human expert would use. The second is a first cut at "mental modeling"(Bobrow 1984): to understand what matters in a model and what qualitative and quantitative properties are affected by the requirements of the situation and the knowledge of the user.

What the Program Does

A model is an abstraction of someone's understanding of a particular system. It is based on observations, depends critically on the physics background of its architect, incorporates some notion of what quantities matter (scope[1]), and adapts to changing scales (resolution). Because exact descriptions are often unknown and/or inappropriate, modeling almost always involves abstractions and approximations. Refinement increases the *order* of a model, adding terms, sharpening approximations, and shrinking "black box" boundaries. In simplification, the dual of refinement, unnecessary terms are removed. Refinement occurs when a model does not match observations and must be improved. Simplification occurs when a model is more complex than necessary. Both operations are governed by the required accuracy and resolution.

Figure 1 shows the connections between the tool described in this paper and the system to be modeled. The program represented by the right-hand block in this figure constructs an ODE model of a target system based on information entered by a user; if the model is to be based on direct observations of a physical system, the program will obtain additional information via sensors, actuators, a hardware I/O channel, and data acquisition software. The *control vector* $\vec{u}$ represents the actuator inputs. The *observation vector* $\vec{x}$ is a set of variables that represent the system state — or at least parts of it that are observable and/or interesting.[2]

The program's output is an ordinary differential equation, of the form $f(\vec{x},t) = 0$ (or $f(\vec{x},t) = \gamma(t)$, if the system is driven), that matches the observations to within the prescribed resolution. The program builds this model by mapping the domain rules encoded in its knowledge base onto the user's hypotheses, checks it against the observations modulo the precision inherent in the specifications, and refines or simplifies it, as necessary, using a collection of techniques that are outlined in the later sections of this paper.

Figure 2 shows an example of how one might instruct the program to build a model of the damped pendulum. The user first sets up the problem, hypothesizes four different force terms, gives three observations about θ, and specifies the required resolutions. The details and implications of each part of this syntax are covered in the rest of this section; the next section describes how the program uses that information to build a model.

The initial problem setup requires several steps. The first specifies the domain and instantiates its associated rules. `(autonomous <force>)` tells the program to apply all `<force>` rules blindly, and not to require the right-hand side of the model to include any particular forcing function. The next two lines identify `<theta>` as a state variable that is a `coordinate` associated with a `point` (the bob). The modeling process works most efficiently if all of the important state variables are identified in the `state-variables` statement, but the program does incorporate a few techniques, addressed in a later section of this paper, that allow it to construct a model, in some cases, even if the user omits important state variables. *Redundant* state variables — ones that play no role in the model — increase the size of the search space but otherwise present no problems.

Hypotheses are ODE fragments (single terms, currently) whose variables are (1) elements of the observation vector and (2) special keywords that provide the connection to domain and ODE rules. The `<time>` keyword is common to all domains; keywords specific to the `mechanics` domain are `<force>` and `<energy>`, with the associated domain rules `(point-sum <force> = 0)` and `(point-change <energy> = 0)`.[3] The electronics domain uses the keywords `<current>` and `<voltage>`, which work with the domain rules `(point-sum <current> = 0)` and `(loop-sum <voltage> = 0)`. Manipulation of these point and loop constructs places some interesting requirements on the internal representations of `coordinates`, as the program must be able to infer connections, cutsets, etc. Note that the concepts of loop and point sums are not only appropriate for these examples, but also generalizable well beyond mechanics or electronics. Finally, its syntax and setup make the program easily extensible to other paradigms (e.g., `volume-change`, etc.) via simple syntax extensions and new rules that tap into those extensions, much as `(point-sum <force> = 0)` works with hypotheses that include the `<force>` keyword.

Multiple hypotheses about a single effect can — *and should* — exist; the program will automatically deter-

[1]The terminology of (Weld 1992) will be adopted here, identified with sans serif font upon first appearance of each term.

[2]Note that the components of $\vec{x}$ may or may not represent a unique function of the internal system state, and that measuring any given quantity may not be possible (i.e., no appropriate sensor may exist). Observability, controllability, and reachability issues will not be addressed here.

[3]A body-centered inertial reference frame is assumed here, together with coordinates that follow the formulation of classical mechanics(Goldstein 1980), which assigns one coordinate to each degree of freedom, thereby allowing all equations to be written without vectors. A *conjugate momentum* is associated with each coordinate; this concept is useful in symmetry identification and symbolic model matching.

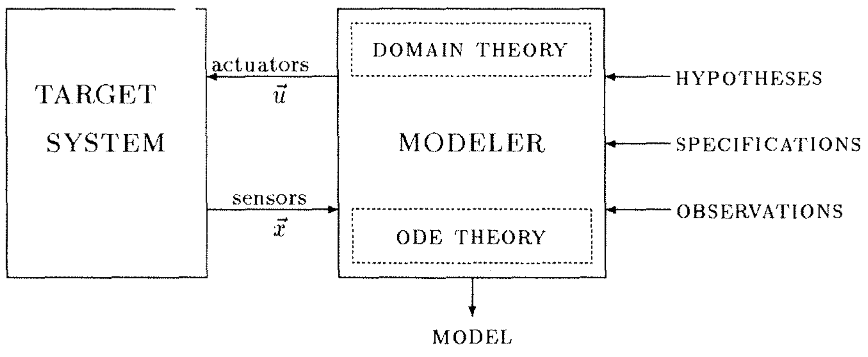
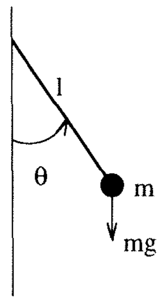


Figure 1: Structure of the modeling tool



```
(find-model
        (domain mechanics)
        (autonomous <force>)
        (state-variables (<theta>))
        (point-coordinate <theta>)
        (hypotheses
             ((<force> (* (constant A1 ()) (deriv (deriv <theta>))))
             ((<force> (* (constant A2 ()) (sin (* (constant A5 ()) <theta>)))
             ((<force> (* (constant A3 ()) (deriv <theta>)))
             ((<force> (* (constant A4 ()) (square (deriv <theta>)))))
        (observations
             ((<theta> (linear (1 0) <theta> (range -.05 .05)))
             ((<theta> (asymptote (eqn 0)
                                  (at <time> *infinity*)
                                  (range 0 *infinity*)))
             ((<theta> (numeric (<theta> <time>) ((0 .1234) (.1 .1003) ...  ))))
        (specifications
             (mesh-width <time> absolute 1e-6 (0 120))
             (mesh-width <theta> absolute 1e-3 (0 (* 2 pi)))))
```

Figure 2: Instructing the program to model the damped pendulum

mine which one is appropriate. Some other modeling programs, e.g., (Falkenhainer & Forbus 1991), define *groupings* of terms in such situations, such as a set of hypotheses about friction. Mutual exclusivity constraints are then imposed within each group, greatly reducing the complexity of the refinement process. We have chosen *not* to use such groupings, for two reasons: (1) to minimize the restrictions on the models that the program can choose and (2) to minimize the high-level conceptual processing required of the user.

Observations describe the behavior of a single element of the observation vector, either in the time domain or in any state-space projection.[4] Unlike hypotheses, observations may not conflict. They have two potential sources: the user and the sensors. User observations may be *descriptive*, *graphical*, or *numeric*. The former use special descriptive keywords, the second are sketches drawn on a computer screen with a mouse, and the third simply specify data points. Descriptive keywords — **`concave, monotonic, oscillation, linear with [slope, intercept]`**, etc. — closely resemble terms in qualitative physics(QP)(Weld & de Kleer 1990). The user's sketches will be processed — curve fitting, interpolation, recognition of linear regions, and so on — using Maple functions, and the results will be used in the same way as descriptive observations. Observations from the hardware I/O channel will be treated much like graphical observations, but at a higher confidence level. Finally, observations of any form must encode the range in which they are valid; the endpoints of these ranges are akin to QP's *landmarks*.

Observations guide the modeling process in a fundamental way. A model constructed by a human expert matches, minimally, a particular set of observations: the model builder does no more work than necessary to effect the match, and does not try to anticipate extensions or further developments until forced to do so by model failure or requirement escalation. The automatic modeler described here does exactly the same thing: at all times, the program attempts to establish the match with the minimum of work, using information at as high a level as possible to do so. The most important implications of this concern the determination of the coefficients `A1, A2`, etc. in figure 2. A descriptive observation often places only qualitative requirements or bounds on those coefficients, while matching a model against a detailed numeric observation usually requires exact coefficient values out to some number of significant figures. Moreover, a single observation, even a qualitative one, can contain information about many different variables — if, for example, it concerns a state variable that appears in all of the terms in the model. Such an observation would significantly focus the model by forcing the evaluation of several coefficients in several different parts of the differential equation. Thus neither the number nor the characteristics — qualitative, quantitative, etc. — of observations required for the construction of a successful model are necessarily related to the number of undetermined coefficients or state variables; these requirements are complicated and possibly nonlinear functions of the terms involved. Finally, a higher-level and more obvious implication of the role of observations in the modeling process is that neither a program nor a human expert can construct a model if no observations are given.

A *specification* concerns any function of any number of observation vector elements; it prescribes range and resolution limits for that quantity and specifies whether the resolution is absolute or relative. The **`mesh-width`** statements in figure 2, for instance, instruct the modeler to impose microsecond and milliradian accuracy over 120 seconds of system evolution. All observation processing is based on the ranges and resolutions given in the specifications; if only the range x_2=[3, 5] is of interest, asymptotes at $x_2 = 40$ are immaterial. Landmarks that overlap can be combined, and any effect that occurs on a smaller scale than the one specified can be subsumed into the intervals around it (e.g., a nanosecond glitch during a millisecond-resolution run). It is important to note that specifications implicitly govern the level of abstraction that the modeler enforces: sharpening the **`mesh-width`** will typically force the modeler to account for lower-level effects and add terms to the ODE, given a fixed set of observations.

This set of inputs was consciously chosen to mimic the information that an expert designer uses when he or she constructs a model. The motivation for this choice is that, in a project whose secondary aim is to understand the human problem-solving process, the program should emulate the process that a human expert would follow, insofar as possible. Moreover, a tool designed for human use should interface smoothly with human skills, reasoning and communication patterns. This choice, and its justification, are specific to this particular project. The debate about whether or not computer problem-solving processes should *in general* emulate their human equivalents has a long and somewhat contentious history, to which we plan no contribution.

How The Program Works

The program encodes a body of general knowledge about ODEs — how to recognize, locate, and quantify equilibria, basins of attraction, integrability, periodicity, etc. — and domain-specific knowledge like `(point-sum <force> 0)`. Together, the ODE and domain rules, operating on the inputs described in the previous section, govern how hypotheses and models are combined, tested, ruled out, augmented, and simplified. Throughout the process, tasks are performed

[4]This program will perform no frequency-domain reasoning.

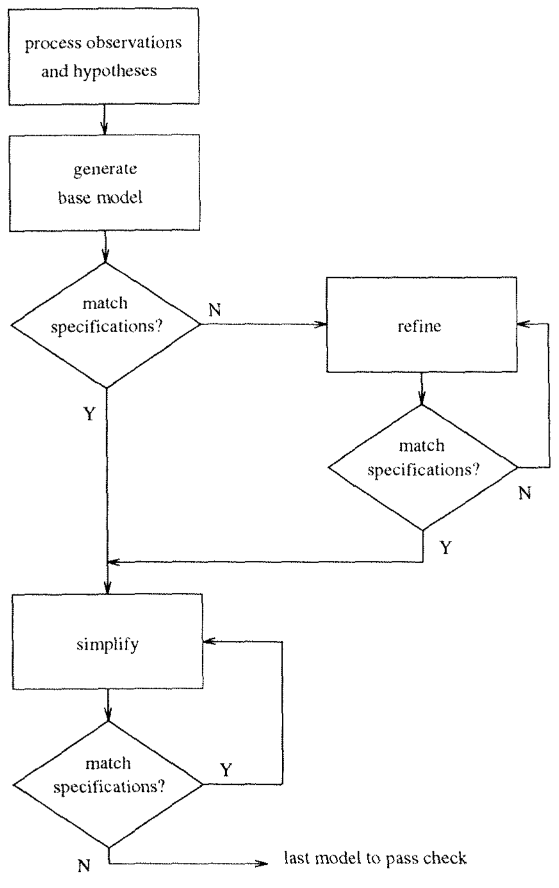


Figure 3: Flowchart of the modeling program

at the highest, most qualitative level possible; symbolic techniques are attempted first and numeric ones only as a last resort. Finally, the modeling process is not passive: the program can use its actuators to manipulate the system and *actively test* the evolving model.

A set of candidates for the program's first attempt at a model is constructed by mapping the domain rules onto the hypotheses — e.g., using `(point-sum <force> 0)` to combine `<force>` = $A_1\ddot{\theta}$ and `<force>` = $A_2 \sin A_5\theta$ into the model $A_1\ddot{\theta} + A_2 \sin A_5\theta = 0$. Most of the resulting candidate models can quickly be ruled out using rough symbolic rules and techniques; for example, unless a state variable is observed to be constant, the model must include its derivative. The simplest remaining model — the *base model* — is then passed to the check/refine/simplify loop, as schematized in figure 3. This model is checked against the specifications and observations; if it passes, the program then attempts simplification, removing each term in succession and checking the resulting ODE. If the consistency check fails, the program *refines* the ODE, using the domain rules to introduce successive hypotheses from the user's list. As a last resort, the refiner calls upon general ODE theory and purely mechanical methods, such as power-series expansion, to synthesize terms from scratch. The refined model undergoes one last round of simplification that finds and removes any newly-superfluous terms, and the final product is returned to the user.

This control flow design has a variety of interesting implications. The simplify/refine loop allows the program to move sideways through the search tree of models, recovering from bad choices and making globally good moves that require *one* locally bad intermediate step. We could also have chosen to loop more than once, which would increase the width of the program's lateral reach through the search space, and, ultimately, allow it to find the *provably minimal model* in that space. However, such a search would have the standard complexity problems(Winston 1992) and a secondary goal of this project is to produce a "good enough" answer in minimal time.

The four blocks of figure 3 — and the mechanics and implications of failure in each — are described in the next four subsections. The following section describes the as yet unimplemented hardware interface and the mechanics of input-output modeling, and is followed by a brief discussion of related work.

Generating the First Model

The base model generator uses domain rules to combine hypotheses, then ascertains which of those combinations (models) are consistent with the user's observations. The idea here is to produce a preliminary solution to an exponentially complex problem very quickly; to do so, the base-model generator uses *only* symbolic manipulation and stops short of an exhaustive check of all possible combinations of hypotheses. When the answer that it produces is suboptimal or even incorrect — a not-unlikely occurrence because of its quick-and-dirty techniques — the refiner and simplifier act as a safety net, as described in the penultimate paragraph of the previous section.

The hypothesis list is first sorted using a simple-minded complexity metric, discussed later in this paper. The base-model generator then constructs a one-term model from the first hypothesis in the list and checks it against the observations. Symbolic linear algebra, for instance, can be used to establish that the candidate model $A_1\dot{\theta} = 0$ is inconsistent with an observed initial condition $\theta(0) = 1.2$ and an equilibrium point $\theta(\infty) = 0$. If the check fails, the process is repeated using the next entry in the hypothesis list, and so on. If none of the one-term models passes, the base model generator then starts testing two-term combinations.[5] If all two-term models fail, the program proceeds to three-term models, and so on. The first ODE in this succession that is not implausible, given the observations, is then passed to the consistency checker.

[5]Note that *combination* need not imply *sum*; the combination operator is implicit in the operative domain rule.

Checking Models Against Observations

The consistency checker compares the behavior of an ODE to a set of observations using the specifications as guidelines for how closely to enforce the match. When operating upon descriptive or graphical observations, the consistency checker reasons symbolically when possible, falls back on numerical simulation if necessary, and recognizes different types of behavior (e.g., chaotic, periodic) in both numeric and symbolic results. With numeric observations, the check proceeds directly to the numeric phase.

As mentioned previously, the evolving precision of the model's coefficients follows the procession of the ODE through the phases of the consistency check. If a symbolic check is coefficient-independent, none of the `Ai` need be determined. If the check proceeds all the way to a point-by-point comparison of Runge-Kutta output to the data in a `numeric` observation, data-point interpolation and Newton-Raphson are used to compute all undetermined coefficients. Many cases will fall between these two extremes, yielding partially determined coefficients. For example, to match a model against an observed oscillation, the program might symbolically construct a root-locus plot and determine the coefficient values that cause the system poles to cross into the right-half plane — e.g., `(A1 greater-than 0)`.

The symbolic techniques used to check a model against a qualitative observation are similar to those used by the base model generator — in fact, the latter are a subset of the former. For instance, if the observations included a description of an asymptote, the consistency checker would use symbolic differentiation and L'Hôpital's rule to take the appropriate limit and corroborate the observation. Regions where the system is linear or where a particular polynomial has been specified — or found by a curve-fitting procedure — can be checked using the first few terms of a symbolically constructed Taylor series. The QSIM program(Kuipers 1986) performs qualitative simulation of *qualitative differential equations* (QDEs); given an ODE↔QDE translator(Crawford, Farquhar, & Kuipers 1990), QSIM could be used to do some of these first-cut symbolic consistency checks.

A numeric consistency check of a model entails a comparison of a mechanical integration of the ODE, computed using Runge-Kutta(Press *et al.* 1988), and a set of point-by-point observations. As mentioned above, this comparison is preceded by a numeric fit of model coefficients to numeric data; if it was invoked by a non-numeric observation, the check may entail geometric recognition of high-level behavior patterns — such as chaos, thresholds, or periodic orbits — in the simulation data. One easy — and proven(Bradley 1992) — way to perform this type of geometric recognition is to superimpose a variable-resolution grid over the system's state space, discretize the trajectories into lists of grid squares, and analyze the patterns in those lists. This ties in well to the input format proposed earlier: specifications and observations can be used to set up the parameters of the mechanical integration and the geometry of the grid, assuring adequate accuracy *and resolution* in the results, attained with the minimum computation.

If any model fails a check, a failure report is generated; this report contains the model, the violated observation(s), the data (qualitative or quantitative) that caused the violation, perhaps a simple interpretation derived from those data, and *the model's full genealogy* — all branches in the search tree that were traversed during the search. This information could be used by the other program modules — a form of *discrepancy-driven refinement*(Addanki, Cremonini, & Penberthy 1991; Weld 1992) — in a variety of ways: to back intelligently out of blind alleys, to avoid duplicating previously performed checks, or to pick up at some appropriate midpoint in the event of a restart.

Simplification: Reducing a Model to Lowest Order

The simplifier identifies a candidate term for removal, deletes it, and then invokes the consistency checker on the remaining ODE. If the check fails, the simplifier replaces that term and removes another. This process is repeated until some $(n-1)$-term "sub-model" of the n-term model passes the check, or until all $n-1$-models have been tested. In the latter case, the model cannot be further simplified and is passed on as a final result. If any sub-model does pass the check, the entire process is repeated, testing all $n-2$-term subsets of that ODE, and so on.[6]

Like the consistency checker, the simplifier takes a symbolic-first approach to identification of candidate terms for removal. It uses some of the same symbolic techniques as do the base model generator and the checker; the main area of overlap is in ODE theory (e.g., asymptote recognition and symbolic differentiation). It also uses some symbolic techniques that would not be useful in the checker, such as pole-zero analysis and conjugate momentum/symmetry recognition. If symbolic reasoning cannot identify a good candidate for removal among the terms in the ODE, the simplifier then turns to less-intelligent techniques, such as removing the "simplest" terms (see below), or terms whose coefficients are much smaller than the others in the model. These choices are somewhat *ad hoc*; it is not at all clear that we gain much from either beyond a defined order of attack. Another approach would be to give the user some leverage by weighting hypotheses according to order of entry. Each of these tactics has advantages and disadvantages; each shifts a different

[6]This raises a subtle issue: sometimes adding or removing *pairs* of terms (or larger groupings) works where successive single-term removal does not: consider coriolis and centripetal forces, which must appear together in a rotating frame. We plan to investigate this in the near future.

amount of the burden of rigor from the simplifier to the checker.

Ordering terms or hypotheses logically is difficult because no satisfying simplicity metric exists. The modeling literature contains no good solutions, and this author has no better suggestions. The somewhat-arbitrary metric that we use is based on number and complexity of terms and derivatives: $x + 3 \rhd x; x^n \rhd x^{n-1}$; $\dot{x} \rhd x$; etc., where $\rhd$ denotes "more complex than." This is unsatisfying for a variety of reasons, not the least of which is how to resolve ties between equations like $x + x^2$ and $x^3 + 4$. One of the seminal papers in qualitative modeling uses a similar metric(Falkenhainer & Forbus 1991) — and gives similar disclaimers about its unsatisfying nature. A more-recent paper(Nayak 1992) defines model simplicity heuristically, terming a model that is "more approximate" or that "uses fewer phenomena" as more simple.

Refinement: Adding Terms to a Model

The refiner uses ODE and domain rules, observations, and hypotheses to add terms to a model, keeping track of previous attempts in order to avoid duplication of effort. It first draws upon the user's hypotheses, then synthesizes ODE terms from scratch using power-series expansions if no successful hypothesis-based model can be found. These power-series methods are a powerful safety net: a one- or two-term expansion in θ and $\dot{\theta}$ would regenerate every form of friction found in freshman physics texts — and a few more besides. Moreover, these methods actually allow the program to *create new state variables* — an important feature if the observation vector is smaller than the true state vector. Like the other program modules, the refiner uses physics concepts before less-intuitive mechanical techniques and follows symbolic reasoning with more-expensive numeric approaches.

The refiner's first task is to prune the sorted hypothesis list, removing any that were used in the base model. The first hypothesis in the sorted list is then added to the base model and the checker is be called on the new ODE. If that model fails, the refiner removes the previously introduced term and successively tries the rest of the hypotheses in the ordering, one at a time. If no one-term addition causes the model to match the observations, the refiner then tries *pairs* of terms, and so on. The first successful model is returned as the refiner's result. If the refiner exhausts the list of hypotheses before finding an adequate model, the power series methods outlined in the previous paragraph are invoked. These expansions are subject to a user-specified limit p; if no match has been established after the refiner tries the p^{th} term in the expansion, the program admits failure and requests additional hypotheses from the user.

Simplicity metric difficulties, discussed at the end of the previous section, also affect the refiner. The metric proposed here may not be rigorous, but it at least provides a starting point. A much more intelligent approach would be to sort the hypotheses according to their behavior, using the same symbolic and numeric techniques used in the checker to make a rough comparison to the observations, all modulo specification precision. For instance, if an observed voltage converged to zero, a hypothesis that caused that value to diverge to infinity should be lower on the list of things to try. Note that such a hypothesis should *not* be removed from the list altogether; it may well appear in the ultimate model, moderated by another term (e.g., $x - \frac{x^3}{6} + \frac{x^5}{120} \ldots$). This *sorting by behavior assessment* may be computationally expensive enough to negate its advantages. Moreover, serious problems can arise when one attempts to extend conclusions about the parts of a system to conclusions about its whole (e.g., closed-loop versus open-loop dynamics in a feedback system). However, the advantages of ruling out some of the hypotheses may dominate, making this approach attractive. Another possibly useful alternative would be, again, to use the entry order verbatim. A half-dozen informal interviews have shown that most users do indeed list physics hypotheses in order of perceived simplicity,[7] so this may not be a bad idea.

The two automatic term-synthesis techniques to be used here are both derived from power-series expansion. *Canonical perturbation theory*(Goldstein 1980, chapter 11) creates new parameters and *Frobenius's method*(Morrison 1991, page 187) creates new state variables, each via expansion in the appropriate quantity. One could also synthesize new state variables using delay coordinates(Gouesbet & Maquet 1992), but doing so would vastly complicate the symbolic algebra. Because the lower-order terms of power-series expansions are likely to match (up to a coefficient) simple physics hypotheses, the modeler will use simple symbolic techniques to avoid duplication of effort, eliminating any power series expansion terms that also appear in a user hypothesis.

The combination of all of this machinery allows the program to preferentially check the user's hypotheses, then synthesize and explore new state variables and parameters if necessary, possibly concocting model terms that do not resemble any of the given hypotheses. It can even construct models in the absence of *any* hypotheses, if the underlying physics admits a power-series description. The ODE and domain rules, together with symbolic reasoning and qualitative behavior descriptions, will allow this tool to reason at a very high level indeed. For example, if a system was observed to be chaotic, but only one state variable was specified, the power series methods would be invoked automatically to introduce a new state variable before the program even attempted to build a base model.[8]

[7]The exception is serious experts, who sometimes add the simplest terms only as an afterthought.

[8]Three state-space dimensions are a necessary condition

This automatic modeling tool will be able to adapt the *scope* of the model on the fly, inferring "state variables" that are internal (e.g., voltages inside a black box) or that were omitted by oversight (e.g., low-amplitude, low-frequency vibration of a bench by a lab's HVAC system). These quantities are diffeomorphically related to the true system state, so they can be used to draw some rigorous conclusions. Again, formal observer theory is not a goal of this project, so the proposed program will not be able to solve this problem in all situations, and this research will include only limited study of the relationships between inferred and true state variables.

Incorporating Physical Measurements

An important feature of the physical link between modeler and system is that data will be able to flow across it in *both* directions, making the modeling process an *active* one — in all parts of the procedure described in the previous three sections. This has a number of important implications. Among other things, the program could autonomously exercise the target system in order to verify or augment the user's observations; it could even construct and check observations that transcend its user's knowledge of physics. For instance, the boundaries of the basins of attraction of the ODE and the target system could be compared using different-energy kicks, even if the user knew nothing about dynamics and basin structure. Obviously, this presents some dangers: if the program is free to use the sensors, the actuators, and the full breadth of its own (significant) knowledge base, the only controls on the sophistication — and intricacy — of the resulting model are the resolution (`mesh-width`) and the expansion limit p. To address this problem, we have introduced another parameter, `max-level`, that allows the user explicit control over the number of terms that the program may use in the model.

Physical measurements will be treated exactly like graphical user observations, from the program's point of view, with one important distinction: in the event of a conflict, the former will be trusted over the latter. Measurements will be processed and translated into the syntax of descriptive user observations and then used in exactly the same ways — as targets for symbolic comparisons. The measurement processing algorithms that extract this qualitative information from the sensor data will essentially duplicate the approach to behavior recognition outlined in the discussion on checking models against observations.

With the hardware data channel in place, the program could potentially be used not only to model a physical system, but also to debug designs — and even to validate and verify devices ostensibly constructed according to a known design. For example, if a particular 2.5KΩ, 1/4 Watt resistor burned up every time a device was turned on, one could place probes across its terminals, set up an observation that specifies "voltage over 25 volts," and observe discrepancies between the resulting model and what was intended. Needless to say, one would want to isolate the sensors from potential damage during such an experiment.

for chaos.

Related Work

Modeling research spans many fields, from the cognitive science-related branch of AI(Langley *et al.* 1987) through dynamic systems(Gershenfeld & Weigend 1993) and control theory(Astrom & Eykhoff 1971) to qualitative reasoning (QR)(Bobrow 1985). Some of the earliest QR/modeling work(Falkenhainer & Forbus 1991) built upon a fixed base of hypotheses, instantiated only those that were appropriate to answer a given query, and chose between them with a truth maintenance system. The GoM approach(Addanki, Cremonini, & Penberthy 1989) is similar in many regards to (Falkenhainer & Forbus 1991), but represents the space of possible models as a directed graph of models where edges between nodes (models) are approximations.[9] Another approach(Weld 1992) adapts models to problems using *model sensitivity analysis*, which formalizes and exploits the effects of parameter changes in the construction of the model. The combinatorial explosion involved in limit checking (e.g., the pendulum's asymptote to $\theta = 0$) can be mitigated(Kuipers 1987) by decomposing and abstracting time into a hierarchy of scales. Rules for determining the behavior of a composite device can be derived from the models of its constituent components(Kleer & Brown 1984).

This is an extremely active research area; many good papers by many other groups, as well as many other papers by the cited groups, have appeared in the past four years. The state of the art in this field is particularly well-summarized in a recent article by Weld(Weld 1992), which is also the source of much of the terminology used in this document. Concepts common to this paper and the bulk of the QR/modeling literature include avoidance of unnecessary terms, model refinement driven by failure at a "lower" modeling level, and reasoning that proceeds at as a high level of abstraction as possible.

Status and Discussion

This paper is a description of the first stages of work-in-progress on a highly ambitious task. To date, we have only built a small, but functional — and hopefully representative — subset of the program.[10] This subset incorporates a few instances of each technique, providing a quick check of the whole symbolic/numeric

[9](Falkenhainer & Forbus 1991)'s propositional reasoning also uses a digraph, but it is used implicitly and constructed somewhat differently.

[10]hence the mix of verb tenses in this paper.

paradigm and the overall control flow (hierarchy of hypotheses, domain rules and ODE rules; control flow between modules, and so on). Most importantly, it has allowed us to test and refine the syntax and the use of the various types of user inputs.

This first version, constructed[11] with the aid of Reinhard Stolle, allows one-term hypotheses involving the keywords `<force>` and `<time>`, sorts them according to the simplistic metric proposed in a previous section, incorporates one rule (`(point-sum <force> 0)`) in one domain (`mechanics`), has a reduced vocabulary of qualitative terms (`above, below`) and a correspondingly small repertoire of symbolic and numeric techniques to verify them, and has been tested on two-dimensional systems with numeric observations, such as figure 2 with all but the `numeric` observation omitted. Even this limited exercise has turned up some interesting problems. Determining the coefficients for the numerical integrator run required linear interpolation of the data points to produce m nonlinear equations in the m unknown `constants`, which were then solved with Newton-Raphson. In general, this problem — known as *parameter estimation* — is solved with much more sophisticated techniques like Kalman filtering(Kalman 1960), and is the topic of a rich body of literature(Sorenson 1985). Our simplistic solution spawned a second problem: a failure to match a Runge-Kutta run against an observation could mean *either* that the model was inadequate *or* that our parameter estimation algorithm was inaccurate. We solved this by imposing an artificially high accuracy on the latter, but this is not a good general solution. The `mechanics` domain syntax has grown far closer to the Lagrangian formulation than we had originally envisioned because its formalized structure is so useful (for example, the coordinate/momentum pair for each degree of freedom, related by a derivative, that suffice to describe the system completely and contain *symbolically extractable* information about symmetries and conserved quantities as well).

Figure 4 depicts a modeling run on a simplified version of the pendulum of figure 2, shown at the top of figure 4, that includes only two hypotheses and one numeric observation. The base-model generator is bypassed in this run because the only observation is numeric. The refiner rules out all one-term models via symbolic methods, then uses the force-balance rule to map the two hypotheses into a two-term candidate model, upon which the checker is invoked. The check proceeds directly to the numeric phase, the parameter estimator computes the coefficients, and the point-by-point comparison succeeds. Since both "sub-models" of this model have already failed the check, this model and its coefficients are returned as the final result. We have also experimented with adding white noise to the numeric data and changing the resolution in the specifications, with predictable results. The program still finds the right model if the added noise is small compared to the resolution, and fails when it is not, as the repertoire of hypotheses does not allow it to model the noise.

Later versions will use the hardware I/O channel, incorporate more symbolic and qualitative methods, be tested with underspecified modeling tasks and sketchier observations, and use more domain rules in multiple domains that operate on multiple, heterogeneous keywords. We expect the first and the last to be the hardest and most interesting of these tasks, particularly since our aim is emphatically not to build a tool that is tuned for one particular application domain. Beyond that, further development will entail the exploration of different implementation paradigms and techniques: sorting hypotheses by behavior or by order of entry, allowing multi-term hypotheses, etc.

Summary

The nascent program described here uses general mathematical theory as a foundation, adds concise and powerful domain-specific rules, and funnels user-specified hypotheses through general ODE theory and domain-specific rules to generate "appropriate" models — models that are well-matched to the task at hand. It exploits intelligent, high-level techniques like symbolic manipulation from the outset, carrying them as far as possible through each phase of the work, and using them to make the type of quick, overall assessment that a human expert uses in the first stages of model-building. The program then resorts to lower-level, less-intuitive methods to complete the analysis and synthesis processes. The chosen set of inputs and the way that they are used closely resembles the process one finds documented on any designer's scratch paper: parts of equations, rough sketches, scratched-out forays up analytical blind alleys, and an overall progression of ideas and abstractions from simpler to more complex.

This mixture of exact and approximate techniques and precise and heuristic knowledge is powerful, but has one important disadvantage: it makes formal, rigorous analysis of the necessary conditions for modeling success very difficult to come by. This is not a pure mathematician's tool, though pure mathematics certainly contributed to the plan for its implementation. It is a design tool that is intended to be *useful*: to find a "good enough" solution with minimum time and effort. This tradeoff motivated many of the design choices described here, notably the control flow between the simplifier and the refiner.

Though the examples here are drawn from the domains of mechanics and electronics, many other potential application areas exist. The general framework described here has been designed to smoothly accommodate rules from many domains, and the program itself has been written to be easily extensible. It should

[11] both code and ideas

The invocation:

```
(find-model
        (domain mechanics)
        (autonomous <force>)
        (state-variables (<theta>))
        (point-coordinate <theta>)
        (hypotheses
             (<force> (* (constant a ()) (deriv (deriv <theta>))))
             (<force> (* (constant b ()) (sin <theta>))))
        (observations
             (<theta> (numeric (<theta> <time>) ((0 .1234) (.1 .1003) ...  ))))
        (specifications
             (mesh-width <time> absolute 1e-6 (0 120))
             (mesh-width <theta> absolute 1e-3 (0 (* 2 pi)))))
```

The transcript:

```
Trying to find model for
hypotheses = ((* a (deriv (deriv <theta>)))
              (* b (sin <theta>)))
with max level = 2.

Trying to find model at level 2...
Checking model
  (model ((= (+ (* (constant b ()) (sin <theta>))
                (* (constant a ()) (deriv (deriv <theta>)))) 0))).
Checking model
  (model ((= (+ (* (constant b ()) (sin <theta>))
                (* (constant a ()) (deriv (deriv <theta>)))) 0)))
numerically.
Checking
  ((= (deriv <theta>) d<theta>)
   (= (deriv d<theta>) (/ (- 0 (* 3.  (sin <theta>))) 2.)))
against data.

((model
  ((=
   (+ (* (constant b ()) (sin <theta>))
       (* (constant a ()) (deriv (deriv <theta>))))
   0)))
  ((a 2.)  (b 3.)))
```

Figure 4: A modeling run on the damped pendulum

be obvious that the definitions and solutions presented here are preliminary and will necessarily undergo much development and refinement as this program is developed, but the preliminary results have been encouraging — and some of the early problems have been subtle, rich, and rewarding to think about and to solve.

References

Addanki, S., Cremonini, R., and Penberthy, J. S. 1989. Reasoning about assumptions in graphs of models. In *Proceedings IJCAI-89*. Detroit, MI.

Addanki, S., Cremonini, R., and Penberthy, J. S. 1991. Graphs of models. *Artificial Intelligence* 51:145–178.

Astrom, K. J., and Eykhoff, P. 1971. System identification — a survey. *Automatica* 7:123–167.

Bobrow, D. G. 1984. Qualitative reasoning about physical systems: An introduction. *Artificial Intelligence* 24:1–5.

Bobrow, D. G., ed. 1985. *Qualitative Reasoning about Physical Systems*. Cambridge MA: M.I.T. Press.

Bradley, E. 1992. *Taming Chaotic Circuits*. Ph.D. Dissertation, M.I.T.

Crawford, J. M., Farquhar, A., and Kuipers, B. J. 1990. QPC: a compiler from physical models into qualitative differential equations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*.

Falkenhainer, B., and Forbus, K. D. 1991. Compositional modeling: Finding the right model for the job. *Artificial Intelligence* 51:95–143.

Gershenfeld, N. S., and Weigend, A. S. 1993. The future of time series. In *Time Series Prediction: Forecasting the Future and Understanding the Past*. Santa Fe, NM: Santa Fe Institute Studies in the Sciences of Complexity.

Goldstein, H. 1980. *Classical Mechanics*. Reading MA: Addison Wesley.

Gouesbet, G., and Maquet, J. 1992. Construction of phenomenological models from numerical scalar time series. *Physica D* 58:202–215.

Kalman, R. E. 1960. A new approach to filtering and prediction problems. *J. Basic Eng.* 82D:35–45.

Kleer, J. D., and Brown, J. S. 1984. A qualitative physics based on confluences. *Artificial Intelligence* 24:7–83.

Kuipers, B. J. 1986. Qualitative simulation. *Artificial Intelligence* 29:289–338.

Kuipers, B. J. 1987. Abstraction by time scale in qualitative simulation. In *Proceedings AAAI-87*, 621–625. Seattle, WA.

Langley, P., Simon, H. A., Bradshaw, G. L., and Zytkow, J. M., eds. 1987. *Scientific Discovery: Computational Explorations of the Creative Process*. Cambridge, MA: MIT Press.

Morrison, F. 1991. *The Art of Modeling Dynamic Systems*. New York: Wiley.

Nayak, P. 1992. Causal approximations. In *Proceedings AAAI-92*.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 1988. *Numerical Recipes: The Art of Scientific Computing*. Cambridge U.K.: Cambridge University Press.

Sorenson, H. W. 1985. *Kalman Filtering: Theory and Application*. IEEE Press.

Weld, D. S., and de Kleer, J., eds. 1990. *Readings in Qualitative Reasoning About Physical Systems*. San Mateo CA: Morgan Kaufman.

Weld, D. S. 1992. Reasoning about model accuracy. *Artificial Intelligence* 56:255–300.

Winston, P. H. 1992. *Artificial Intelligence*. Redwood City CA: Addison Wesley. Third Edition.