

Code Synthesis for Mathematical Modeling

Elaine Kant

Schlumberger Laboratory for Computer Science

P.O. Box 200015

Austin, Texas 78720-0015 USA

kant@slcs.slb.com

INTRODUCTION

Scientific computing involves both creativity on the part of the human scientist and a great deal of mechanical drudgery that can and should be automated. In the domain of mathematical modeling, problems can be specified naturally and concisely in terms of the mathematics and physics of the application. Our goal is to minimize the time required for scientists and engineers to *implement* these mathematical models. Much of the necessary implementation knowledge is available in books and journal articles and can be encoded in a knowledge-based program synthesis system. SINAPSE is one such system that illustrates how to have the scientist or engineer provide the major design decisions for problem solving and have an automated assistant carry out the details of coding the algorithms into the desired target language. The basic implementation paradigm is program transformation based on object-oriented representations of the underlying mathematical and programming concepts. *Mathematica* [Wolfram 88] is the implementation platform.

The SINAPSE system focuses on the generation of finite difference programs (in Fortran, Connection Machine Fortran, and C) from mathematical models described by partial differential equations. Because our primary goal is to provide scientists and engineers the freedom from learning details of many new target hardware and languages, the bulk of the SINAPSE system concerns the automatic generation of an efficient implementation from modeling equations. Although we do not focus on model formulation, the system *can* help users formulate problems if they use standard classes of governing equations. Alternatively, a model specification could come from an automated model formulation system. Because we use a symbolic manipulation language as an implementation platform, the scientist can perform some analytic problem solving to (re)formulate the specification equations if they do not fit one of our standard classes. The use of a symbolic-manipulation implementation language also makes it particularly easy to specify the problem, refinements, and optimizations for the mathematical modeling programs. Once the model is formulated, SINAPSE can guide the user through the design decisions for the algorithms that the system can generate. There is also a history mechanism for recording the design decisions,

which enables the modeler to quickly experiment with alternative models and algorithms (by modifying the specifications and letting the system reimplement).

The SINAPSE system has successfully synthesized on the order of a dozen finite difference programs that have proved useful to Schlumberger scientists and engineers. From these codes, the modelers have learned new facts about their application domain. For example, the system has been used to generate Connection Machine (CM) Fortran code for sonic modeling (to explore dipping beds and other complex formations), to generate CM Fortran code for exploring models of tube waves on axisymmetric geometries (for cross-well seismics), to generate Fortran77 code to measure the propagation transit time of sonic waves in a moving fluid (for a novel ultrasonic flow meter), and to generate Fortran77 code for a poroelastic hydraulic fracture model. In each case study, SINAPSE generated dozens of variations which were tested by the end users for suitability. The modelers did not always know in advance, for example, whether a particular boundary technique that was adequate for a 2D problem would have sufficiently low reflectivity in 3D.

The specifications for the programs SINAPSE has generated average about 50 lines. From these specifications, between 200 and 5000 lines of target language code are generated in up to ten minutes on a Sun SPARCstation 2 (the average time is about three minutes). Most of the synthesized programs have been in Fortran because of user preferences. However, SINAPSE has also produced a few C and C++ programs. The CM Fortran code is not highly optimized, but by hand-tuning less than 10% of the generated code we were able to achieve code that takes only 4 seconds per time step to simulate propagation of a wave through a 64x64x128 grid.

PROBLEM-SOLVING FRAMEWORK

The SINAPSE system is based on the problem-solving framework for mathematical modeling summarized in Figure 1. For a more detailed description of an earlier version of this framework, see [Kant et al. 91]. For related work, see the discussion in that article and other papers in this proceedings.

Corresponding to this problem-solving framework, a SINAPSE specification has three main sections: the ba-

From: AAAI Technical Report FS-92-01, Copyright © 1992, AAAI (www.aaai.org). All rights reserved.

requirements description (independent of the implementation concerns), those implementation decisions that the modeler does not want to leave to the automatic system, and decisions that affect runtime handling (if desired). The user can supply the specification as a text file or can answer questions interactively using menus and forms.

The specification is subdivided into levels moving from abstract problem description through concrete design details. The basic requirements consists of a goal description, a symbolic model (optional), a mathematical model, and target environment properties. A mathematical model consists of equations, a physical interpretation, a geometric model, and a classified model. The implementation stage consists of the algorithm and program levels. At the program level, the specification has been translated into an intermediate array-level language with parallel constructs. By writing directly in this language, a user can bypass the domain-specific requirements specification and specify arbitrary programs. The remainder of the system translates this representation into a conventional target language program that the modeler can carry away. However, SINAPSE can also mediate the execution phase, which involves compiling and running the program (including parameter initialization) to produce the desired results. The geometric modeling interface may also be involved in such mediated execution. Although the current version of SINAPSE addresses each level of problem-solving to some degree, the focus is on supporting the transformations from mathematics to code; not all of the capabilities described in the goal, classified model, target properties, and geometric model levels have been implemented.

A Goal of a program synthesis session is something like "compute pressure over an entire field with second order accuracy." A goal has an action (such as storing or plotting) on an expression involving the quantities being solved for; the action is computed over some range of the given quantities. There can also be performance constraints. A more complex goal would be to do a parametric study of such computations while varying the geometry in a certain way, minimizing overall turn-around time while maximizing accuracy.

In applications (such as wave propagation or heat flow) for which the system has sufficient domain knowledge, the modeler can specify a Symbolic Model by using keywords such as "elastic wave propagation in 3 dimensional Cartesian coordinates with absorbing boundaries." (This is done with menu choices, not natural language text.)

If the user selects a standard formulation, SINAPSE applies domain knowledge and symbolic manipulation to derive the equations in the Mathematical Model. Alternatively, the user can build up the set of interior and boundary equations in an augmentation of *Mathematica's* equation notation. Named domains of equation applicability can be included in the specification. The Physical Interpretation optionally records the physical meaning, units, and characteristic sizes of variables used in the equations.

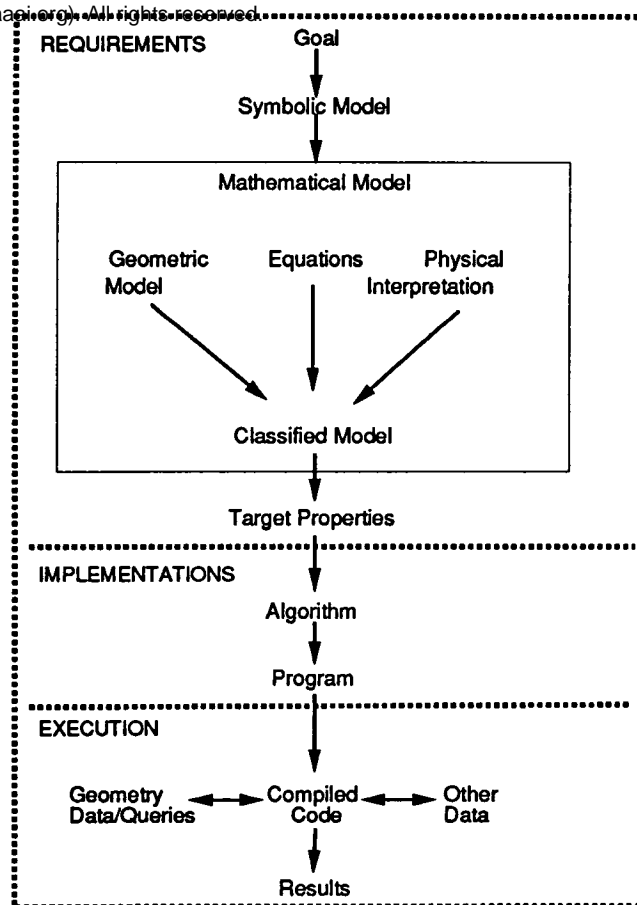


Figure 1: A problem-solving framework for mathematical modeling.

The Geometric Model describes the details of the domains over which the differential equations will be simulated. A geometric specification can include definitions of the domains of various equations and boundary conditions and the material property values suitable for simulation. The continuous geometry will be discretized into finite difference grids. In our examples, the geometry and material property values on which the model is to be run are described in files that are read by the code at execution time. Other researchers at Schlumberger have a prototype of an interactive geometry specification and meshing environment that we will eventually integrate with SINAPSE.

The Classified Model is a standardized representation intended to simplify analysis and code generation. For example, for a boundary value form, all time derivatives are moved to the left hand sides of equations and coefficients of the unknowns are categorized. The required information variable attributes such as dependencies, type, and whether the variable is to be solved for, is given, or is unknown. If the problem cannot be put into a standard form, the user should be warned that the system will probably not generate good code. Once the problem is in standard form, rough estimates should be made of the problem size required to achieve

the desired accuracy and of the computational time required (in terms of operation counts such as matrix multiplies).

Target Environment Properties describe such attributes as the target machine architecture, programming language, compiler, the details of the desired inputs and outputs, and coefficient definitions. The modeler should define any parametric variables (such as external forcing function or other boundary and initialization functions) either by a formula, by reference to an external subroutine, by reference to a precomputed table in a file, or by request to the geometric modeler (at generation time or at run time). Constants can also be determined by asking the user at run time. Useful facts or approximations about the problem, such as that a certain quantity is non-oscillatory or is always positive (perhaps used to avoid conditionals in upwind differencing), may also be included at this level.

Algorithm specification level constructs determine an abstract algorithm for solving the problem. At this level the modeler must choose an algorithm class such as finite difference; specifics of the algorithm, such as explicit method; and other parameters such as the time step of the explicit method, or the weight in an implicit method. The algorithm method is targeted to the specified architectural class (such as serial, shared-memory parallel, or massively parallel). Based on these decisions, SINAPSE selects schema representations that become the algorithm outlines. Comments about the program structure, which propagate into the target code, are inserted here. Other algorithm choices determine how the components of the schemas are filled in. These choices typically involve numerical approximations that produce discretized representations for all dependent variables represented as continuous functions. For example, the specifications can include the type of difference operator for each pattern of derivatives (the default is central differencing; there are a number of standard techniques, or the modeler can define their own) and the order of expansion of the difference operators (default 2). We are adding a partial ordering representation among schema components so that it is easy to specify all the information related to boundary handling, material averaging, and so on in one schema. The ordering information gives SINAPSE freedom to move around schema components based on true data flow constraints, efficiency considerations such as grouping read statements, and format conventions such as grouping initializations.

SINAPSE next refines the algorithm and applies control-structure optimizations, resulting in a generic Program description. We attempt to make the intermediate details of transforming from algorithm to program transparent to the user by having enough refinement rules to generate efficient programs for a reasonable variety of architectures and software environments without asking the user many questions. The refinement rules expand concepts such as "enumerate regions" or "discretize equations" that are steps of the algorithm schema representations. Eventually, the algorithms are expressed in terms of mathematical con-

cepts such as difference expressions and regions of a grid. Additional constructs specify data handling and various kinds of parallel executions. One set of important decisions concern representations such as exactly how to represent arrays (do not store the time dimension because only the most recent value is needed, compress diagonal arrays); other details may involve generating calls to external numerical analysis routines. Using symbolic transformations, SINAPSE can generate very different implementations for different architectures. For example, taper boundaries are specified symbolically as products of dependent variables and a conceptual array that is a product of conditionals in each dimension. The conditional values are based on exponentially decaying functions at the edges, and 1's in the middle. For a large-memory, massively parallel architecture such as the CM2, the "store" implementation is chosen for the conceptual array. One large array is initialized and then used in many places for fast matrix multiplication. For a small-memory, serial architecture, "recompute" is chosen, eventually resulting in a set of loops over the boundary regions doing smaller multiplications. Symbolic manipulation is used to turn conditionals into bounded loop enumerations and to simplify away the multiplications by 1.

For a final translation to code, translation from abstract program into a specific executable language such as Connection Machine Fortran or C, SINAPSE applies simple syntax-table based parsing rules and code-generation action rules.

Visualization of results is important to understanding the solution and evaluating the code, but it is not our research focus. Currently, we provide several graphical output routines but do not attempt to automatically generate display programs. We plan to integrate some standard display packages into the system and automatically generate calls to those packages.

AN EXAMPLE SYNTHESIS

Setting up a problem. Most design decisions made by modelers concern model formulation and algorithm selection. These choices can be made interactively, with SINAPSE offering increasingly specific choices, or read from a file, as shown in Figure 2.

In the interactive version, a menu of choices is offered. In some cases, such as in selecting from a standard set of model equation formulations, more formulations are known to the system than are presented in the initial menu. This is because some choices are filtered by heuristics that say, for example, that some formulations are not typically used in the application domain just specified. The modeler who insists on other choices can inspect and select them with a special menu item. In general, the modeler may select an alternative and proceed, ask for status information, or change system settings. The design decisions made so far can be displayed for reference.

To make some decisions, such as the dimensions, the user can type in arbitrary values for variable names. As another example, the number of neighboring points chosen for the approximation depends on the accuracy

```
(* model formulation *)
ApplicationDomains "is" WavePropagation;
ModelType "is" StressStrain;
Medium "is" Acoustic;
Boundaries "is" AbsorbingBoundary;
(* algorithm description *)
TargetLanguage "is" Fortran77;
AlgorithmSelection "is" FiniteDifference;
FDMMethod "is" ExplicitMethod;
BoundaryImplementation "is" TaperBoundary;
IndependentVars "is" {x,y};
DefaultOrder "is" 2;
Stagger "is" True;
```

Figure 2: Specification in design history form.

desired and on the order of the function derivative. The modeler can specify any integer. When SINAPSE requests numeric values or expressions for properties in this free-form style, help strings and defaults are offered; inputs are type checked.

In order for the user to understand what needs to be specified and how the system works, a description command is provided. For example, typing `Describe[FDMMethod]` yields a description that says it is a decision with a specific set of alternatives. It occurs first in a sequence of synthesis tasks and is followed by an algorithm analysis step.

Deriving the mathematical model. The program generation process itself is controlled in SINAPSE by task ordering constructs. For example, if we issue the command `Describe[SetCentralEquations]` we can see how the system sets up a wave equation by collecting decision results (from the specification file or querying the user) and then running a generator. If the application domain is not specified, the user is required to provide the equations and classify the variables. The original version of the system had only sequences of tasks, but the new (still incomplete) version has partial orderings on tasks. Some of the explicit representation of the tasks could perhaps be avoided by using backward chaining or means-ends analysis such as that described in [Kant 91].

`SetCentralEquations` is a synthesis sequence.

```
In class WavePropagation
  HelpText: Sets up the equations for the
             main region.
  HasSequence: {IndependentVars,Elasticity,
                Medium,GenerateEqns}
  InSynthesisSequence: EquationSetUp
  Precedes: SetBoundaryEquations
In class GenericApplication
  HasSequence: {AskEqns,IndependentVars,
                DependentVars,ParamVars}
```

In our example, the system sets up a model consisting of three first-order, coupled partial differential equations relating the time and space derivatives of the stresses S and particle velocities U . The user had already specified that the independent variables were

x, y, t . Spatially-dependent parameter variables produced by the equation generator are ρ , the density of the medium, and λ , a stiffness index.

$$\frac{\partial S_{xx}[x,y,t]}{\partial t} = \lambda[x,y] \left(\frac{\partial U_y[x,y,t]}{\partial y} + \frac{\partial U_x[x,y,t]}{\partial x} \right) \quad (1)$$

$$\frac{\partial U_x[x,y,t]}{\partial t} \rho[x,y] = \frac{\partial S_{xx}[x,y,t]}{\partial x} \quad (2)$$

$$\frac{\partial U_y[x,y,t]}{\partial t} \rho[x,y] = \frac{\partial S_{xy}[x,y,t]}{\partial y} \quad (3)$$

The system normalizes these equations, determining dependencies in the computation order and putting the unknowns onto the left-hand sides of the equations.

Outlining the algorithm. The user-specified decisions about the algorithm determine the outline or program schemas for SINAPSE to set up. Additional decisions (grid staggering, grid size, time-step size, region of tapering, quantities changed by source, quantities read at receivers) are needed to elaborate a complete program, but in most cases those can be inferred from problem class using inheritance. Because of the inference capability, SINAPSE specifications tend to be fairly concise.

The choices of algorithm methods determine the overall structure of the program, again using the inheritance structure to assemble algorithm fragments. For example, to implement the finite difference technique, SINAPSE builds a program structure with declarations, time-stepping loop, boundary conditions, and input and output statements (for example, to save entire wavefield snapshots or to save signal traces at the receivers). The initial construction is an algorithm skeleton that looks something like the following.

```
Initialization
  InitializeBoundary
TimeStepLoop
  EnumerateRegions
  UpdateQuantities
  PostStep
  UpdateBoundaries
  ReadReceivers
```

After processing decisions to use taper boundaries and expand regions in sequential rather than parallel style, the following algorithm is produced (*Mathematica* uses square brackets for function calls, curly braces for lists, and a right arrow for replacements).

```
Initialization
  InitializeTapers
TimeStepLoop
  ForEachQuantity Q in {Sxx, Ux, Uy}
    ForEachPoint Pt in Grid[{x, y}]
      Approximate[Equation[Q],Pt,order->2]
  PostStep
  UpdateTapers
  ReadReceivers
```

Refining the program. The detailed refinements of solution methods are encoded as *Mathematica* rules. For example, to implement the time-stepping loop, SINAPSE has rules that use *Mathematica*'s pattern

matching capabilities to scan the differential equations and replace continuous functions by their discretized approximations. Additional rules produce finite difference assignment statements and convert representations in functional notation into array operations with the necessary declarations. The enumeration of the grid can be quite complex if all boundaries are treated differently. However, in the simple case on a sequential machine it is a nested loop. The result of program refinement is a pseudocode program.

```
sequence[
  comment["Set up absorbing taper"],
  <code>,
  comment["Time loop"],
  all[{t, 1, Size[t]}, sequence,
    <code for other quantities>,
    comment["Update Ux discrete form"],
    all[{x, 1, -1 + Nx}, parallel,
      all[{y, 1, Ny}, parallel,
        assign[Ux[x, y],
          Ux[x,y]-(Sxx[x,y]-Sxx[1+x,y])
            / rho[x,y]]],
        <code for other quantities>]
    comment["Update Taper Boundaries"],
    <code>,
    comment["Interpolate receiver signals"],
    <code>]]
```

Producing the target code. From the pseudocode plus declaration information, SINAPSE generates target language code using syntactic rules. On our example, the entire process from problem formulation to target code, takes about one CPU minute on and yields about 350 lines of code. Larger (thousand line), more complex problems can take up to 10 minutes. Complete programs are generated, including declarations, input and output statements, subroutines, and comments about the original equations and each update statement for each variable on each dimension.

Visualization. A good way to understand the output of a mathematical modeling program is to generate graphical representations of the data. We provide several kinds of displays. One of the outputs of the program is a set of values representing seismic echoes at a set of vertically aligned receivers. The synthesized programs can also output snapshots of wave fields at a sequence of points in time. We have a manually written program for a Silicon Graphics machine that displays such outputs as animation sequences.

Making changes. The alternatives and parameter values resulting from the interaction sequence (combined with any pre-specified statements) form the design history of a program, which is useful for documentation, design revision, or alternatives comparisons. The interaction history can be saved out in a specification file for further editing.

A simple example modification is to change the dimension specification from `IndependentVars is {x,y}` to `IndependentVars is {x,y,z}`, yielding a 3D wave propagation version (manual implementation of 3D programs are quite tedious, especially when there are complex cases around the boundaries). Other vari-

ations are using implicit rather than explicit methods, changing the finite difference operator's order, changing the boundary conditions, and averaging material values around critical transition regions.

IMPLEMENTATION PLATFORM

Automated scientific problem solving uses symbolic and algebraic reasoning as well as numerical computation. Program generation requires a powerful symbolic language allowing implementation techniques such as rules, pattern matching, and object representation. A practical synthesis system should be portable to enable wide availability without excessive recoding. We have found that *Mathematica*, while not the only possible solution for these requirements, has been a satisfactory environment for both the rapid prototyping and continued development of SINAPSE. Representative symbolic manipulation systems are Macsyma, Reduce, Maple, and Scratchpad. We selected *Mathematica* for its availability on virtually all platforms used by engineers, its suitability for both symbolic manipulation and programming, and the familiarity of the notation to modelers.

Using the one system for both symbolic manipulation and program synthesis enables us to use one notation for the mathematics of problem specification and the procedural knowledge operating on those mathematics with no unnecessary representation conversions. The scientific modeler's preliminary activities, such as derivation of mathematical models, are simplified by the availability of a symbolic manipulation facility. In addition, during synthesis, SINAPSE applies some simple analysis techniques to determine, for example, whether a set of parameters satisfies a convergence criteria. Many transformation steps are conveniently represented as algebraic transforms (such as series approximations to derivatives and substitution of variables to effect a change of coordinates).

SINAPSE adds many capabilities to *Mathematica*'s basic primitives (such as differentiation). Examples are application-specific refinement knowledge, code-generation knowledge that is efficient for very large data structures or parallel architectures, and a record of design history. We have also added a substrate for representing objects, tasks, and transformations. Although methods such as finite difference approximations can be coded in *Mathematica*, interpretation of such programs for our applications is useful only for prototyping. Even on moderate sized arrays, such code is much too slow and the accuracy and stability of built-in algorithms is not always appropriate.

SHARING

The amount of knowledge required for automating code generation is very large, even for quite restricted classes of problems. Possible ways to facilitate sharing *among* code generation systems (vs. within a single system) include reuse of system components (some domain-independent), reuse of reasoning algorithms, and reuse of interface languages (e.g., high-level specification languages, array-level languages).

Reuse of system components might be possible if we could divide systems into components with well-defined interfaces. This means we first need to agree on the *meaning* or content of any specification languages or intermediate representations. We also need to formalize the *form* of the interfaces. Ironically, the methodology for figuring out how to implement a specified need in terms of existing components, or how to adapt components to a function, will probably itself exploit automated software design techniques. Some components may be large, some may be clusters of knowledge about well-defined concepts.

In SINAPSE we are attempting to identify some major phases in the design of scientific computing software and to provide different languages for some of the levels. The languages may vary to exploit mathematical formulations, array-manipulation, and conventional applicative languages so that specifications can be entered in the most convenient style. Next, we need to determine whether these stages make sense for other applications. Within these levels, there might be formalizations of abstractions such as coordinate transforms, pointers, I/O, and parallelization. Ideally, SINAPSE would then be able to interface to other systems, for example to generate a different target language, or call subroutines rather than generate code for specific tasks. Similarly, a model formulation system might generate high-level specifications from which SINAPSE can generate code.

The reasoning-technique approach is another cut at providing tools. For example, SINAPSE could use someone else's inequality prover, assumption-maintenance tool, data flow analyzer, or an expression optimizer to minimize operator costs according to a declarative cost model or to order for optimal numerical stability. It would be useful to have language-independent compiler optimization tools.

If we could find a useful set of common tools or components, major barriers (besides the not-invented-here syndrome) might be standardizing the interfaces and achieving portability of tools. Even though it is now possible to interface many different languages, in a system with multiple implementation languages, the overhead in both execution and modifiability can be quite high. Nevertheless, even if it requires reimplementing, a clearly specified set of tools and algorithms for accomplishing the goals of the tools should facilitate reuse.

DISCUSSION

SINAPSE contains approximately 20,000 lines of *Mathematica* source code. System interaction and control account for 15% of the code; model formulation and algorithm selection, 15%; algorithm refinement, 35%; code generation, 30%, and example program specifications, 5%. We estimate that slightly over half the existing system could be reused for synthesis of other types of scientific programs. As we continue to work towards the goal of practical application, we need to work on code generation for multiple architectures, tools for adapting SINAPSE to new applications, and data management. Other important areas such as visualization

and input geometries are being addressed by other research groups and are not the focus of our efforts.

Our first priorities for code generation are improving performance of Connection Machine code and increasing the number of language-independent optimization transformations not typically found in compilers. Later, other architectures will be of interest; we want to incorporate performance prediction to help guide the transformation process.

We are also working on improved code generation for the sizable portion of modeling programs devoted to data management (reading and writing data sets stored on files that contain such elements as model parameters and geometry descriptions). Subroutines to read and write data sets are complicated by requirements to parse input files, validate inputs, convert formats, and traverse data structures. If data sets are large, I/O performance may be important and may involve sequential tape processing.

The current SINAPSE system demonstrates that the approach is suitable for scientific programming, although much work remains in extending the system to more of the application domain and in generating better code for parallel architectures. Specifications are much easier to understand than the generated code, and are usually less than 20% the size of the generated code. New chunks of program generation knowledge, such as for different boundary or difference operators, can be added in the order of days and then reused in other applications. We estimate that at least half the system would be reusable in different scientific programming applications. The system has already generated more working application code than the 20,000 lines that it contains.

Acknowledgements

Many thanks to Ira Baxter, François Daube, William MacGregor, and Joe Wald who have implemented much of SINAPSE. Hung-Wen Chang and Ping Lee have offered the modeler's perspective on the system. We could not have built a knowledge-based program synthesis system without domain experts; we are grateful to Michael Oristaglio, Curt Randall, Charles Watson, and Barbara Zielinska for their time and patience.

References

- E. Kant. "Data Relationships and Software Design." Chapter 7 in *Automating Software Development*, M. Lowry and R. McCartney, (editors), AAAI Press/MIT Press, Menlo Park, CA, 1991, pp. 141-168.
- E. Kant, F. Daube, W. MacGregor, J. Wald. "Scientific Programming by Automated Synthesis." Chapter 8 in *Automating Software Development*, M. Lowry and R. McCartney, (editors), AAAI Press/MIT Press, Menlo Park, CA, 1991, pp. 169-205.
- S. Wolfram. *A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, Inc. Redwood City, California, 1988.