SciNapse: A Problem Solving Environment for Partial Differential Equations

Robert L. Akers
Elaine Kant Curtis J. Randall
Stanly Steinberg* Robert L. Young

SciComp Inc.
5806 Mesa Drive, Suite 250
Austin, TX 78731
Phone: 512-451-1050, FAX: 512-451-1622
email: info@scicomp.com
WWW: http://www.sig.net/~scicomp/

March 12, 1998

KEYWORDS: knowledge-based problem solving environment, high level specification, program synthesis, code generation, program transformation, code synthesis

Abstract

The SciNapse code generation system transforms high-level descriptions of partial differential equation problems into customized, efficient, and documented C or Fortran code. Modelers can specify mathematical problems, solution techniques, and I/O formats with a concise blend of mathematical expressions and keywords. An algorithm template language supports convenient extension of the system's built-in knowledge base.

*Also, Department of Mathematics and Statistics University of New Mexico Albuquerque NM 87131-1141 USA

1 Introduction

Partial differential equations can represent the essence of a broad range of problems in engineering, science, and other technical fields. Those who need to solve systems of these equations numerically, however, often do not have the right combination of knowledge – expertise in a technical discipline, in numerical analysis, and in computer science or programming – to do an efficient job of it. Therefore many researchers have been attracted to the vision of a *problem-solving environment*, or PSE, that could provide comprehensive help in solving systems of PDEs.[1]

We have worked on such PSEs for nearly 10 years, using applications ranging from wave propagation [2, 3] and computational fluid dynamics to financial modeling [4, 5]. Our current PSE, SciNapse© focuses on code generation; it is a system for solving scientific computing problems without actually programming by hand, and could function as part of a larger PSE system. SciNapse has generated codes that solve:

- the unsteady Maxwell's equations in 3D dispersive, anisotropic media;
- the Black-Scholes equation for valuation of multiple asset derivative securities in computational finance, including the effects of stochastic asset price volatility and interest rates, discrete sampling of spot prices, discrete dividends, early exercise, and discretely and continuously sampled stationary or moving barriers;
- non-linear, multi-dimensional, multi-species, reaction diffusion equations for chemical and nuclear applications; and
- time domain solution of visco-elastodynamic equations in 3D, anisotropic media.

SciNapse currently can generate codes that solve a wide range of initial boundary value problems for systems of PDEs, as well as many steady-state problems. The system can apply finite-difference methods to any region that can be mapped to a rectangle in any number of dimensions, though codes using very high dimensions may require excessive computational power to execute.

The codes SciNapse generates for these applications can include features such as general coordinate transformations and grid generators, various linear solvers and preconditioners, higher-order differencing techniques, automatic interpolation of equation parameters from multi-dimensional tabular input data, jump conditions in both space and time dimensions, free boundaries, and imposition of solution constraints such as positivity. SciNapse currently does not have the data structures to represent finite-element solutions on unstructured grids. It also currently lacks the knowledge to solve PDE problems with boundary element, boundary integral, or Monte Carlo techniques. We are in the process of adding algorithms for solving nonlinear PDEs and knowledge about nonlinear optimization algorithms. The goal is to generate codes in which the finite-difference PDE solution becomes the forward engine for solving multiparameter inverse problems via nonlinear optimization.

Crucial aspects of our PDE technology that make it useful in practice include the specification language, refinement of specification to code, reporting and help systems, customization of the environment, and the use of templates for representing algorithms. Problem specifications in SciNapse typically range from several lines to a half page, and the synthesized codes can be thousands of lines long. The ratio of code length to specification length ranges from 10-1 to 120-1, depending on whether macro file lengths are included and on the complexity of the problem. More complex problems usually have higher expansion ratios. On a 200-MHz personal computer, the system generates code at about 50 lines per minute.

1.1 History and plans

The SciNapse project has evolved over about eight years with an average of two or three computer scientists as implementers and one or two mathematicians and physicists as advisors, testers, and users. Three of the project members have been involved continuously. SciNapse currently comprises approximately 75,000 lines of Mathematica code. About half (the most rapidly growing section) represents knowledge of mathematics and PDE solutions, about one quarter is general computer science knowledge, and about one quarter is problem-solving system and interface support.

We originally developed the system at Schlumberger, focusing on oilfield applications such as the modeling of seismic and acoustic logging tools, with Connection Machine Fortran as a target language. Several of the generated codes, after some hand tuning, were used in company design projects. After founding SciComp and transferring the project to this new company, we gradually shifted much of the focus to financial applications such as the pricing and hedging of derivative securities and the generation of C code. The generated codes have been extensively tested, with some in use at major investment banks.

At this point the system development is well advanced in the financial modeling area, and SciComp is releasing a commercial finance product, SciFinance. We plan to produce PDE packages for students and professional engineers in the future. We are also broadening the financial application domain and are providing interfaces for less technically oriented users.

1.2 An overview of SciNapse

A critical feature of any PSE for PDEs is having an easily-understood, high-level *problem specification language*, or PSL, that supports natural descritions of the problem's geometry, mathematics, and desired interfaces. SciNapse's PSL allows problems and solution strategies to be stated concisely and abstractly, much as one might describe them to a colleague. The language allows easy specification and modification of input and output formats, and (optionally) of solution algorithms and interfaces to numerical libraries.

Because the specifications are at such a high level, the synthesis system must bridge the gap from coordinate-free mathematics to target language code. SciNapse applies an extensive knowledge base to produce code. The knowledge base includes coordinate-free constructs (e.g. laplacian), equations (e.g. Navier-Stokes), discretization rules (e.g. Crank Nichol-

son), time-stepping algorithms, solvers (e.g. Preconditioned Conjugate Gradient, SOR), optimization rules, and so on. Users can extend and customize the knowledge base. Using this knowledge base, the system converts equations into discretized, scalar components and weaves them into algorithm templates that can be specified by name. The system chooses appropriate data structures and generates a pseudocode solution that is then translated into the desired target language. Along the way, it optimizes the mathematical problem, the abstract algorithms and data structures, and the resulting code.

The user interface in SciNapse consists of the PSL, "level summaries" that track the progress of the code generation process, an information system or help system that contains a hyper-linked glossary and specification language descriptions, and a simple graphics display system.

SciNapse is constructed on top of a general-purpose knowledge-based system that includes objects, rules, and the use of computer algebra (the entire system is implemented in Mathematica© [6]. Much of the algorithmic information is stored as transformation rules or in extensible libraries in the form of templates.

1.3 Important features of code synthesis

Code *synthesis* systems such as SciNapse are much more powerful for numerically solving PDEs than are numerical prototyping environments such as Matlab© [7], the classical assembly of components from a numerical library, or even a C++ numerical library with an intelligent front end. SciNapse can, in fact, do all of these things, but it is far more flexible.

A system that can automatically synthesize code makes it possible to have specification languages of even higher level than prototyping languages. For example, simple keywords can stand for known equations, algorithms, and discretization methods. Synthesis allows the generation of efficient implementations, whereas interpreted code executes slowly and may not scale up to large data sets. Because the knowledge-based kernel of SciNapse is a general intelligent agent, the system could be extended in the areas of equation and algorithm selection, model identification, and results analysis.

The combination of highly abstract SciNapse templates with code synthesis allows users to focus on the physics and mathematics of problem solutions rather than on the mechanics of combining library components. This methodology allows the use of arbitrary higher order methods without explicit library representation, validates whether specific compositions satisfy assumptions made by individual components, and performs *global* optimization of programs. Components can be created to have the target language, data structures, and interface needed for a particular environment, yet be easily modified for reuse in other environments.

Although some of SciNapse's templates generally correspond to the modules or objects found in conventional or C++ libraries (solvers, time-steppers, evolution algorithms), the combination of highly abstract templates with code synthesis has several important differences. With most libraries, you have to write the stencil setups or discretization routines yourself, make data structure decisions, and explicitly call many low-level routines to ini-

tialize, allocate, free, and finalize codes. SciNapse's templates are not code subroutines that are called. Rather, they are very abstract skeletons of algorithms (such as solvers) that are independent of spatial dimensionality, data structures, and target language considerations. The system automatically generates optimized data structures based on the automaticaly generated stencil arrays or discretized equations and uses those data structures in custom-generated solvers. It automatically optimizes control structures according to data flow requirements. Further global optimization is used to generate code appropriate to the specified output architecture (serial, parallel, or distributed) and language (the choice of which may allow array-level operations or may affect the order of nested loops).

The ability to generate a variety of program interfaces makes synthesis especially useful for generating target codes suitable for PSEs having paradigms such plug and play or dynamic assembly. Flexible generation of interfaces also facilitates incremental replacement of components of legacy codes with automatically synthesized codes, retaining the high-level specifications for modification and reuse. Easily understandable specifications also promote cooperative use of PSEs. Although we are quite some distance from the full vision, high-level problem specification with synthesis of target codes provides many opportunities for realizing the full power of PSEs.

2 Specifying PDE problems in SciNapse

To illustrate the problem specification language, we will describe a simple diffusion equation and summarize how it is specified in SciNapse.

2.1 Specifying a diffusion problem

Diffusion equations have many applications, such as cooling of electrical and mechanical components, the diffusion of chemicals, and the diffusion of populations in biology. Despite their simplicity, solving even the most elementary diffusion equations numerically is nontrivial.

The diffusion equation specification illustrated in the PSL (Figure 1) is often given in textbooks in coordinate-free notation as

$$\frac{\partial f}{\partial t} = a\nabla^2 \ f \,, \tag{1}$$

where ∇^2 is the Laplacian, which is written `laplacian[]` in the PSL, and a is the diffusion coefficient. When described in a Cartesian coordinate system, with x and y as the spatial variables and t as the time variable, then the coordinate-free diffusion equation (1) becomes

$$\frac{\partial f}{\partial t} = a\left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\right) \,, \tag{2}$$

where the solution $f = f(x, y, t)$ is a function of the space and time variables.

The physical region in x-y-space where the PDE is to be solved in the simple rectangle $0 \leq y \leq 3$ and $0 \leq x \leq 2$. A Dirichlet boundary condition is chosen for the full boundary;

```
(* Geometry *)
   Region[0 <= x <= 2 && 0 <= y <= 3
                       && 0 <= t <= 1, Cartesian[{x,y},t]];
(* The PDE *)
   When[Interior,
       der[f,t] == a laplacian[f]; a == 2; SolveFor[f]];
   Variable[a, Scalar, DiffusionConstant, "diffusion constant"];
(* Boundary Condition (for full boundary) *)
   When[Boundary, f == 0];
(* Boundary Condition (on one edge of boundary) *)
   When[y == 3, f == 100; ];
(* Initial Condition *)
   When[InitialValue, f == 50 Sin[Pi x/2] Sin[Pi y/3]];
(* Parameters *)
   Inline[{a}];
(* Evolution Algorithm *)
   Movie[frames == 5];
(* Discretization *)
   CrankNicholson;
   RelativeErrorTolerance[.01];
   Double;
(* Solver Algorithm*)
   ConjugateGradient[SSORPre[omega == 1.0, maxit == 1]];
(* Runtime Interfaces *)
   Output[f, OneFile, Labelled, "PCG-SSOR.dat"];
```

Figure 1: Specification for a Diffusion Problem

that is, the solution is set to zero on each piece of the boundary, except where overridden by $f = 100$ on the $y = 3$ edge. Time is the infinite interval starting at $t = 0$, and the initial condition is

$$f(x, y, 0) = 50 \sin\left(\frac{\pi x}{2}\right) \sin\left(\frac{\pi y}{3}\right) . \tag{3}$$

Because the solution of this problem decays rapidly to zero, we set the final time t_f to a rather low value, say 1. The goal is to create a movie of the solution: a sequence of 3D plots of the solution $f(x, y, t)$ at equally spaced times.

Figure 1 gives a problem specification that will generate a Fortran code to solve this problem. Comments, delineated as *(* comment *)*, are optional. The specification language mirrors a typical mathematical specification to make it as natural as possible. So the statements in Figure 1 are direct translations of the mathematical statements into specifications in the PSL. We discuss such specifications in detail elsewhere[8].

For straightforward problems, this is the most that needs to be specified. SciNapse makes any remaining choices needed to translate the problem specification into a numerical code. For example, SciNapse will select appropriate discretization methods and solvers if the specification does not indicate what to do.

2.2 How specifications are organized

A specification typically has a mathematical part and an interface part.

The mathematical specification is organized as an outline, with the description of the geometric region's parts providing the structure. Equations can be associated with the interior, boundary and boundary parts, and initial time. This outline also provides a structure for expressing the discretization rules to be used on particular parts of the regions and for algorithms used to solve the problem. Equations have a similar parts structure including systems of equations, individual equations, and terms of equations. Discretizations can be applied to systems, equations, or terms.

The interface part includes specifications of the target code language, of how variables are to be initialized (read interactively, from files, inlined, computed from functions, and so on), and of output formats. SciNapse can generate data appropriate for use with visualization packages, it can read and interpolate data sets, and it has a simple interface to Mathematica's graphics utilities. The system also can generate some types of external subroutines and calls to them. We will extend it so that generated code can include calls to sophisticated graphics packages, extensive external libraries, and complex external data sets.

A specification can occur at multiple levels, as shown in Figure 2. Specifying at the highest possible level of abstraction maximizes clarity of understanding and ease of revision and optimization. Most specification is done at the top three levels in the figure. For example, the user can simply specify keywords like `ConvectionDiffusion` to obtain the convection-diffusion equation or `CrankNicholson` to get the Crank-Nicholson time discretization. The coordinate-free level allows the specification of differential equations in terms of the divergence, gradient, curl and Laplacian.

```
Here is where the box diagram from the CS&E paper goes, the content
of which is a stack of boxes labelled as follows (editors: you guys
generated this graphic)

Keyword specification of model  -|
Coordinate-free equations        |- Produces physical/mathematical model
Component-form equations        -|
------------------------------
Approximation/discretization    -|
Algorithm templates              |- Produces computational model
Data structures                 -|
------------------------------
Optimizing transformations      -|
Program/code generation         -|- Produces executable code
```

Figure 2: Levels of specification and abstraction in the SciNapse code generation system.

Many problems require only keywords and coordinate-free specifications. At the next step down, the component level, differential equations can be specified using a particular coordinate system for the variables and vector components. The user rarely specifies directly at the discrete level. SciNapse refines specifications into successively lower levels, with the final stages involving a numerical algorithm specified in the internal numerical programming language that as a last step is translated into a standard compilable programming language.

3 Program refinement

Paralleling the steps an expert might take when programming by hand, SciNapse automatically refines a specification in a stepwise fashion from the most abstract level through several more concrete levels, finally creating a numerical code. The system constructs a complete program description at a given level by combining constructs from that level in the specification with refinements of constructs from earlier levels. It expands keyword choices made by the user at the appropriate level. The mathematical and programming constructs used during refinement are represented as objects in the knowledge base; transformation rules expand the keywords and more abstract constructs into elaborated constructs. After each stage, SciNapse checks the problem state for consistency appropriate to that level of abstraction.

First, if any PDEs, boundary conditions, or initial conditions are given in coordinate-free form, the system refines them into component equations using a specific coordinate system. The resulting scalar equations must be transformed from continuous to discrete form, so SciNapse analyzes the equations and selects appropriate discretization methods

(unless overridden by recommendations from the user that are consistent with the rest of the specification).

Next comes the algorithm selection level. User design decisions, if given, are validated; if they are not specified or not valid, SciNapse makes the decisions based on the nature of the problem. Next come data structure decisions, which the system normally makes without the user's input, based on the equations, discretizations, and algorithm choices. SciNapse then makes generic programming choices at the program level, constructing a pseudocode from the templates as elaborated by the data structure choices. Control structures and operations are optimized at the pseudocode level. Finally, SciNapse generates target-language code from the optimized pseudocode.

3.1 "Level summaries" report on progress

After each stage of refinement, SciNapse generates a *level summary*, an output report that enables users to see directly how their specifications were understood and integrated with the system's inferences. The summary's structure reflects the geometric organization of the problem specification, and like the specifications, the summary uses a notation as close to that of traditional math as possible. We want SciNapse to be a mathematical partner, solving the problem with the user.

Figure 3 shows a small part of the Mathematica notebook that SciNapse produces after the coordinate-free synthesis stage in processing the specification of Figure 1. Here the function f in the equations appears with no arguments in the Laplacian expression. Users manipulate the standard Mathematica notebook interface for viewing and interacting with level summary reports. The menu buttons across the window top show the possible interactions. The nested brackets along the right break the report contents into hierarchically grouped cells that can be closed and opened to show different levels of outlining. Clicking on an active element (such as a variable or equation name) produces a display elaborating on that element. Combining this sort of hyperlinking with outlining provides flexibility without requiring that the user remember too much context.

Figure 4 shows the component-level summary, with equations expanded into the variables determined by the choice of a Cartesian coordinate system. The Laplacian is expanded into derivatives of the spatial coordinates. Cells marked with downward-pointing arrowheads, such as the line "Summary for Discrete level" near the bottom, indicate groups that have been closed; the visible line is the first cell of the group. To open or close the display of a group, a user double-clicks on its bracket.

Figure 5 shows the problem after discretization. The derivatives have now been replaced by difference expressions, and f is now a function of the space and time coordinates. The values for the solver and the evolution algorithm are now present.

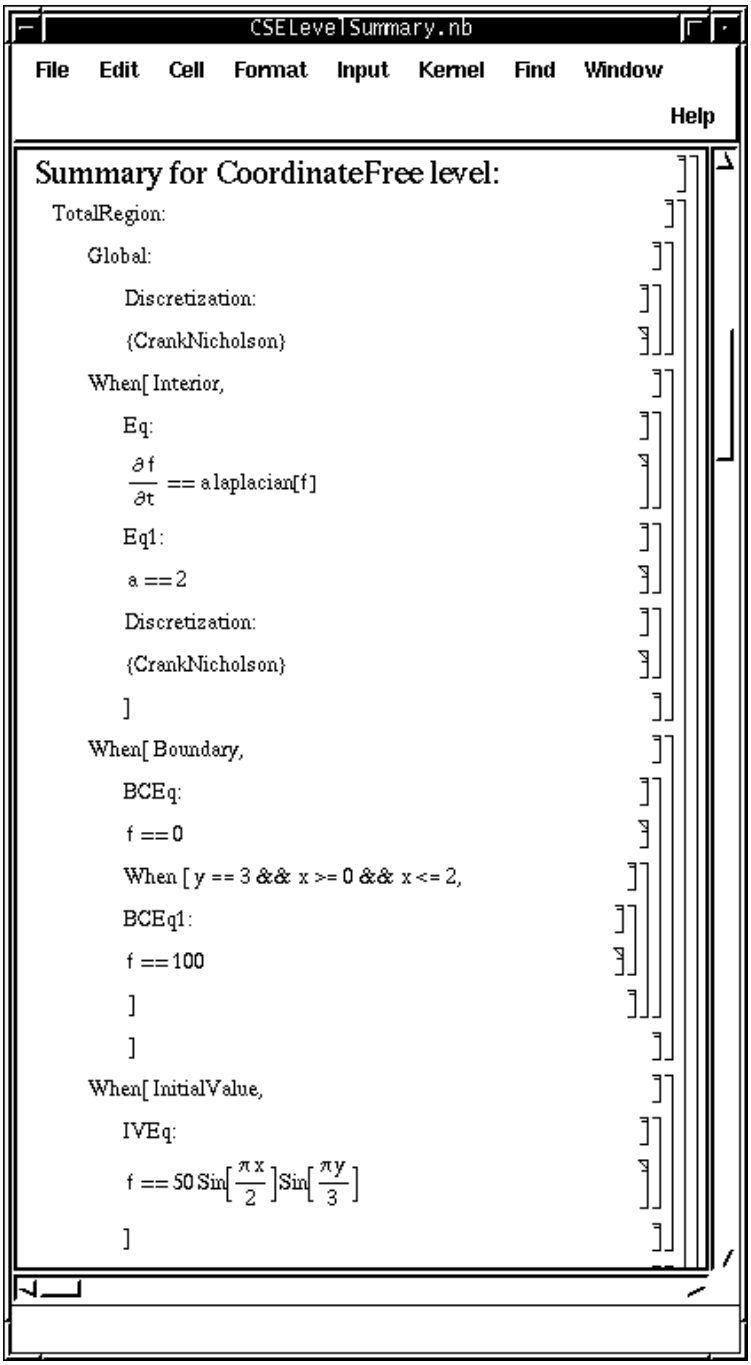

Figure 3: Summary at the coordinate-free level

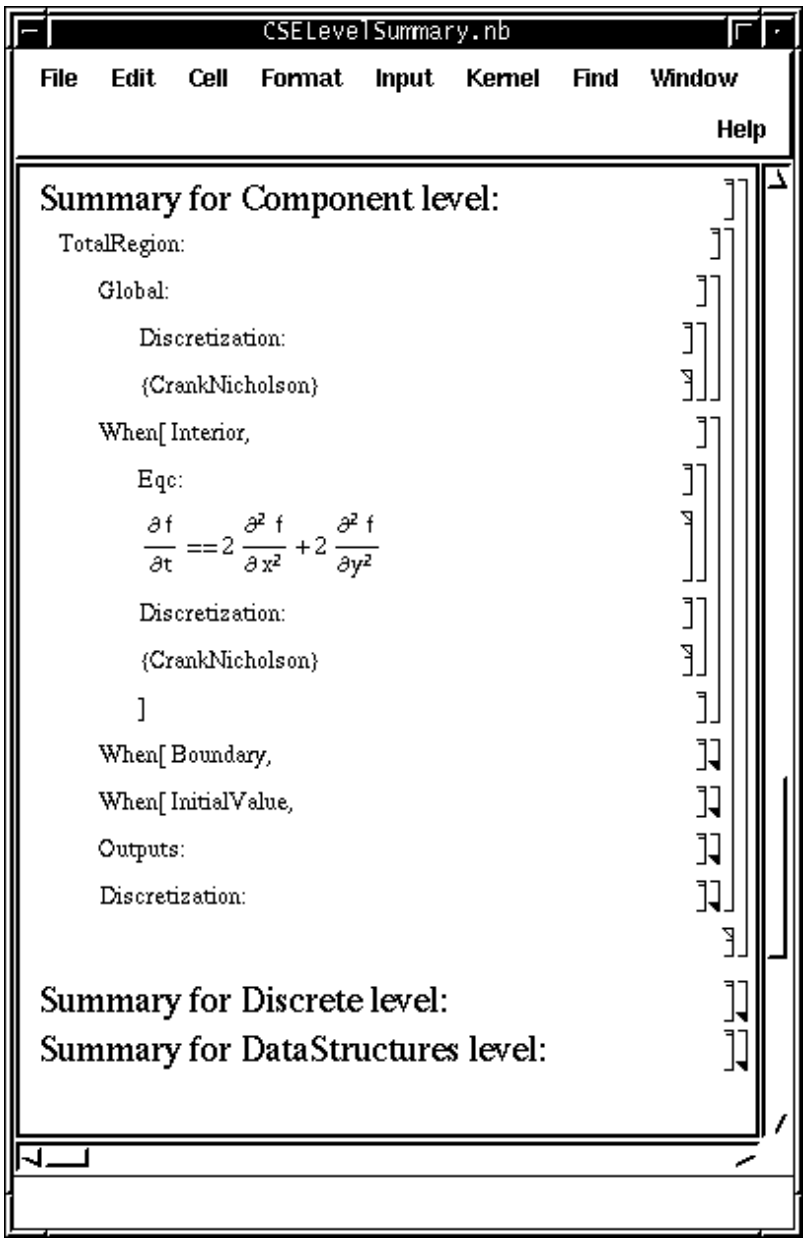

Figure 4: Summary at the component level

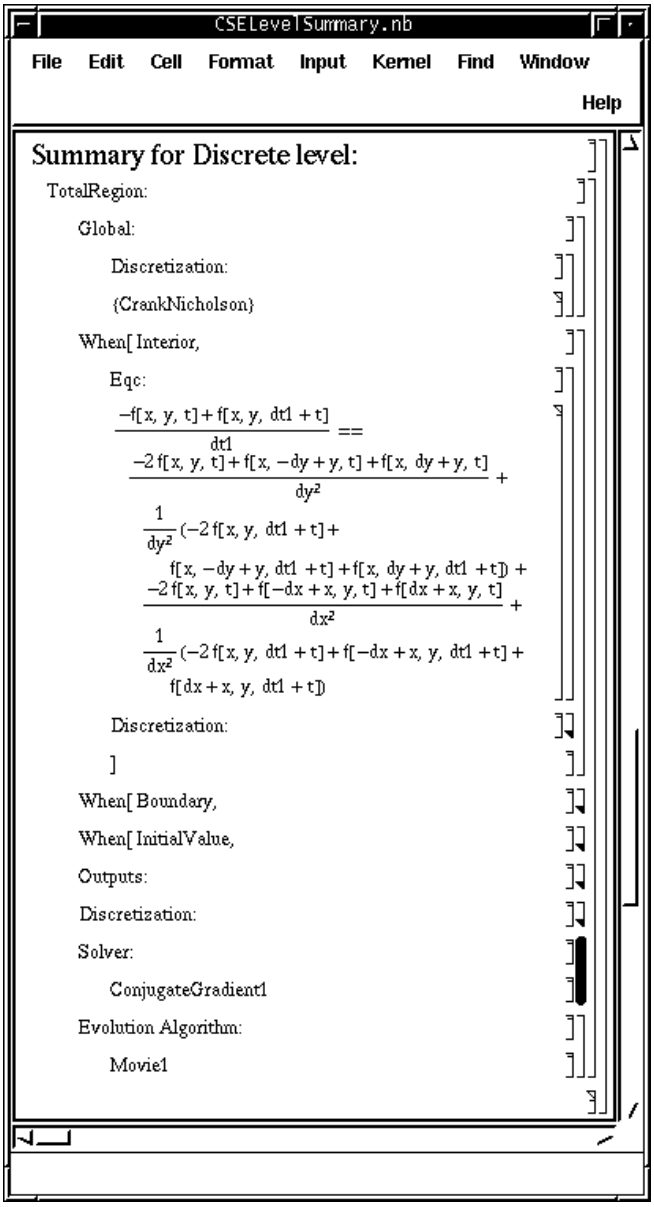

Figure 5: Summary at the discrete level.

4 The general purpose knowledge-based system

SciNapse's high-level specification language supports and encourages users in the iterative design and experimentation so critical in problem solving. Automatically translating such specifications into efficient code requires code synthesis.

Our code synthesis system is built on top of a general-purpose knowledge-based system which we have written in Mathematica. The system includes an integrated object system, rule system, and planning system. It supports a mixed decision style in which the system can make any design decisions not specified by the user. For instance, the system will make any discretization or algorithm choices left unspecified. The algorithmic information is stored as transformation rules or in an extensible collection of templates. SciNapse contains its own internal numerical programming language which is easily translated to the commonly used languages such as Fortran or C.

4.1 Representation via objects and rules

SciNapse objects explicitly represent common mathematical constructs such as a geometric region or part thereof, a system of equations, an individual equation, a term, or a problem variable. Objects also represent programming constructs such as a linear solver, a subroutine, or a program variable. Objects carry attributes identifying methods for elaborating the object and its properties, attributes describing associated design decisions, and explanatory help information.

Refinement rules detail the methods to which the object attributes refer. Rules may expand keywords and abstract constructs, make discretizations, or check specifications for internal consistency of physical units and tensor order.

SciNapse creates instances of objects during the refinement process that persist throughout the entire synthesis. Some attributes on object instances record relationships between objects, such as linking an equation to all the variables it uses and linking a variable to all its equations. Users may inspect object instances to examine intermediate results such as an equation's discretization.

4.2 Explicitly represented design choices

SciNapse's knowledge base explicitly represents design choices to be made by the user or the system. Considering only the world of initial boundary value problems, hundreds of design choices exist, leading to thousands of possible programs. To maximize flexibility, we want users to be able to specify any design choices they desire, but *not* be required to specify those for which they have no preferences or those that should be obvious to the system. Some choices are simple selection among algorithms, but the algorithms are represented as highly abstract templates that will be customized to the problem. Other choices involve the method of input (such as reading from a file, entering interactively, or computing from a subroutine), file formats, and data structures. (For instance, an array can be full, diagonal,

time-independent, stencil, and so on.) These choices help interface the generated code to the rest of the PSE. Typically the system will optimize programs without advice from the specification.

To make design choices easier for users to find and understand and easier for developers to extend, design choices are represented by attributes on objects. For example, the DiscretizationMethod attribute on an equation, system, or term can have values such as CrankNicholson or CentralDifference[4] (the "4" indicates fourth order). If a design choice is identified by a unique keyword, simply mentioning that keyword in the appropriate "When[region, ...]" context is sufficient (see Figure 1).

Associated with each choice can be constraint rules, heuristic rules, and defaults. Constraints are definite requirements on the values allowed. They can be based on previous choices or the nature of the mathematical problem. For example, some preconditioner choices can be ruled out because of solver choices and matrix properties. Constraints cannot be ignored: violating them will lead to incorrect codes. Heuristics are system suggestions and can be overridden by the specification. Defaults specify what to do in the absence of decisions made by the heuristics or the specification file.

4.3 The planning system

SciNapse's planning system sets goals to instantiate objects and them refine them by filling in attributes. This includes an agenda mechanism that schedules refinements and design decisions for making selections such as algorithms and data structures. The planing system uses the method descriptions to automatically determine an ordering of refinements that ensures that all data to make choices is in place before a decision is considered.

Given more knowledge, the planning mechanism could be used in several new ways. For example, it could derive and/or select equations or variants thereof based on problem properties, set up default parameters, or analyze data resulting from program runs.

4.4 Context-dependent code optimization

SciNapse's internal representation of numerical programs is independent of target language and records the maximum parallelism possible in algorithms and equation. In this abstract representation, context-dependent global optimizations are easy and productive. These include early elimination of problem variables not needed to compute the desired outputs, and selection of optimal data structures, such as arrays based on problem-specific stencil sizes. The system also applies standard optimizations such as introducing temporary variables, loop merging and loop unrolling, and algebraic simplification.

Although the internal representation is language independent, the last step of optimization before code generation does provide for target-language specific optimizations. Currently SciNapse generates codes in C and Fortran 77, so for example, loop enumerations are reversed for C and Fortran arrays, and dynamic allocation is available in C.

Because the parallelism inherent in a problem solution is retained until the very last step of code synthesis, adding architecture-specific code generators is reasonably straightforward. (At one point we added a generator for Connection Machine Fortran, which is similar to Fortran 90.) Here, no loop enumerations will be generated for array operations. We anticipate generating codes for distributed architectures using languages like MPI in the future.

4.5 Extensive use of computer algebra

The availability of a computer algebra makes it easy for SciNapse to estimate truncation errors, compute derived quantities (such as sensitivities), transform equations into general coordinates, and discretize terms with arbitrarily high-order differencing formulas. Computer algebra capabilities allow the system to represent equations and templates independently from the problem details, yet still generate efficient code and optimize it – for instance by determining stencils from discretized equations and by using the knowledge about stencil shape to optimize array representations.

4.6 Testing

Correctness of the codes generated by SciNapse is ensured in several ways, both by overall system testing and by testing of individual codes. We have collected a large number of test problems for which we compare the results of SciNapse-generated codes against analytic solutions (when possible) and also against some alternative computations. We have a large regression test suite that we re-run whenever the system is modified. In testing individual problems, the equations of the specification are checked for consistency of tensor order and physical units (if specified). Also, the truncation error for the approximations is automatically computed and inserted as a comment in the code. A numerical convergence rate test wrapper is available to verify that the synthesized code performs in accordance with the symbolically calculated truncation error.

4.7 Customization

A knowledge-based PSE should be easy to customize for different application areas. SciNapse's PSL allows any chunk of specification language to be put in a separate file and be called by name. A typical use for this "macro" construct is to specify frequently used systems of equations with "equation generators." Conditional specification keywords can provide more customization. For example, a keyword parameter to the Navier-Stokes equation generator can control whether to include an internal or a total energy conservation equation. Macros also provide a means for succinctly specifying a collection of design choices appropriate for a particular application (sub)domain.

Macro invocations can include keywords and also rules that replace any symbol with another symbol, constant value, or expression. Such rules can change variable names, set

variable values, substitute expressions, or even pass in a set of problem-specific state equations.

SciNapse users and developers can customize algorithms, discretization methods, and decision rules. The template language, discussed in the next section, enables the definition of algorithms tailored to a specific application area. Language constructs exist to name collections of discretization methods and to define new ones. It is also possible to add new constraints and heuristics on design choices.

Using the preceding constructs, developers can define all the common names of equations, boundary conditions, derivatives and sensitivity terms, outputs, solvers, or any other design choice defined by the specification language. Application users can then write specifications completely in terms of those keywords, yielding very concise specifications that are easy for practitioners in that application area to understand and modify.

5 Algorithm Templates

Some of the expert knowledge brought to bear in code synthesis is represented as abstract algorithm templates [9]. These templates capture the essentials of an algorithm; they are elaborated with problem-specific aspects and then translated into the chosen target language. Templates support the scalability of a PSE because they make it easy for end users as well as developers to add new algorithms. Many linear solvers and other algorithms are recorded in the literature in a form very similar to SciNapse's templates ([10], [11]).

5.1 The template declaration language

Templates are defined in an algorithm description language that is independent of spatial dimensionality, data structures, and target language considerations. The rich language of pseudocode expressions includes various matrix operators (e.g., dot product, the arithmetic operators, transpose, norm, diagonal, and identity) and transcendental operators, as well as the usual scalar operators for numbers and Booleans. It also includes easily recognizable procedural control structures. In addition to conventional loop, while, and until constructs, special constructs allow looping over the range of abstractly specified subsets or supersets of array dimensions, and mapping of operation sequences over variable lists. A special dimension can be identified and loops defined over just the special or the non-special dimensions. Some parallel constructs record opportunities for distributed computation.

Data declarations in templates are quite different than those of Fortran or C. SciNapse's type annotations, rather than providing fixed storage declarations, allow declarations in terms of configurations of other variables, including those from problem's equations. In this way they resemble the *automatic data objects* of Fortran 90. Declarations can state that a new array has the same shape as another array, or the same shape but with some dimensions deleted or added, or with dimensionality and range based on the ranges and shapes of other problem or template variables.

```
Template[ConjugateGradient, SolverTemplate,
   SubroutineName[CGSolver],
   LocalVars[p[SameAs[y], "search direction vector"],
         q[SameAs[y], "image of search direction vector"],
         zz[SameAs[y], "conditioned residual"],
         alpha[Real, "multiple of search direction vector"],
         beta[Real, "orthogonalization constant"],
         rho[Real, "temporary variable"],
         old[Real, "old value of rho"],
         iter[Integer, "iteration counter"]],
   Templates[Preconditioner],
   Code[r = y - svar.xx;
        old = r.r;
        p = 0;
        Until[iter, maxit, StopCode,
          zz = r;
          commentAbout["Apply Preconditioner",
              Preconditioner[y -> r, xx -> zz]];
          rho = r.zz;
          beta = rho / old;
          p = zz + beta p;
          q = svar.p;
          alpha = p.q;
          alpha = rho / alpha;
          xx = xx + alpha p;
          r = r - alpha q;
          old = rho]]];
```

Figure 6: Template declaration for a conjugate gradient solver.

```
EvolutionTemplate
  SteadyState
  TimeDependent
    TimeDependentEvolve
      Movie
      Evolve
      Motion
      DiscreteEventsGeneral
    TimeDependentFixed
      TestingTemplate
        ConvergenceRateTest
      EvolveFixed
      MotionFixed
```

Figure 7: Partial class hierarchy for evolution templates.

Templates can be composed just like subroutines in a programming language. The need for a subordinate or "called" template is indicated by including its name as an identifier in the local "Templates" declaration. In the pseudocode, the "call" is represented as a reference to that identifier, signifying only that some class of routine, such as a solver or a preconditioner, is needed at that point. The specific choice of solver or preconditioner is not fixed in the calling template declaration, but rather is a design choice. During synthesis, the template identifier takes on the "value" of an instance of the subordinate template class selected by the specification or system's decision mechanism. For example, in Figure 6, we declare the identifier Preconditioner. Its appearance in the code indicates where the preconditioner call will occur. Arguments to the called template may be automatically inferred by rules.

SciNapse's knowledge base represents each template as a class in a hierarchy of template classes. This hierarchy allows for a separation of concern in template declarations as well as defining families of options for various kinds of algorithms. Among the major classes of templates are solvers, time evolution templates, stepping algorithms, and inverse problem or test wrappers (e.g., for automatically generating convergence rate testing code). Using normal object inheritance, aspects like variables or template placeholders defined high in the hierarchy are known in all the subclasses and need not be re-declared. For example, in Figure 6, the StopCode declaration is inherited by the ConjugateGradient solver from its super-class, SolverTemplate.

A partial class hierarchy for time evolution templates appears in Figure 7.

Names for both variables and template place-holders can also be drawn from other templates. Mechanisms are available to refer to the variables that exist in the template's "caller" or in the SciNapse problem state, and to pass information from templates to their subordinate templates.

A template declaration can also provide an indexing scheme, which is a function that transforms the generic indexing operations in the code declaration into more concrete data representations and operations appropriate for the synthesized data structures.

A template declaration can include programming directives such as whether the system should expand the template in line or encapsulate it as a named subroutine.

5.2 Template translation

SciNapse contains a rich collection of templates written by its developers and submitted to the system via a translator that builds them into the system's knowledge base of objects and rules. The same translator will be available to SciNapse users, so that if the templates they need are not already in the system, they may declare them and have them incorporated as first class objects in the synthesis environment. To ensure all the pieces fit together in a sensible manner, the language supports good syntax-checking and type-checking, and conducts a variety of semantic checks both at template declaration time and at synthesis time.

When a template is declared, the template translator automatically creates an attribute in the template class corresponding to each variable that the template can access and each subordinate template placeholder. It also creates methods for filling in the values of these attributes.

5.3 Templates during synthesis

During synthesis, SciNapse creates an instance of the selected template class and fills in its attributes as the problem demands. Next, the system fills in the names of the specific variables that can be accessed by extracting them from the problem state or from calling templates, or, in the case of local variables, by defining variables of the appropriate kind and shape. For the attributes corresponding to template placeholders, the methods instantiate the new templates, and the process continues recursively.

In the next step of template processing, the system builds pseudocode for the various template instances. The work is done by methods derived by the template translator from the code part of the template declaration. The first step is to fill in the pseudocode from the declaration with the variables and code for subordinate templates previously derived. Thus, where a subordinate template is called, the template code (see below) for that template is inserted, either in-line or as a subroutine call. The pseudocode for the conjugate gradient solver in the diffusion example appears in Figure 8.

Next, the indexing function, if present, is applied to the pseudocode. This function transforms the abstract indexing given in the pseudocode to a concrete form appropriate for the shape of the data structures for the particular variables in the code. No such transformation is specified for the conjugate gradient solver in our example.

Finally, if the template is to be cast into a subroutine, SciNapse encapsulates its code as a subroutine object with appropriate variable declarations and replaces the pseudocode

```
seq[assign[r1, g1 - SA1 . xx1],
   assign[old1, r1 . r1], assign[p1, 0],
   seq[assign[iter1, 1],
      seq[if[tol1 < 8*eps1,
         print["Warning: Linear tolerance is too small.",
            tol1, " < 8*", eps1, " ."], seq[]],
         comment["Compute the static stopping criteria information"],
         commentAbout["The norm of the right hand side",
           assign[normg1, norm[g1]]],
 . . .]]]
```

Figure 8: Pseudocode fragment from preconditioned conjugate gradient.

```
call[CGSolver, eps1, f, g1, iMax, jMax, SA1]
```

Figure 9: Template code for calling conjugate gradient.

with a call to the subroutine. This call will be incorporated into the code for the calling templates. If the code is to be in line, no action is needed. The resulting code for calling the conjugate gradient solver in the diffusion example appears in Figure 9.

6 Discussion

Because SciNapse already can handle large and complex problems, we believe that the technologies that we have in place form the basis of a powerful PSE to assist with the solving of initial-boundary value problems for partial differential equations. We plan to extend the system to include knowledge about more complex algorithms (such as fractional stepping methods) and geometries (such as unstructured grids). Toward this end, we continue to work on improvements to the system's algorithm and geometric representations as well as adding objects and rules to the knowlege base.

SciNapse has now been tested by many users: some PDE experts, some not. The financial package SciFinance [4, 5] is being used by major financial institutions. While testing has uncovered a number of problems, none have to do with the fundamental design of the system. In fact, all reports indicate that the system is easy to understand and use, and impressive in what it can do.

Acknowledgments

This work was supported in part by the National Institute of Standards and Technology under Advanced Technology Program Cooperative Agreement #70NANB5H1017.

References

[1] A summary of problem solving environments, especially PSEs for PDEs, http://www.cs.purdue.edu/research/cse/pses/ (Current May 10, 1997).

[2] E. Kant, *Synthesis of Mathematical Modeling Software*, *IEEE Software*, **10** no. 3, (1993), 30-41.

[3] M. R. Lowry and R. D. McCartney, eds. *Automating Software Design*, AAAI Press / The MIT Press, Menlo Park CA, 1991.

[4] C. Randall, E. Kant, and S. Kostek, *Automatic Synthesis of Financial Modeling Codes*, *Proceedings of the First Annual Computational Finance Conference*, International Association of Financial Engineers, Stanford CA, August 23 1996.

[5] C. Randall, E. Kant, and A. Chhabra, *Using program synthesis to price derivatives*, *Journal of Computational Finance*, Vol. 1, No. 2, Winter 1997/1998, pp. 97-129.

[6] S. Wolfram, *Mathematica, A System for Doing Mathematics by Computer*, Addison-Wesley, 1991. Mathematica is sold by Wolfram Research, ttp://www.wri.com/ (Current May 10, 1997).

[7] MatLab, a matrix manipulation package sold by The MathWorks, http://www.mathworks.com/ (Current May 10, 1997).

[8] R.L. Akers, P. Baffes, E. Kant, C. Randall, S. Steinberg, and R.L. Young, *Automatic Synthesis of Numerical Codes for Solving Partial Differential Equations*, To appear in the special issue *Non-Standard Applications of Computer Algebra* of Mathematics and Computers in Simulation, edited by H. Hong, E. Roanes Lozano, and S. Steinberg.

[9] E. Kant and S. Steinberg, *Automatic Program Synthesis from Abstract PDE Specifications*, To appear in the proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics Berlin, August 24-29, 1997.

[10] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *Templates for the Solution of Linear System: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[11] C.T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, Frontiers in Applied Mathematics, SIAM, Philadelphia, 1995.

Robert L. Akers is a Computer Scientist at SciComp, Inc. He earned his M.A. and Ph.D. in Computer Sciences at the University of Texas at Austin. His interests include programming languages, program synthesis, and the formal analysis and algebraic manipulation of programs. He is a member of the ACM.

Robert L. (Larry) Akers
SciComp Inc.
5806 Mesa, Suite 250
Austin, Texas 78731
Phone: 512-451-1456
Fax: 512-451-1622
Email: akers@scicomp.com

Elaine Kant is the founder and President of SciComp Inc. She received a B.S. in Mathematics from M.I.T. and a Ph.D. in Computer Science from Stanford University. Before starting SciComp, she worked for Schlumberger and taught Computer Science at Carnegie-Mellon University. Her major research interests are in automating certain aspects of scientific problem solving, and in the design, implementation, and analysis of algorithms and programs. She is a Fellow of the American Association for Artifical Intelligence, a fellow of the American Association for the Advancement of Science, and a member of the ACM and the IEEE Computer Society.

Elaine Kant
SciComp Inc.
5806 Mesa, Suite 250
Austin, Texas 78731
Phone: 512-451-1430
Fax: 512-451-1622
Email: kant@scicomp.com
WWW: http://www.scicomp.com.

Curt Randall is Vice President of SciComp. He received the B.S. and M.S. degrees in Nuclear Engineering from the University of Wisconsin, Madison and a Ph.D. in Applied Science from the University of California, Davis-Livermore. Before joining SciComp, he worked at Lawrence Livermore Laboratory and Schlumberger. His research interests include computational physics and computational finance.

He is a member of IAFE, SEG, and ASA.

Curt Randall
SciComp Inc.
5806 Mesa, Suite 250
Austin, Texas 78731
Phone: 512-451-1603
Email: randall@scicomp.com

Stanly Steinberg is Professor of Mathematics and Statistics at the University of New Mexico and a consultant for SciComp Inc. He received his Ph.D. in Mathematics from Stanford University. His research interests include problem solving environments, variational numerical grid generation, the study of stability problems through quantifier elimination algorithms, porous media flow problems, and discretizations for PDES. He is a member of SIAM, IMACS, and ACM.

Stanly Steinberg
Dept. of Mathematics and Statistics
University of New Mexico
Albuquerque, NM 97131
Phone: 505-277-5323
Email: stanly@math.unm.edu

Robert L. Young is the Director of Technology at SciComp Inc. He received his PhD in Computer Sciences, from the University of Texas at Austin. His research interests center on knowledge based, interactive systems. He is a member of ACM and AAAI.

Robert L. Young
SciComp Inc.
5806 Mesa, Suite 250
Austin, Texas 78731
Phone: 512-451-1434
Email: ryoung@scicomp.com