Automating the Design of Scientific Computing Software

Elaine Kant
Schlumberger Laboratory for Computer Science
P.O. Box 200015
Austin, Texas 78720–0015 USA
kant@slcs.slb.com

Abstract

SINAPSE is a domain-specific software design system that generates code from specifications of equations and algorithm methods. This paper describes the system's design techniques (planning in a space of knowledge-based refinement and optimization rules), user interaction style (user has option to control decision making), and representation of knowledge (rules and objects). It also summarizes how the system knowledge has evolved over time and suggests some issues in building software design systems to facilitate reuse.

Introduction

SINAPSE is a domain-specific software design system that generates code from specifications of equations and algorithm methods. Our goal is for SINAPSE to be a practical program-synthesis system that solves a restricted class of problems. In particular, we aim to reduce mathematical modelers' programming chores by enabling modelers to specify programs at the level at which they are described in technical papers.

A trend toward three-dimensional modeling (previously too expensive to attempt for many applications) is both making programs more complex and requiring implementation on parallel architectures (for acceptable performance). Both consequences of this trend argue strongly for automatic code generation – to avoid errors in programs and to save modelers from having to learn about rapidly changing architectures. Because efficiency of code and interfacing with other codes are factors for many of our users, the code generation system must be understandable and modifiable.

The current SINAPSE implementation focuses on one class of algorithms – finite difference methods for solving partial differential equations. We have used the system to generate about a dozen families of programs for solving acoustic wave propagation problems of interest to Schlumberger modelers. With these programs (for which no comparable hand-coded versions existed), the modelers have achieved new results in the application areas. However, all the programs were specified by knowledgeable users, and we manually optimized critical code sections after experimenting with the automatically generated program. Current research involves generating more efficient code and making the system more easily accessible to modelers.

Although we primarily apply the system to finite difference problems, we have also generated several rather different types of codes and have used subsets of the system in other applications. Approximately half of the system (consisting of the synthesis framework and an array-level language to target code translation) is independent of the domain, although focused on scientific computing. We have used this part of the system to generate some geometric modeling codes, starting from an array-level specification language.

The lessons from SINAPSE are similar to those of other knowledge-intensive systems: it is important to design representations that are close to the users' mental models; abstraction of concepts is important; and rules and objects provide useful representation techniques. An emerging concern is how to encourage more sharing among software design systems. The last section of this paper suggests that reuse of components and reasoning algorithms may be possible among different software design systems themselves.

Specializing Design Techniques

The driving force in the implementation of SINAPSE has been the collection of design techniques appropriate for our applications. The classes of design techniques as well as the problem itself then determine the types of user interaction that are required. Finally, the knowledge representation is strongly suggested by the reasoning techniques and user interface requirements.

Given our fairly narrow application domain and goal of practical program synthesis, the most appropriate design technique is knowledge-based refinement, including the application of optimizing transformations. Refinement choices are made by heuristics or modeler specification. Although our approach includes knowledge-based optimization, as the performance demands on synthesized code have scaled up, we have seen more need for traditional types of optimization such as code motion supported by data-flow analysis.

We have explicitly chosen not to use some types of reasoning techniques. For example, learning about choices in context and learning about run-time code performance might eventually be appropriate, but we chose not to address learning, discovery, or complex

search issues in the current system. We also do not attempt inference by theorem proving; this would require very detailed domain models before any progress could be made, and these formalisms would make it difficult to allow the kinds of not-strictly-correctness-preserving approximations that modelers frequently make. However, we are attempting to develop a clean characterization of the semantics of the synthesis constructs. This is a good guideline for domain analysis and helps make the meaning of the constructs independent of the implementation. A clean semantics makes a construct easier to explain to users and easier for developers to modify.

The states in the problem space in which SINAPSE operates include descriptions of (partially implemented) programs and facts about the specifications and implementations. The space is navigated by carrying out sequences of synthesis tasks. Originally we tried to streamline the problem-solving mechanism by letting the actions in program synthesis carry out the navigation, with design choices being presented to the user as needed, but this proved confusing to the users and difficult to modify. Therefore we are moving towards an explicit plan representation. We expect to conclude by declaratively encoding the set of goals about program function and performance, plans for achieving those goals, and control knowledge about which plans to use for different circumstances. The plans consist of partially-ordered (sub)goals, bottoming out at actions that include asking the user for information, applying program refinement rules, and applying program optimizations.

A specification in SINAPSE is a collection of design decisions, most of which can be thought of as control information about which program refinements to make, or which facts to declare. In addition, sometimes a specification actually defines a new refinement and then asserts that the new alternative is the refinement that should be made.

SINAPSE is implemented in *Mathematica*[Wol88]. *Mathematica* is both an algebraic manipulation system, useful for scientific programming, and a programming language with modern features such as a pattern language and rules. Other implementation languages would also be reasonable choices, but *Mathematica* allows us to have everything in one language in which our target users are comfortable.

Phases of Design

In order to make the system comprehensible to developers and end users, and to encourage collaboration with others, we have divided the software design process into a series of phases:

- **problem set up**
- **algorithm synthesis**
- **program optimization**
- **target code generation**

How common these stages are in other design systems for scientific computation is an open question, but evidence for them can be found in [PC91; AEH+89; Coo90]. A more detailed, though somewhat dated, description of these phases is given in [KDMW91].

The first phase, **problem set up**, involves helping the user define the problem. The result should be a set of equations such as would be described in a modeling article. In our application, problem set up is accomplished by working through a network of choices (goals and tasks) that set up the equations to be solved. For applications about which SINAPSE is knowledgeable, it presents parameterized equation generators; otherwise the user must define the equations mathematically.[1] Mathematical formalization, when the equations are not given directly by the user, involves a relatively straightforward knowledge-based expansion. Next, SINAPSE may reformulate the equations via simplification, normalization, and redundant equation elimination. Other reformulations, such as averaging of material values, depend on user specification. *Mathematica*'s algebraic manipulation is especially useful at this stage.

The problem set up phase should probably be viewed as three distinct phases. Two, which are independent, are describing the **physical model** in general terms, reusable for a number of specific problems, and describing the **target properties** of the computing environment in which a specific problem must be solved. Properties of the target environment might include machine architecture (such as type of parallelism available) and limitations on run time and storage space. A specific **problem description** would then be the next phase, that would customize a physical model to a specific set of knowns and unknowns (and any desired interpretation or analysis of the computed results) and might modify the equations to be used based on the specification of target environment properties.

SINAPSE's **algorithm synthesis** begins with selecting an algorithm schema corresponding to the modeler's design decision(s) and then filling in the details. This level includes all the domain-specific computing knowledge that an applications expert would have, typically the numerical approximation methods to be applied to the equations. The types of implementation decisions are those that would be reported in a detailed technical article. At the end of this phase, programs will be expressed in *Psiam*, an array-level language that we are developing. The search for effective combinations of design decisions is currently left to the user if the default choices are not acceptable. Program details are filled in by refinement rules. Elaboration of the design decisions often involves the use of algebraic manipulation for computing approximations. If desired, the modeler can specify fragments of code directly in *Psiam*. The schema instantiation may involve elaborating parts of code such as initializations or outputs that eventually need to migrate to other sections

[1] In other applications, such as mechanics and circuits problems, systems often have more detailed descriptions of the physics of the systems and tools to instantiate the physical laws in a specific problem. The instantiation often involves much unguided object slot filling rather than the guided, dependent, goal satisfaction used in SINAPSE.

of code. The migration is done too explicitly now; we will evolve to a more general mechanism with partial orderings and data flow analysis.

Performance choices are made at the next stage, **program optimization**. This level includes all the types of knowledge that any good scientific programmer should know regardless of the application domain. Some examples of design decisions made at this stage are store vs. recompute decisions, data structure selection (array representation, primarily space compression techniques), and the corresponding operator implementations. Data and control parallelism from the domain have been explicitly represented and maintained through the program transformations until, at this level, parallelism is either exploited or, for target languages not supporting it, expanded into looping constructs or sequentialized. A number of optimizing transformations are applied. To support the data structure selection and optimizations, there is some inferencing to determine data types of dimensions, properties of arrays, and simplifications of conditionals (for example, to transform conditionals on array indices into loops with specific bounds). Currently SINAPSE uses special case reasoning for such inferences; it would benefit from an interface with an inequality prover and probably other provers or decision procedures.

The result of expansions of the previous step is expressed in *MathCode*, another language that we have developed. *MathCode* is a procedural language that abstracts away from Fortran and C constructs but has almost no remaining implementation freedoms. The final phase of **target code generation** from *MathCode* is accomplished by a recursive-descent parser with action rules for each different target language.

Interacting with Users

Our initial concern in user interaction was simply to ensure that modelers could specify their problems and override SINAPSE's default design decisions. A SINAPSE specification, which contains a set of design decisions, might "ideally" contain just decisions at the level of specifying the problem. In reality, of course, the system does not have enough information to make all the algorithm and implementation choices. Even when the system thinks it has enough knowledge, not all modelers will agree with the choices. The evolution of these aspects of the interface will be discussed here. Some other issues concerning the modeler's interface to scientific codes are outside the scope of this discussion. For example, while our total environment will involve an interface for specifying the geometry of the world being modeled and an interface for visualization of the results, these are largely separate research efforts.

The philosophy of partitioning the problem-solving load between the user and SINAPSE was discussed in [Kan90]. The conclusions, to which we still subscribe, can be summarized by:

- SINAPSE should structure the problem-solving sessions because people are smarter than software design systems and can adapt; however, SINAPSE should present the user with significant decision points and alternative implementation choices that match problem-solving models.
- SINAPSE should cooperate by making suggestions (heuristics about appropriate choices, help in finding similar specifications or concepts); however, people should have ultimate veto power over system choices.
- SINAPSE should be able to explain, at least minimally, specification choices and decisions that have been made.
- SINAPSE should have a system for helping users and developers add new knowledge.
- SINAPSE should share knowledge bases so progress for any purpose (synthesis, explanation, knowledge acquisition, or system integration) is tested by and contributes to progress for all purposes.

Current Interface

Currently, the user must be reasonably knowledgeable to set up a SINAPSE specification. Specifications are usually made in a text file that is loaded at the beginning of a session, but most choices can also be specified interactively with simple menus (for enumerable choices) or fill-in-the-blank interfaces. Also, although program fragments can be specified at the array level (effectively defining new refinement rules at specification time), there is no interactive support for this. In the interactive mode, the user can request text string explanations of the decision issues, alternatives, and system heuristics. Answers provided by the user are checked against legitimate patterns. In addition, the user can confirm or modify interactively the choices suggested by system heuristics or a previously loaded text-file specification. SINAPSE can write out a text file of the decisions made interactively, or made by a mix of previously specified text and interactions.

We have begun to make SINAPSE more accessible to modelers. We are adding pointers to examples of specific choices and their realization in target programs based on our demo suite. A graphical interface with modern menus, multiple status and help windows, and hypertext navigating is being implemented, and a minimalist-style user manual is being written. Because of the large number of design decisions and the different classes of anticipated users (some modelers care more about approximation method choices, some about efficiency of implementation), we also will need a mechanism to control which design decisions are visible to the user. One possibility is to make visibility dependent on the phase in which the decisions are made and on whether the decisions are based on hard constraints (forced choices) or heuristics or simple defaults.

Declarative Decision Structures

A good interface is critically dependent on the correctness and understandability of the underlying domain models. Indeed, users cannot even write text-file specifications if they do not have a good understanding of what needs to be specified. Although we have had some difficulty in explaining how the system works to different domain experts, the specification language

seems to be converging as we gain better understanding of the domain. Earlier versions of SINAPSE did not have all decisions explicitly represented, but we are adding a definition mechanism that ensures that all design decisions are properly inserted in a global task network. Correctly representing the domain means not only having the right set of design decisions, but ordering the decisions sensibly and representing dependencies between decisions. Although SINAPSE was able to generate the same set of programs with a more unstructured representation, having a good, declarative representation of the decision structure turns out to be critical for acquiring a specification, for storing out a specification in text format for later use, and for explaining specifications and system decisions to users.

Dependencies between Decisions

An explicit representation of all dependencies between design decisions would be useful for helping the user understand what must go into a specification, for recording specifications made interactively, and for replaying revised specifications. For example, the dependency network helps the user understand that a particular decision may not even be relevant unless some other set of choices has been made. SINAPSE distinguishes between user-specified decisions and decisions inferred by the system based on those decisions. Only decisions in the first class need be recorded in the text-file specification. Decisions in the second class can be made again automatically if the specification is resubmitted. This argument assumes a static synthesis system. If more alternatives for a decision are added at a later date, the existing heuristics may no longer force a choice. Hence, it might also be useful to record the full history of inferences to help the user augment the specification in the face of an evolving system.

Currently, synthesis times are all under 20 minutes, and the decision making portions are usually on the order of minutes, so simply recording the primary decisions and recomputing the rest has been acceptable and it has not seemed necessary to build a full-fledged truth maintenance system. We do have a simple dependency network that records definitions and uses of synthesis facts. Because we wish to record decision dependencies for purposes of explanation, at some point the expense of building an incremental change system may be justified.

Because the user can help make implementation decisions, we also foresee a need for representing dependencies between user specifications. This general phenomenon of specifications accommodating to implementations is discussed in [Swa82]. One example that we have seen in SINAPSE is that a modeler may combine periodic and taper boundary declarations to implement an absorbing boundary condition when the target language is SIMD Connection Machine Fortran (to enable the use of an efficient circular shift operation). Even if a boundary isn't really periodic, the tapering operation makes the effective boundary value nearly zero on both edges, which means declaring the boundary to be periodic is not harmful. These dependencies should be recorded because if the target architecture is changed, we want to reconsider the choices of periodic and taper boundaries (even though both were user-specified) in the light of the new architecture.

Ordering Decisions

Users are sensitive to the order in which specification decisions are made; this order must make sense to them. Ordering is constrained by dependencies between decisions. In general, of course, the ordering of the decisions will follow the ordering of the phases described in the previous section, with implementation decisions such as data structure representations following problem set up specifications such as boundary conditions. However, some details can vary with the application. For example, in some cases all dependent variables may depend on the same independent variables so it might make sense to define independent variables first and then list dependent variables. In other cases, it might make more sense to define each dependent variable in terms of its specific independent variables. To support this, SINAPSE can present a different set of design decisions with alternative orderings for different applications.

Currently, when used in the interactive mode, the SINAPSE system presents the design choices in a linear sequence, and modelers do not always understand why a particular ordering is used. It would help considerably if we represented the *partial* ordering on the design choices, with a user interface that allows specification according to the partial ordering rather than an arbitrary linearization of that ordering. We do believe however that the system should explicitly present the decisions in the partial ordering rather than expecting the user to write the decision in arbitrary order in a text file or to navigate around a large collection of objects and to know what properties must be filled in or what commands must be issued. We plan to experiment with a graphical depiction of the decision network that is actively modified as choices are made.

Explanation

Representing information about decisions could help generate good explanations for how to set up specifications or why the system made the specific choices [WMK92; Swa83]. It both cases, a likely priority is: most heavily weight the choices involving problem description decisions (user choices before system choices), then the state of the implementation design so far, then the user's generic preferences, then the system's heuristic rules, and finally the system defaults.

Representing the Knowledge

The representation of knowledge in Sinapse has been discussed elsewhere [KDMW91] and so will not be repeated in detail here. We simply note that our goals for code generation and user interaction suggest that our knowledge representations be declarative, object-oriented descriptions of design choices and algorithm schemas. The object-oriented representation for design constructs includes the use of multiple inheritance,

with a small number of fairly flat hierarchies for algorithm type, application type, and so on. As discussed earlier, since the initial system design, the importance of more explicit goals and plans for the user interface has become clear. In addition to the declarative representations, there are procedural languages that can be used in describing programs: *Psiam* at the array level, and *MathCode* at the imperative level. The semantics of *Psiam* are still evolving; *MathCode* is the most mature and stable of all the representations. *Mathematica*'s pattern matching and symbolic simplifications are useful in defining transformation rules for both refinement (elaboration) and optimization. Recently we have also added a mechanism to record some of the major transformation steps (by transformation name and by before and after states). While we do not expect to record every single transformation step, we expect to eventually have more control over transformation applications; currently most are just anonymous *Mathematica* rules that fire whenever they match rather than being explicitly applied. Most likely there will be named sets of transformation rules that are applied at specific phases.

Evolving the Knowledge

To measure the evolution of knowledge in SINAPSE, for the past 16 months we have kept records about changes to the system. A regression test suite is maintained so that changes can be tested for compatibility and compared for performance. Although the records are only as good as the effort people put into keeping them and more careful analysis is need, some rough generalizations can be made.

Overall, the total system has grown steadily. The initial effort, before detailed records were kept, was mostly in adding domain knowledge and very primitive code generation knowledge. Since that time, we have focused on generating efficient code for multiple target languages and architectures, on adding domain knowledge that fills gaps in our application domain, and on making the system more understandable via additional explicit knowledge about design decisions and explicit representation of dependencies between decisions. There have been no huge waves of expansion and compression of the entire system representation, although individual components do grow and shrink as knowledge is added or more concisely represented.

Some basic information about size may give a general picture of the evolution of knowledge. The current system is now more than 20,000 lines of *Mathematica* code, a 38% increase over the system of 16 months ago. The declarative representations of the domain knowledge and problem-solving structure have grown the most – from 13% to 19% of the system, a 111% increase. There are currently about 100 types of design facts of the fill-in-the-blank form and 33 menu-choice decisions with an average of 3 alternatives. There are currently about 60 program-synthesis tasks; as well as adding new tasks, the ordering among the tasks has been refined over time. Procedural knowledge about how to refine domain descriptions to algorithms and coding constructs has grown only 15% and slipped from 41% to 35% of the system. (No count on the number of rules or functions is available. This is a place where the content of the knowledge has increased, but the representation has gotten more concise, so the overall growth looks low.) Knowledge about code generation has increased 35%, but as a percentage of the entire system held almost even, moving from 30% to 29%. (Much of the work that has gone into code optimization is not complete and is not reflected in the version of the system described here. The code-optimization techniques will add approximately 5,000 more lines of code.) The program-synthesis framework, while growing 54%, has only gone from 16% to 17% of the total system. The growth has been in the areas of mechanisms for the expanded knowledge about synthesis tasks and the recording of major transformation steps.

Of the 360 recorded changes to the system (in terms of number of entries, not number of lines of code or numbers of facts involved), 30% have involved changes to the internal representation or knowledge about the program-synthesis process, 15% have been changes visible in the human interface, 15% have been changes to domain knowledge, 35% changes to programming knowledge (reflected in the generated code), and 5% to the operating system interface. Overall, 20% of these changes were described as new capabilities, 24% as generalizations of existing capabilities, 20% as bug fixes, 5% as efficiency improvements, 28% as improvements in the clarity of the system or the code it generates, and 5% as other.

The frequent occurrence of changes to improve representation clarity reflects both improved understanding of the domain and deficiencies in the original representations of design goals and actions. Improvement is still needed in expressing dependencies between decisions, both the order required by the decisions, in terms of definition-use chains, and task-ordering preferences. We also expect it would be useful to be able to express a difference between hard constraints (forced choices), heuristics based on available information, and default choices (based on no information).

Analyzing the types of changes that are made should help us determine what sort of knowledge acquisition tools we should build. At present only a minimal number of rudimentary knowledge-building aids exist in SINAPSE. They help inspect the structure of synthesis tasks and dependencies and check for completeness of information about design decisions. Based on analysis of the changes and conversations with modelers, we have identified a small number of knowledge-acquisition activities that we would like to support more automatically for end users as well as for developers. These activities include the addition of new approximation operators, of variations on input/output handling, of new algorithm schemas, and the packaging of existing algorithms inside user-defined outer loops.

Sharing among Design Systems

The amount of knowledge required for automating software design is very large, even for quite restricted

classes of problems. The automated software design community would be likely to make faster progress if it explored the possibilities of reuse *among* design systems as well as reuse within a single domain-specific system. How do we design our systems to facilitate this sharing? Possibilities include reuse of system components (some domain-independent), reuse of reasoning algorithms, and reuse of interface languages (such as a *Psiam*-like array-level language). Similar proposals have been made before of course, such as generic tasks for expert system building blocks [Cha86], compositional modeling for engineering modeling [FF91], standardization work in the knowledge-representation community, and the suggestion of working out theories for program synthesis [Smi91].

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.

The reasoning-technique (shell) approach is another cut at providing tools. We might ask what sorts of tools for different reasoning strategies would be useful for automating software design. For example, SINAPSE could use someone else's inequality prover, or an outside tool for analyzing data flow, 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. Neverless, even if it requires reimplementation, a clearly specified set of tools and algorithms for accomplishing the goals of the tools should facilitate reuse.

Acknowledgements

Current and past members of the SINAPSE project who should be recognized for their work on the concepts and implementation of the system include Ira Baxter, Hung-Wen Chang, François Daube, Bill MacGregor, and Joe Wald. Many thanks to Ira Baxter and Ursula Wolz for comments on drafts of this paper.

References

H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. J. Sussman, J. Wisdom, and K. Yip. Intelligence in Scientific Computing. *Communications of the ACM*, 32(5):546–562, May 1989.

B. Chandrasekaran. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert*, 1(3):23–30, Fall 1986.

G. O. Cook. ALPAL, a Program to Generate Physics Simulation Codes from Natural Descriptions. *International Journal of Modern Physics*, 1(1):1–55, 1990.

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

E. Kant. Human and Computer Responsibilities in Program Synthesis. In *Workshop Notes–Knowledge-Based Human-Computer Communication*, pages 65–67, Stanford, CA, March 1990.

E. Kant, F. Daube, W. MacGregor, and J. Wald. Scientific Programming by Automated Synthesis. In M. R. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 8, pages 169–205. AAAI Press/The MIT Press, Menlo Park, CA, 1991.

R. S. Palmer and J. F. Cremer. SIMLAB: Automatically Creating Physical Systems Simulators. Technical Report TR 91-1246, Department of Computer Science, Cornell University, Ithaca, New York, November 1991.

D. Smith. Theory-Based Support for Software Development. In *Workshop Notes–Automating Software Design: Interactive Design*, pages 162–165, Los Angeles, CA, July 1991.

W. Swartout. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

W. Swartout. XPLAIN: A System for Creating and Explaining Expert Consulting Systems. *Artificial Intelligence*, 21(3):285–325, September 1983.

U. Wolz, K.R. McKeown, and G.E Kaiser. Automated Tutoring in Interactive Environments: A Task-Centered Approach. In M.J. Farr and J. Psotka, editors, *Intelligent Instruction by Computer, theory and practice*. Taylor and Francis, Washington DC, 1992.

S. Wolfram. *Mathematica: a System for doing Mathematics by Computer*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.