Practical Issues in Building Knowledge-Based Code Synthesis Systems

Ira Baxter

Schlumberger Laboratory for Computer Science
Austin, Texas, 78720 USA
Tel: 1-512-331-3714
Fax: 1-512-331-3760
Email: baxter@austin.slcs.slb.com

Abstract

We present some issues and lessons on building a practical "generative reuse" system. We discuss the cost of building a useful system, and the tensions between prototyping and production systems, often induced by scale demands.

Keywords: Synthesis, transformation, generative, knowledge-based systems

Workshop Goals: Exchange ideas on capture and mechanized reuse of design knowledge. Particular interest in learning about other work related to knowledge acquisition for program generation.

Working Groups: reuse and formal methods, reusable component certification, domain analysis/engineering

1 Author's Background

Much of the work in reuse is focused on code reuse. In contrast, much of my work is related to reuse of design and analysis information in a formal context, in an effort to enhance software maintenance.

The Draco tool [1] reused composable components resulting from domain analysis. The experience of the Advanced Software Engineering group at UC Irvine with Draco led to ideas about porting software by partial domain analysis, component abstraction, and re-implementation [2].

Work on *Design Maintenance Systems* (DMS) [3] defined how to generate and capture a complete design rationale for a transformationally-derived program in terms of specified performance predicates. The design rationale was reused (and revised) to control the modification of the synthesized program according to formal *maintenance deltas*, representing desired specification changes. Code intended for reuse often needs to be modified, but few concrete methods for carrying out that modification have been studied. *The DMS strategies can be viewed as an explicit prescription for how to modify code in a particular reuse instance.*

My current work is on developing infrastructure needed by the Sinapse system to generate efficient parallel programs. This work shows the value of both declarative capture of potential parallelism in abstract components [4], and the utility of conventional compiler optimizations such as parallelization detection and common subexpression elimination on abstract domains rather than low-level (*i.e.*, Fortran) synthesized code.

2 Position

Building knowledge-based tools to aid generative reuse is a difficult process. Both theory and practical issues must be addressed. Many of the theoretical issues are already addressed in the literature: domain analysis and evolution [5], transformational development [6], etc.

This position paper discusses a number of practical issues, using our experience developing Sinapse as examples. A good software engineering process ought to identify and address these issues. However, since building generative reuse systems is not yet a common SE task, we decided to risk stating the obvious for would-be reusers of our experience.

2.1 Issues

We will discuss the following issues:

- Construction costs
- Scaling up for real synthesized codes
- Ad hoc knowledge vs. algorithms
- Coding vs. specifying optimization tasks
- Knowledge acquisition tools vs. knowledge capture
- Leveraging the synthesis tools
- Testing system robustness

We first sketch Sinapse to provide necessary background.

2.2 Sinapse Overview

The Schlumberger Laboratory for Computer Science has developed a system called Sinapse [7] [8] [9] to synthesize mathematical modeling programs for restricted application domains. These are primarily wave propagation problems, typically used to validate seismic or geophysical models. Sinapse accepts a 20-50 line specification which defines a set of partial differential equations (PDEs) to be solved, either directly by stating the equations, or indirectly by use of domain-specific terminology. Sinapse produces complete C, (sequential) Fortran, or (data parallel) Connection Machine Fortran programs that solve the PDEs in a particular geophysical context by using finite differencing methods, including the necessary input and output code. Resulting programs are 500-1500 lines in size.

Sinapse is a knowledge-based synthesis system, containing knowledge about the wave propagation problem domain, knowledge of abstract algorithms for solution techniques for problems in that domain, parallelization methods for those codes, general programming knowledge, and control knowledge to sequence the synthesis process. The domain knowledge is used to choose abstract solutions by classifying domain-specific specifications. These solutions are transformationally refined into a production code. The system stores synthesis sequencing knowledge as a set of hierarchical, nonlinear plans. Abstract algorithms are stored in a mixed form consisting of abstract syntax trees (ASTs) with procedural attachments. Decisions about branches in the design space are handled by built-in system constraints, heuristics and defaults; because we always expect the system to have limited knowledge, the software engineer operating Sinapse can override any heuristic decision. Refinements consist of tree-to-tree rewrite rules coupled with execution of the procedural attachments. The refinement process is reminiscent of Draco's repeated refine-to-domain, optimize-within-domain method. Optimizers based on conventional compiler technology concepts were implemented to allow optimizations in abstract domains; this allows Sinapse to detect common subexpressions and control parallelism by analyzing the set of PDEs rather than expecting a Fortran compiler to discover these in the final output.

3 Construction Costs

The development of a production-quality synthesis system, even when limited to addressing restricted domains, and reusing existing tools, can easily consume 15-20 person-years.

Sinapse today consists of 37,000 lines of code (LOC) built on top of the commercial Mathematica [10] system. It was developed at the cost of 12 person-years, spread over 4 elapsed years. We intend to reach a production system that can be reliably used by physicists and engineers this year.

To provide the barest indication of where the effort goes, we include a system breakdown as an LOC percentage of subsystem vs. total system:

5%	Textual User Interface
9%	Graphical User Interface
5%	Synthesis control
25%	Domain-specific synthesis knowledge
5%	Local optimizer: type inference, simplification
16%	Conversion of ASTs to target languages: C, Fortran, CM Fortran, Lisp
27%	Global optimizer: parallelization, code motion, CSE
4%	Computation partial order management (used by optimizer)
4%	Utilities/debugging tools

We caution the reader not to interpret these percentages as effort, because the tasks had differing complexities, and considerable activity occurred outside of coding.

It should come as no surprise that much of the effort went into domain-specific knowledge acquisition. It may be more of a surprise that significant effort went into domain-independent optimizing technology.

4 Scaling up for Realistic Codes

One must design synthesis systems for performance with scale. We address how prototyping and knowledge interact with scale.

4.1 Prototyping vs. Production

Well-intentioned reuse of existing tools may make scaling more difficult.

As a system starts to show promise of utility, ambitions turn from small demos to practical codes, and scale difficulties begin to show up. An initially attractive system foundation may have other properties which exact a relatively large cost in comparison to the expected benefits. One must consider these benefits carefully; engineering for scale may require paying higher costs during the early stages of development.

As an example, we built Sinapse on top of a symbolic mathematical system, Mathematica (MMa), because it had several attractive properties from a prototyping point of view. It provided a single program development environment which contained considerable built-in knowledge useful for manipulating symbolic (algebraic and differential) equations, a procedural programming language, and a transformation system for free. It promised portability of the result because MMa ran on several platforms, and it was interactive, which eased debugging by making inspection simpler. Early versions of Sinapse benefited because the availability of these mechanisms made it easier to prototype parts of the system, thereby selling the system concept via small demos.

Growing sophistication caused Sinapse's internal algorithm form to become more specialized. It became progressively more difficult to keep a form compatible with that of the underlying MMa system. Eventually, we gave up direct compatibility, opting for conversion to MMa's form when we needed MMa's mathematical knowledge, and then converting the result back to the Sinapse form. Happily the number of conversions per run have turned out to be infrequent. Obviously, we wish to (re)use the knowledge in a symbolic algebra system; the need for our own representations hints that staying within the environment defined by such a tool is not necessarily required. Ideally, we'd like to have a way of compiling symbolic manipulation knowledge stored in a knowledge-representation language such as KIF [11] into a form compatible with our desired representations. This is itself a program synthesis problem we don't know how to solve.

The MMa tree-to-tree rewrite system works well on small trees representing "typical" size mathematical formulas. Unfortunately, MMa's built-in control strategies work slowly on large sets of transformations (100's of rules) on large ASTs (10K tree nodes) which are typical of the size of the codes we synthesize. Further, MMa is an interpreter without any corresponding compiler, costing us an additional runtime factor of 10x. Synthesis times without optimization run to 30 minutes on a Sparc 2 and can be much longer with the optimizations. This has made building and testing the Sinapse system components relatively painful. If one insists on using ASTs, a tree-to-tree transformation system engine is cheap to replicate compared to the investment in a synthesis system; we would recommend instead looking into more efficient production system tools like OPS5 [12].

Even the choice of ASTs is called into question by scale. Most "purist"" transformation systems, MMa being no exception, operate by manipulation of ASTs. However, serious optimization of refined code requires that dataflow analysis be performed; ASTs are a poor choice for this. The lion's share of the cost in the optimizer is the need to recompute the data flow analysis information after an optimization has been made. An approach we are pursuing is the use of compiler representations such as Static Single Assignment form [13], in which the data flow has been made explicit, and optimizations directly transform the dataflows, keeping them up to date. An ideal solution would be to treat the data flow analysis problem as a problem in incremental re-computation, and use finite difference methods (unfortunately, this term is common to both PDE solving and program synthesis but doesn't mean the same thing) to keep dataflows current [14].

Many difficulties can be traced to the tension between prototyping and production. *Synthesis systems are controversial enough so that the need to prototype them early can predispose evolutionary descendents to later scale troubles.* For conventional SE tasks, we know that prototypes are a poor foundation for production code; for knowledge-based systems, evidence that a prototype works should similarly not be treated as evidence that the prototype can be easily extended to a production code by mere incremental addition of knowledge.

4.2 Ad Hoc Knowledge vs. Algorithms

Special cases may sometimes be more easily handled algorithmically. Knowledge-based methods should be reserved for cases where the algorithms fail because of intractability or lack of information. System designers should be prepared to pay the price of installing the appropriate algorithms when scaling up.

Sinapse uses some simple methods to effect certain kinds of optimizations. To ensure that reasonably high-performance code is generated, a kind of knowledge-based code motion is used to move initialization code for refined components out of loops. Other special mechanisms were installed to allow results of computations in generated programs to be stored and reused at later points. These mechanisms have the advantage of being easy to define and install, which is attractive for small examples.

Both of these mechanisms are subsumed by conventional compiler optimization techniques that couple code motion with common subexpression elimination. The knowledge-based versions are somewhat fragile, and KB encoders of algorithms had to think carefully about how to use them. The compiler methods simply work without thought; this makes the system more robust, and it makes KB augmentation faster because less has to be explicitly encoded.

There will still be cases where compiler methods fail because the cost of inference is simply too high. In these cases, knowledge can play a useful role.

5 Coding vs. Specifying Optimization tasks

Since many synthesis systems will have unique internal representations, there should be a way to manufacture conventional optimizers easily.

Program specification is about defining what a program will do. Since, given a specification, one can simply try enumerate-and-test, program synthesis is only about optimization. For scale purposes, conventional optimizers seem necessary, but building them by hand is expensive. Much of the code in an optimizer consists of routines that interpret how information flows across various syntactic constructs.

While this can be encoded by hand, we found that as Sinapse's internal form evolved and grew in complexity, the optimizer had to evolve in lock step. We often wished that we could instead specify something like the denotational semantics [15], and "compile" that into practical optimizers. In essence, we need an optimizer generator. Basic research is needed to accomplish this task.

Another possibility is for the synthesis community to agree on a widely usable internal form. This would allow tools to be made generically available. We believe this avenue is unlikely because a long search for UNCOL has largely been unsuccessful, and we expect that the utility of internal forms will come from their domain specificity.

6 Knowledge Acquisition Tools vs. Knowledge Capture

System designers must choose between capturing knowledge themselves and defining knowledge acquisition tools. Experts often have little patience for unfamiliar detail. Failure to get them directly involved in knowledge capture can limit the rate at which a system's ability grows.

Much of Sinapse's knowledge is coded in MMa notation for defining trees or as MMa procedures. This required that encoders know MMa well, and made it difficult to prevent some low-level implementation details of Sinapse from leaking into the components. Attempts to get wave propagation experts, who were expert parallel machine programmers, directly involved in the knowledge encoding task were not very successful because of the high cost of learning the system. The system designers have had to substitute for the experts when encoding the knowledge, making the designers a bottleneck in the acquisition process.

Getting the experts involved requires that they be taught basic principles of knowledge representation and domain analysis. The system designers should spend their energy building knowledge acquisition tools, to allowing the experts to easily find out what the system already knows, and revise that knowledge, using notational schemes which are familiar [16], [17].

7 Leveraging the Synthesis Tools

We should try for an "avalanche" technology, in which the tools can be used to enhance themselves. Much of a synthesis tool is focused around generation and optimization of mundane code. It would be ideal if such mechanisms could be used to help generate more of the synthesizer code.

Much of the Sinapse system was built by conventional coding methods. Considerable leverage might have been obtained if we could have used such synthesis tools to build part of Sinapse itself. As an example, data flow analyzers on arbitrary internal forms might have been built in this fashion.

8 Testing System Robustness

A system must have a methodology for testing and configuration management to ensure robustness.

We successfully used some conventional SE practices to help ensure the robustness of Sinapse. Modules of the optimizer were tested dynamically on each system load during development. RCS, a source-code control tool, was augmented with a number of procedures to ensure that the base system had always been regression-tested, and that differences in test results from previous runs had been validated by team members. These methods ensured that the developers were always working with trusted components.

9 Conclusion

We have presented some issues that must be faced by designers of generative reuse systems. Generative systems appear to cost a lot to build; there must be considerable payoff to justify building one. The classic tensions between the need to prototype cheaply to sell a system concept and the need to design for scale will probably appear as repeated development. Algorithmic methods should be used where known, and augmented with knowledge-based methods where they fail. Generation of optimized code requires the construction of complex optimizers; abstract syntax trees seem to be a weak representation for this purpose, and we have few tools to help us build even conventional optimizer mechanisms.

Most of these issues are related to effects of scaling up, which cannot be avoided for most practical systems.

References

[1] J. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, vol. SE-10, Sept. 1984.

[2] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software Maintenance by Transformation," *IEEE Software*, vol. 3, pp. 27–39, May 1986.

[3] I. D. Baxter, "Design Maintenance Systems," *Communications of the ACM*, vol. 35, pp. 73–89, Apr. 1992.

[4] I. D. Baxter and E. Kant, "Using Domain-Specific, Abstract Parallelism," in *Proceedings of Parallel-Code Generation Workshop at ICLP91*, IEEE Computer Society, Oct. 1991.

[5] G. Arango, *Domain Engineering for Software Reuse*. PhD thesis, Department of Information and Computer Science, University of California at Irvine, July 1988. ICS-RTP-88-27.

[6] H. A. Partsch, *Specification and Transformation of Programs*. Springer-Verlag, 1990. ISBN 52356-1.

[7] E. Kant, F. Daube, W. MacGregor, and J. Wald, "Automated Synthesis of Finite Difference Programs," in *Symbolic Computations and Their Impact on Mechanics, PVP-Volume 205*, New York, NY: The American Society of Mechanical Engineers 1990, 1990. ISBN 0-791800598-0.

[8] E. Kant, F. Daube, W. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," in *Automating Software Design* (M. Lowry and R. McCartney, eds.), AAAI Press, 1991.

[9] E. Kant, "Synthesis of Mathematical Modeling Software," *IEEE Software*, vol. 10, pp. 30–41, May 1993.

[10] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1988. ISBN 0-201-19334-5, QA76.95.W651988.

[11] M. R. Geneserth and R. E. Fikes, "Knowledge Interchange Format Version 3.0 Reference Manual," Tech. Rep. Logic Group Report Logic-92-1, Computer Science Department, Stanford University, June 1992.

[12] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley, 1985. ISBN 0-201-10647-7.

[13] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[14] D. R. Smith, "KIDS: A Semi-Automatic Program Development System," tech. rep., Kestrel Institute, Palo Alto, California 94304, Oct. 1989.

[15] F. G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer.* Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1981. ISBN 0-13-329052-2.

[16] S. Marcus, "SALT: A Knowledge Acquistion Tool for Propose-and-Revise Systems," in *Automating Knowledge Acquisition for Expert Systems* (S. Marcus, ed.), pp. 81–123, Boston: Kluwer Academic Publishers, 1988.

[17] J. Boose and B. Gaines, *The Foundations of Knowledge Acquisition, Vol. 4.* New York: Academic Press, 1990.

10 Biography

Ira D. Baxter is a research scientist with the Schlumberger Laboratory for Computer Science in Austin, Texas, working on tools to help engineers generate scientific modeling programs for parallel computers. He received a Ph.D. in Computer Science from the University of California at Irvine in 1990. A previous life included 20 years of system software design for mini- and micro-computers. His interests include program synthesis/transformation, software engineering, and pizza.