Marti A. Hearst, Editor
University of California, Berkeley
hearst@sims.berkeley.edu

Will domain-specific code synthesis become a silver bullet?

Wray Buntine, UC Berkeley and Ultimode Systems, Guest Editor

As a young NASA scientist looking for ways to apply machine-learning technology, my saddest moments came when I saw potential applications slip away. Shortages of experts, management shuffling, and interdepartmental conflicts—those common application-killers—were not always to blame. Sometimes, after I had spent weeks scoping out a particular application with the expert and found that my technology matched perfectly, I'd still come up empty. Software-engineering costs would kill an otherwise ideal application. These defeats were tough to accept, because I could see great solutions looming just over the horizon.

In the commercial world, I now find that the biggest hurdles to delivering embedded intelligence in software are again the software-engineering costs. We members of the intelligent-systems community have brilliant techniques available, and no end of applications is in sight, but the gap between intelligent-systems techniques and delivered software is too great. Standard programming languages and tools are too general-purpose, and the standard means of delivering a technology—as code libraries or new-fangled *componentware*—is simply too restricting for some specialties.

The skills many of my methodologically oriented colleagues have honed—in learning, optimization, and image analysis, for example—involve designing and refining algorithms. But there is no commercially accepted means for delivering such skills. I earnestly hope that someday we will be able to deliver our algorithm-design skills. One way would be through domain-specific code synthesis. Support for the development of domain-specific extensions to programming languages and of *adaptive software*—where code synthesis is in the inner-loop of a self-modifying programming system—would have a profound effect in this regard. This is the future of intelligent-systems development!

Recent years have seen a significant groundswell of activity in this area, so much so that someone like me, with absolutely no formal training in automated software engineering, programming languages, or compilers, can be asked to organize a discussion on this critical technology—as Marti Hearst has asked in this case. These days, scientists with no specialized training in computer science are developing code-generation techniques for their own classes of languages, while computer scientists with no formal training in automated software engineering are developing automated software-engineering tools.

In this month's installment of "Trends & Controversies," we will hear from three different groups or individuals at the forefront of these developments. These authors cover the full gamut of motivations, commercial settings, and research backgrounds.

Peter Norvig, a well-known voice in our community, presents his perspective as the chief scientist of Junglee, a start-up software development company. Automated software engineering appears necessary to gain a commercial advantage. Jeffrey Van Baalen, a NASA principal investigator, presents his view of some of NASA's goals. Jeff does come with an automated-software-engineering background, and we will see that Jeff's group is able to deliver its technology in a domain-specific context, avoiding the pitfalls suffered by earlier proponents of formal methods. Finally, David Spiegelhalter, a respected biostatistician at Cambridge's Medical Research Council, and Andrew Thomas of the Imperial College, London, present a rather astonishing system that supports statisticians, based on the special-purpose language of probability networks.

We have no real naysayers in this issue's discussion, but if you are a disbeliever, I hope our three discussants will help change your mind.

Code synthesis as enabling technology for mass customized agents

Peter Norvig, Junglee Corp.

I define code synthesis as any process that takes a representation of a problem as input and produces a representation of a function as output (see Figure 1). Many technologies in widespread use fit this broad definition: compilers, macro processors, compiler-compilers such as YACC, and special-purpose languages such as VHDL. Today, these tools all seem routine, but remember that at one time they were not; the first Fortran translator was originally called an automatic programming system, not a compiler.

To eliminate the mundane, this essay considers only cases where the transformation is *surprisingly good*. Either the representation of the problem is unusually succinct, unconstrained, or high-level; or the resulting code is unusually accurate or makes especially good use of resources. Clearly, *surprisingly good* is a moving target. Over time, we will see a gradual rise in people's expectations of their tools. Today, we expect compilers to inline short func-

tion calls, integrated development environments to synthesize window event-processing code, and runtime systems to manage memory allocation. Next year, we will expect more. I predict that we will also see specialized domains where more radical code synthesis will revolutionize how work is done. This happened with spreadsheets, which opened up analytical processing to a new class of nonprogramming users. In the coming years, the same sort of revolution will appear around agent programs.

Problem representations

There are many ways to describe a problem. We can use axioms to logically describe the relationship between input and output (the *theorem proving* approach). We can describe the problem as a set of constraints on schedules and resources (the *planning* approach). We can give sample input/output pairs (the *statistical* approach). Finally, we can provide a real or simulated environment—and some time to practice in it—and define an objective function to optimize (the *agents* approach).

Theorem proving. Succinct problem descriptions can often be given as a set of logical axioms. For example,

```
Sort(X, Y) :- Permutation(X,
Y), Ordered(X).
```

says that `Y` is the result of sorting `X` if `Y` is a permutation of `X` and `Y` is an ordered sequence. (We would need additional axioms for `Permutation` and `Ordered`.) This is a logical definition of the sorting problem, and it is also an executable Prolog program, but it is a very inefficient program. Some theorem provers, however, can transform a specification such as this into a more efficient program.[1,2] Although such systems can produce programs with thousandfold speedups over hand-coded solutions, they are not widely used because the user requires some mathematical sophistication (such as understanding what an axiom is) and because initially specifying a domain incurs a large cost.

One pervasive kind of code optimization is *partial evaluation*, in which a general program specification is specialized by filling in particular parameters, and the resulting code propagates the constraints imposed by the parameter.[3] For example, given a general exponentiation routine,

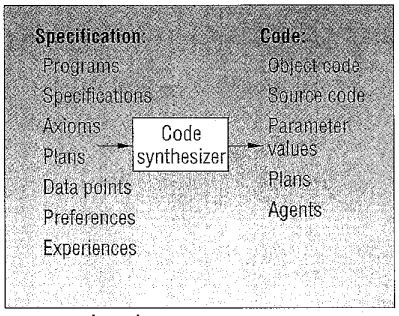


Figure 1. Code synthesis.

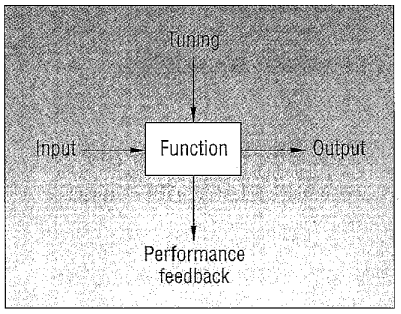


Figure 2. Adaptive-software approach.

`expt(x,n)`, a partial evaluator could compile `y=expt(x,4);` into `y=x*x; y=y*y`, in effect unrolling the loop inside `expt`. Partial evaluators are often used as part of *domain-specific languages*,[4] which are designed to include exactly those abstractions that are useful in a particular domain. A program in a domain-specific language has the feel of high-level specification but is executable by conventional means (including partial evaluation, higher-order functions, and syntactic abstraction).

Planning. A great deal of practical work has gone into solving problems that are expressed as a set of constraints on goals and tasks, and the time and resources they require.There are many general algorithms for planning and scheduling, and although they all have dismal worst-case performance bounds, in practice you can usually find a good algorithm for a particular problem domain. The trick is in choosing the right algorithm, but this can often be done by an automatic[5] or user-assisted[6] search through the space of possible planning algorithms.

Statistical. What happens if you don't have a clear problem specification and so cannot generate candidate solutions? For example, suppose you are trying to write a program to predict the weather or the stock market. You won't have a complete theory of the domain, but you will have a good set of historical data. A nonlinear regression routine, a neural network, or one of many other machine-learning techniques can synthesize a function that fits the set of data points and that you can use to make predictions. Statistical function approximation and prediction is widely used, but it is not normally considered a code-synthesis approach, because people hesitate to accept code they know is only an approximation.

However, use of the statistical approach as a component of a larger system is increasing, particularly when making the wrong prediction leads only to a slower program, rather than an incorrect one. For example, a memory-allocation routine (such as `malloc`) could perform better if it knew how long the allocated memory would be alive before being freed. Decision trees have been used to accurately predict the lifetime of objects.[7] This automatic-learning version of `malloc` performed better than the standard `malloc`, and better even than very sophisticated handcrafted allocation routines. Similar work has been done for choosing an efficient data-type implementation, doing branch prediction, scheduling straight-line code on a parallel machine, and accomplishing other tasks.

Agents. Currently, the World Wide Web is a collection of powerful server programs that mostly point to, but do not communicate with, each other. Interaction is generally driven by a human sitting at a browser client, except for a small number of specialized servers, such as AltaVista, that aggregate information from other servers. We are just beginning to see tools to synthesize domain-specific or personalized software agents that automate the tasks a human must now do. For example, in Yahoo's shopping guide,[8] a user can search for the title of a book and see all the offerings of that title by the major online booksellers, presented on one page in a common, easy-to-compare format. It would take a hundred clicks or so to gather this information manually. The technology behind this differs from traditional search engines in that there is a custom agent to gather information from each Web site (Amazon, Barnes & Noble, and so forth), rather than one generic program that indexes the whole Web.[9]

Today, developers write each custom agent manually, using a specialized toolkit.

But because this manual coding is time-consuming, Junglee is investing in code-synthesis technology to automatically generate custom agents for each new Web site. The idea is to define a specification of the book-shopping domain in particular and of Web-site structure in general. Then, when faced with a new online bookseller, we can run an agent-synthesis program that automatically discovers the proper way to interact with the Web site to search for books. Synthesizing such an agent is essentially a complex grammar-induction problem, driven by knowledge of Web protocols (HTTP and HTML, for example), Web-site structuring principles, and natural-language expressions that are used in the domain. Other researchers are working on synthesis of natural-language extraction agents.[10]

Function quality

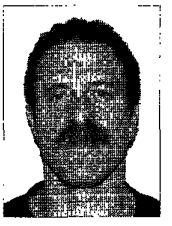
Now that we've seen some alternative representations of a problem specification, let's consider what it would mean for the synthesized code to be surprisingly good. First, the execution speed or memory use could be surprisingly good, as in the 90-fold speedup achieved through partial evaluation by Andrew Berlin and Daniel Wiese.[3] Second, the code could be more nearly optimal than expected, as in the lifetime prediction module that achieved 99.9% accuracy on some programs.[7] Third, the programmer effort to develop and maintain the program could be reduced.[4,9] This is probably the most important aspect.

Code synthesis might prove to be a key silver bullet for software engineering—not because all programs will be automatically synthesized, but because synthesis might be the only way to efficiently combine other software pieces. A well-designed class, framework, or component consists of a basic algorithm or algorithms and a series of implementation trade-offs, all packaged as an abstraction. One thing that is abstracted away is performance considerations, which sometimes makes the abstraction unusable. This is especially true when we build a large system out of a tower of abstractions.

Traditionally, the only solution is to break the abstraction barrier and write new code rather than reuse existing code. The adaptive-software approach (see Figure 2) separates a function's description and run-time operation into two orthogonal axes.[11] The horizontal axis gives the traditional input/output specification; the vertical view gives a performance specification: it describes the implementation choices and parameter settings that can be altered, and lets you measure performance at each setting. We can use adaptive software by

- Composing a program using the input/output specifications;
- Analyzing the performance characteristics, either analytically (by looking at the information in the vertical axis) or empirically (by running the program on test data); or
- Synthesizing a more efficient program by searching through the space of different implementation choices for one that performs well. Companies such as Harlequin[12] and Microsoft[13] are actively pursuing this approach.

Peter Norvig is chief scientist at Junglee Corporation. His interests span all of artificial intelligence, with an emphasis on practical information extraction and integration from text sources. He received a BS in applied mathematics from Brown University and a PhD in computer science from the University of California, Berkeley. He recently published *Artificial Intelligence: A Modern Approach* (Prentice Hall, 1995). He is a member of the ACM and AAAI, and a board member of the Evaluation of Intelligent Systems online resource. Contact him at Junglee Corp., 1250 Oakmead Parkway, Sunnyvale, CA 94086; norvig@junglee.com; http://www.norvig.com.



Jeffrey Van Baalen is a principal research scientist at NASA Ames Research Center and an associate professor of computer science at the University of Wyoming. His interests include automated software engineering, automated deduction, and artificial intelligence. He received his BS and MS in computer science from the University of Wyoming and his PhD in electrical engineering and computer science from MIT. Contact him at MS 269-2 Code IC, NASA Ames Research Center, Moffett Field, CA 94035; jvb@ptolemy.arc.nasa.gov.

David Spiegelhalter is a senior statistician at the Medical Research Council Biostatistics Unit in Cambridge, UK. His main research area has been Bayesian reasoning, whether applied to statistics, artificial intelligence, or health-technology assessment. He received an MA in mathematics from Oxford University and a PhD in mathematical statistics from London University. Contact him at MRC Biostatistics Unit, Inst. of Public Health, Univ. Forvie Site, Robinson Way, Cambridge CB2 2SR, UK; david.spiegelhalter@mrc-bsu.cam.ac.uk.

Andrew Thomas is a software engineer at Imperial College, London. His main research area is designing and implementing software for probabilistic reasoning. He received a BA in natural sciences from Cambridge University and a PhD in theoretical physics from Liverpool University. Contact him at the Dept. of Epidemiology and Public Health, Imperial College School of Medicine at St Mary's, Norfolk Place, London, W2 1PG; andrew.thomas@ic.ac.uk.

Wray Buntine is a research engineer with the CAD group in the Electrical Engineering and Computer Science Department at the University of California, Berkeley, and vice president of R&D at Ultimode Systems. His research interests include probabilistic machine-learning methods and theory, data mining, graphical models, and inductive-logic programming. He received a BS in pure and applied mathematics from the University of Queensland and a PhD in computer science from the University of Technology, Sydney. Contact him at UC Berkeley, EECS Dept., Cory Hall, Room 550, Berkeley, CA 94720-1770; wray@ultimode.com; http://www-cad.eecs.berkeley.edu/~wray/.

References

1. "Specware," Kestrel Inst., Palo Alto, Calif., 1997; http://www.kestrel.edu/HTML/prototypes/specware.html.
2. M. Kaufman and J. Moore, "A Computational Logic," Univ. of Texas, Austin, 1997; http://www.cs.utexas.edu/users/moore/acl2/index.html.

3. A. Berlin and D. Weise, "Compiling Scientific Code Using Partial Evaluation," *Computer*, Vol. 23, No. 12, Dec. 1990, pp. 25–37; see also ftp://quilty.stanford.edu/pub/fuse-papers/README.html.

4. P. Hudak, "The Promise of Domain Specific Languages," keynote address from Usenix DSL Conf., Yale Univ., New Haven, Conn., 1997; http://www.cs.yale.edu/users/hudakdir/dsl/index.htm.

5. S. Minton, *Machine Learning Methods for Planning*, Morgan Kaufman, San Francisco, 1997; http://www.mkp.com/books_catalog/1-55860-248-8.asp.

6. D. Smith and S. Kambhampati, "Automated Synthesis of Planners and Schedulers," Kestrel Inst., 1997; http://www.kestrel.edu/HTML/projects/arpa-plan2/index.html.

7. D. Cohn and S. Singh, "Predicting Lifetimes in Dynamically Allocated Memory," *Advances in Neural Information Processing Systems 9*, M. Mozer et al., eds, MIT Press, Cambridge, Mass., 1997; see also http://www.ai.mit.edu/people/cohn/memory.ps.

8. Visa Shopping Guide by Yahoo!, Yahoo! Inc., Santa Clara, Calif., 1998; http://shopguide.yahoo.com/.

9. "Junglee Technology," Junglee, Sunnyvale, Calif., 1997; http://www.junglee.com/tech/index.html.

10. S. Soderland, "Learning Information Extraction Rules for Semi-Structured and Free Text," draft paper, Univ. of Washington, Seattle, 1997; http://www.cs.washington.edu/homes/soderlan/WHISK.ps.

11. P. Norvig and D. Cohn, "Adaptive Software," *PC AI Magazine*, 1997; http://www.norvig.com/adapaper-pcai.html.

12. "Adaptive Systems Group," Harlequin Inc., Cambridge, Mass., 1998; http://www.harlequin.com/products/asg/asg.html.

13. "Decision Theory and Adaptive Systems," Microsoft, Redmond, Wash., 1998; http://www.research.microsoft.com/dtas/.

Coming Next Issue

Sketching Intelligent Systems

Meta-Amphion: cost-effective development of high-assurance software generators

Jeffrey Van Baalen, NASA Ames and the University of Wyoming

The need for rapid generation and verification of highly complex software has pervaded many government and industrial settings, particularly so for NASA. NASA missions inherently need high assurance. In some cases, lives depend on the software: in every space shuttle flight, for example, astronauts' lives depend on the flight-control software. In other cases, the software must be autonomous: if it malfunctions, it's too far away to fix—as with the recent Mars Pathfinder Rover (see Figure 3). Software is increasingly difficult to develop; to be cost-effective, most future software development and verification will need to be at a higher level than is practiced currently. This problem is particularly acute for high-assurance software.

We plan to demonstrate this data-understanding program-synthesis tool in the synthesis of software for onboard spacecraft data analysis and summarization coordinated with remote-sensing/prospecting robots.

Domain-specific software generators

By automatically generating programs from problem specifications, DSSGs offer a promising approach to elevating the level of software development. For instance, the Amphion/NAIF system, developed by the Automated Software Engineering Group at NASA's Ames Research Center, assists space scientists with solar system observation opportunity analysis. NASA's space scientists used this type of analysis, for instance, when planning the use of the Cassini spacecraft to investigate the composition of Saturn's rings. For this purpose, they designed experiments in which a radio signal was to be transmitted through the rings and received on Earth.

The Amphion/NAIF system allows space scientists with limited or no computer science background to construct opportunity-analysis programs of this type.[1] An end user employs the Amphion/NAIF graphical tool to specify a desired observation geometry. Amphion/NAIF generates a Fortran program to compute values based on the geometry or to search for occurrences of the geometry such as, "When will Cassini be in position to transmit a signal through the rings of Saturn so that the signal can be received on Earth?"

Amphion/NAIF includes an animation component that produces scientifically realistic simulations of the programs it generates. Space scientists use these animations in their analyses.

DSSGs speed the development of application software because they enable non-computer science experts to automatically generate this software from problem specifications. Rather than developing and debugging programs, DSSG users develop and debug problem specifications that describe what a system should do, rather than how the system should do it. The software lifecycle rises to a problem-specification lifecycle.

Before the Amphion/NAIF system came along, space scientists had to write Fortran programs to assist themselves in solar system observation opportunity analysis. With Amphion/NAIF, they can do analysis without knowing anything about programming. They need only know about space science. Our experience has shown that space scientists who are expert Fortran programmers can generate observation opportunity analyzers an order of magnitude faster with Amphion/NAIF than by writing Fortran programs directly. Obviously, the time savings are far more dramatic for scientists who are not expert Fortran programmers.

Unfortunately, DSSGs are themselves difficult to construct. All the problems associated with software development have been elevated from the domain level to the software-generator level. Most existing DSSGs rely on nonoptimizing compiler technology. Consequently, these systems generate software efficiently but are expensive to build and cannot guarantee the assurance level of the code they generate.

Deductive-synthesis technology

The Amphion/NAIF system is unique among DSSGs because it relies on *deductive-synthesis* technology.[2] In deductive syn-

thesis, a problem specification is described as a theorem, and an automatic theorem prover generates a program as a byproduct of proving this theorem. Deductive-synthesis technology enables the construction of high-assurance software generators.

So why don't all DSSGs incorporate deductive-synthesis technology? First, in using deductive synthesis, a software generator is developed by constructing a declarative domain theory, which is a formal description of a domain. In principle, such theories should be much easier to develop than developing a DSSG using compiler technology. However, in practice, domain-theory notation requires a great deal of expertise to understand. Second, tuning a domain theory for deductive analysis usually requires considerable time and expertise. Each step in the tuning process preserves the correctness of the synthesis but makes it more efficient.

Three tools, collectively called Meta-Amphion, are under development at NASA Ames. They address the problems with deductive-synthesis technology and will make cost-effective construction of high-assurance software generators a reality:

- We are developing an environment that helps domain experts create usable domain theories without necessarily knowing domain-theory notation.
- We are developing a tool that automatically specializes a deductive-synthesis system by transforming naively developed domain theories into theories that are tuned for synthesis.
- The general-purpose approach lets users generate software that is provably correct relative to a domain theory. Hence, the problem of verification focuses on the correctness of the domain theory. Domain theories are easier to develop and verify than software. However, to make this process even easier, we are developing a mechanism to help identify errors in domain theories.

We have already made significant progress on all three tools, particularly on the second one. Deductive synthesis relies on a general-purpose, automated theorem prover to generate programs. Theorem provers are subject to exponential growth in the search space required to find proofs, which is why deductive synthesis can be very inefficient. Domain theories must be tuned to avoid this exponential search. In previous attempts to address exponential search problems, system developers have generally tried to manually tune both a domain theory and the theorem prover's parameters, based on the observed behavior of the theorem prover over a series of test cases.[3,4]

Eliminating the exponential search

Meta-Amphion takes a different automatable approach to eliminating the exponential search. This enables the tuning component of Meta-Amphion to automatically analyze an easy-to-validate but inefficient deductive-synthesis system and transform it into an efficient DSSG.[5] We used deductive synthesis to construct Amphion/NAIF, but then used Meta-Amphion to automatically tune it to an efficient DSSG.

As our tools mature, we will demonstrate them in a number of space-enterprise domains, by using Meta-Amphion to develop software generators. Two new domains are in early development. One is the automatic generation of numerical simulations for computational fluid dynamics problems in the support of high-lift aircraft design. The second is data-understanding software based on probabilistic algorithms. In this case, a domain theory is given as a set of probability formulas. Specifications to the synthesis tool include error bounds. We will use validated transformations that exploit these error bounds to generate compact and efficient code with verified termination, computational complexity, and other attributes. We plan to demonstrate this data-understanding program-synthesis tool in the synthesis of software for onboard spacecraft data analysis and summarization coordinated with remote-sensing/prospecting robots.

References

1. M. Lowry et al., "Amphion: Automatic Programming for Scientific Subroutine Libraries," *Proc. Eighth Int'l Symp. Methodologies for Intelligent Systems, Lecture Notes in Computer Science*, Vol. 869, Springer-Verlag, New York, 1994, pp. 326–335.
2. Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Trans. Software Eng.*, Vol. 18, No. 8, Aug. 1992, pp. 674–704.
3. C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, San Deigo, 1973.
4. L. Wos et al., *Automated Reasoning: Introduction and Applications*, Prentice Hall, Upper Saddle River, N.J., 1984.
5. M.R. Lowry and J. Van Baalen, "META-AMPHION: Synthesis of Efficient Domain-Specific Synthesis Systems," *J. Automated Software Eng.*, No. 4, 1997.

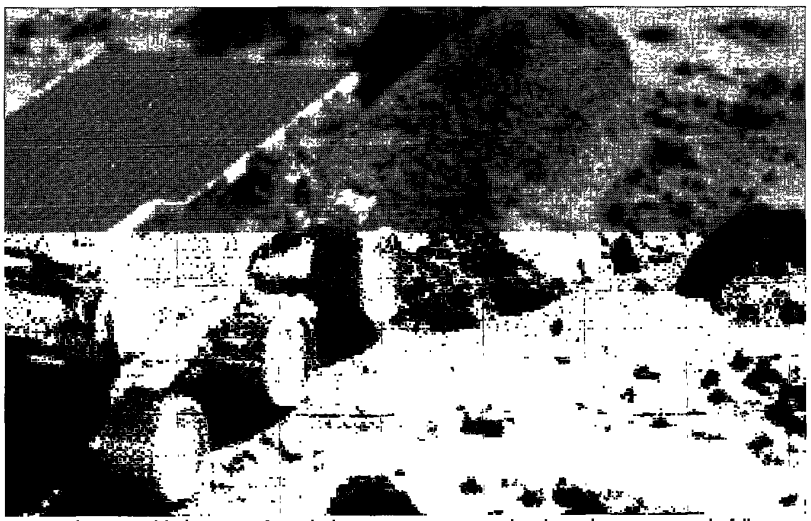


Figure 3. The Mars Pathfinder Rover software had a process-communication bug that, at least twice, caused a full system reset. Fortunately, this did not result in catastrophe, only in loss of contact with the rover for periods of up to 24 hours.

Graphical modeling for complex stochastic systems: the BUGS Project

David Spiegelhalter, MRC Biostatistics Unit, Cambridge, and Andrew Thomas, Imperial College, London

This is an exciting time to be a statistician. Not any statistician, of course—most are still doing the same dull analyses—but if you are a *Bayesian* statistician, you find yourself in the middle of an extraordinary coming-together of apparently disparate subjects. Researchers in artificial intelligence, engineering, genetics, image interpretation, epidemiology, and many other disciplines are discovering the common language of Bayesian graphical models. As a result, they have begun pooling ideas on models, computational algorithms, and implementation methods.

A crucial idea underlying this confluence is that arbitrarily complex systems can be represented as a graph of locally communicating components, making these systems ideally suited to software-generating tools that let users avoid hand-coding each new problem. One small contribution, the BUGS project shows how a generic and flexible tool can generate solutions to complex statistical-inference problems.

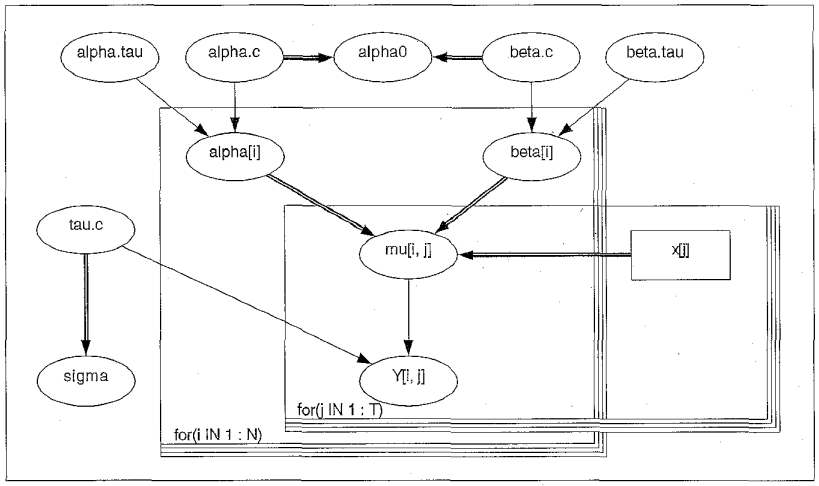


Figure 4. A Doodle for a model of growth patterns of rats. Each observation `Y[i,j]` represents the `j`th weighing of the `i`th rat, where the rectangular plates depict repetitive structure as `i` goes from 1 to `N`, and `j` from 1 to `T`. `Y[i,j]` depends stochastically on its mean `mu[i,j]` and its measurement precision `tau.c`, where the distributional shape for this dependence is selected from a pull-down menu (in particular, allowing it to be non-Gaussian). `mu[i,j]` is assumed to follow a linear-regression model depending on age `x[j]` and parameterized by an intercept `alpha[i]` and gradient `beta[i]`: the double arrows represent a deterministic relationship. Each rat's intercept and gradient are assumed to stochastically depend on some high-level parameters describing the whole population. Again, the precise stochastic relationship is specified from a menu, while the graph depicts the essential qualitative structure that drives the computational algorithm. Additional parameters of interest, such as the error standard-deviation `sigma`, can be added as deterministic nodes, allowing inferences to be made on arbitrary functions of parameters. The Doodle is created by a simple series of point-and-click operations, and then generates the code to make inferences about any unknown quantity in the model.

Bayesian networks

In the 1980s, Judea Pearl insisted that probability could handle uncertainty in complex intelligent systems, and that reasonable conditional-independence assumptions led to a graphical representation known as a Bayesian network.[1] (*Bayesian* refers to the use of a full probability model to describe the chance of any combination of events, using Bayes theorem to produce inferences on the basis of any observed data.) Not only was such a network intuitive and attractive, but it also formed the basis for efficient local computation algorithms.[2] Commercial packages rapidly appeared that exploited graphical user interfaces to construct systems of arbitrary structure and that then, at the compilation stage, automatically wrote the necessary code for propagating evidence. The Association for Uncertainty in Artificial Intelligence is a good starting point for exploring this area (*http://www.auai.org/*), while the Microsoft Research Decision Theory and Adaptive Systems group (*http://www.research.microsoft.com/research/dtg/*) is particularly active in many aspects of Bayesian-network research.

The relationship of Bayesian networks to other areas rapidly became apparent. Within genetics, pedigree analysis used similar representations and had developed similar computational algorithms; in engineering, filtering operations proved to have the same essential ingredients. Most important, because for computational reasons Bayesian networks become reexpressed as Markov random fields, connections were made to statistical physics, the spatial analysis of images, and geographical epidemiology. Exact propagation algorithms were not feasible in these areas, so a range of approximate methods were developed: in particular, Markov chain Monte Carlo (MCMC) methods entered mainstream (Bayesian) statistical analysis in the early 1990s. The stage was set for the conceptual integration alluded to above.[3] A very broad community is now working on probabilistic graphical models, which also now includes neural networks and coding problems.[4–6]

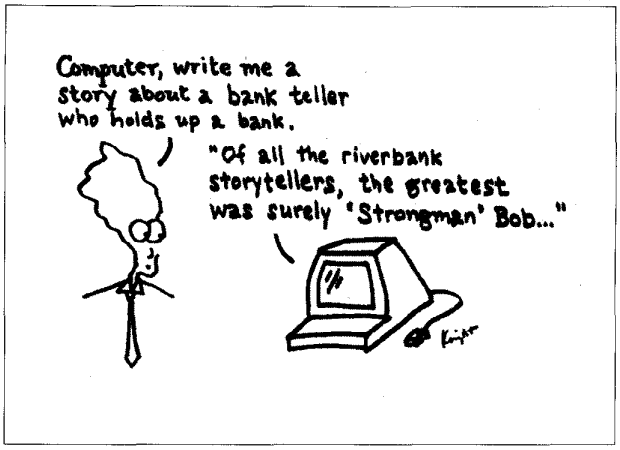
Code synthesis and BUGS

So what has this to do with code synthesis? The unifying theme behind all these research areas is the decomposition of complex problems into smaller components, which communicate locally to make global inferences. Hence, it is natural to have a graphical representation, a GUI for specifying structure, and a facility for automatically writing the code for carrying out the necessary inferences. In theory, users can assemble a basic set of components in whatever way they like that still provides a reasonably efficient and accurate analysis.

Our BUGS project has attempted a step toward this ultimate goal (*http://www.mrc-bsu.cam.ac.uk/bugs/*). BUGS stands for *B*ayesian inference *u*sing *G*ibbs *s*ampling, revealing that the basic computational algorithm was initially a specific MCMC simulation method that is ideally suited to graphical models, in which all unknown quantities are repeatedly simulated from their "full conditional" distributions. Newer versions of BUGS have extended the MCMC repertoire to include more general Metropolis sampling. The most recent version has a graphics editor (DoodleBUGS), which lets users construct directed acyclic graphs of arbitrary complexity through point-and-click operations and then use pull-down menus to select probability distributions on the links of that graph. Figure 4 shows an example of a Doodle.

How BUGS works

The BUGS system parses data structures underlying the graphical description of the



model in the DoodleBUGS editor into a tree form. It then traverses this tree and creates and wires together a series of objects to form a low-level representation of the Bayesian model (the graph layer). This is a compilation process: the Doodle is the "source code" and the wired objects the "executable code." The wired objects have methods and state variables that match the appropriate statistical concepts, and each different class of object is implemented as a separate component and is only loaded if required. BUGS stores the component names and the model specification language's syntax in a file, which it uses to parameterize the DoodleBUGS parser. A second layer of objects (the updater layer) carries out statistical inference using MCMC simulation, with each variant of the MCMC algorithm again implemented as a component. BUGS uses a distributed rule base to choose which variant of MCMC to use for a particular simulation. The updater layer uses methods implemented in the graph layer and copies simulated values back down to this layer. A third layer of objects watches the graph layer and produces summaries of the simulated values on user demand.

This software design is very flexible. Very general statistical models built out of standard building blocks can be analyzed, although we are now writing specialized components to handle special areas such as pharmacokinetics and spatial statistics. Once written, these components are on the same footing as those already existing, and new components can be added to the software at any time without any recompiling, indeed without any access to the source code. We used the Component Pascal language (conceptually very similar to Java) to write the software and used the BlackBox framework (*http://www.oberon.ch*) to implement a visual user interface.[7]

Who is BUGS for?

We designed BUGS for a particular class of intermediate-level users. It is not suitable for people who want to fit standard regression models to data and can get by with a standard $500 statistics package. Neither is it suitable for cutting-edge researchers in MCMC techniques. However, a surprisingly large group of researchers want to use realistically complex models for their particular problem, realize this will require nonstandard statistical methodology, yet are (very sensibly) loath to start writing their own code. BUGS lets them do fairly rapid prototyping and explore a range of model assumptions that would take a huge effort to program from scratch. The range of subject areas and associated models is unpredictable: users have been interested in predicting stock prices, modeling prawn abundance off the Great Barrier Reef, and identifying geographical clusters of leukemia.

BUGS is a natural extension of the successful expert system packages that implemented Bayesian networks, but the transition from exact to Monte Carlo methods greatly widens its potential applicability. Its development has been an iterative process, driven in equal measure by the programming philosophy and the statistical objectives. It has been very exciting to see how well these have integrated. There are certain restrictions to the distributions that can be used and to the sampling algorithms, but these are due to necessary prioritization rather than essential incapacity. We can implement additional distributions as new low-level components, while also introducing more flexible algorithms by making the higher level more "intelligent" at recognizing appropriate model structures.

Currently, the program's biostatistical background is evident and, although we are adapting it to specific areas (pharmacokinetics and geographical epidemiology), it will not be the most efficient program for the huge range of problems that might be considered under the broad heading of complex stochastic systems. However, as the neural-network, engineering, social science, and other communities increasingly adopt the Bayesian graphical modeling language, we feel confident that the basic philosophy of component-based programming is the correct one for developing flexible and incremental tools for analysis.

References

1. J. Pearl, *Probabilistic Inference in Intelligent Systems*, Morgan Kaufmann, San Francisco, 1988.
2. S.L. Lauritzen and D.J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems (with Discussion)," *J. Royal Statistics Soc. B*, Vol. 50, 1988, pp. 157–224.
3. W. Buntine, "Operations for Learning with Graphical Models," *J. AI Research*, Vol. 2, 1994, pp. 159–255.
4. M.I. Jordan, *Learning in Graphical Models*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.
5. R. Neal, *Bayesian Learning for Neural Networks*, Springer-Verlag, New York, 1996.
6. B.J. Frey, *Bayesian Networks for Pattern Classification, Data Compression, and Channel Coding*, MIT Press, Cambridge, Mass., 1998.
7. C. Szyperski, *Component Software*, Addison-Wesley, Harlow, UK, 1997.