IMPROVING RAM IN LARGE SOFTWARE SYSTEM DEVELOPMENT AND MAINTENANCE*

JOSEPH M. SCANDURAI and ALICE B. SCANDURA

Scandura Flexsoft Mtn? and University of Pennsylvania, USA

EXECUTIVE SUMMARY

Better development and maintenance methods and tools are needed to improve the reliability, availability and maintainability (RAM) of large software systems. Of particular importance are reducing costs and facilitating communication between developers and maintainers throughout the life cycle. Moreover, this needs to be done in an efficient manner — without the heavy overhead involved in the cumbersome exchange of information between programs not originally designed to work together. Under the best conditions, for example, the use of central repositories help provide needed coordination but this can significantly reduce efficiency.

Phase I SBIR research demonstrated the potential of dramatically improving RAM using a new cognitive methodology (and supporting tools). This cognitive methodology offers a simpler, more uniform

---

* This article is excerpted from the Final Report of a Phase II SBIR research contract for the United States Army (DAA21-90-C-0091) covering research from 1990 to 1995. It is composed of the Executive Summary of the research and an abridged transcript from a Working Group conference held at DISA in Washington on 11 and 12 July 1995 where research activities were reported and discussed. Since this conference, much of the research proposed has been implemented in Flexsys software and a prototype AutoBuilder project (e.g., see Scandura, J.M. A Cognitive approach to software engineering: next generation 00 paradigm. THIS JOURNAL, 1997). Interested readers are welcome to visit www.scandura.com for updated particulars.

* Corresponding author. scanduraRscandura.com.
approach, which combines reengineering with the waterfall, rapid prototyping and object oriented models of development. Cognitive technology is highly intuitive and involves modeling systems from the highest levels of abstraction. Data and processes are refined in parallel until contact is made with reusable resources. Evolving models are tested at successive levels of refinement, reducing by orders of magnitude the amount of testing required to verify correctness.

The overall goal of this Phase II research was to demonstrate applicability of the cognitive technology to large software systems on real projects. These projects included software (re)development, reengineering and conversion into C/C++ and Ada. Specific attention was given to several critical problems: (a) how to build software which is correct by design, (b) how to efficiently convert legacy code into C/C++ or Ada, (c) how to extract reusable components from legacy code and (d) how to get independently developed software to interoperate not only with each other but with logically consistent software models.

Toward these ends, a variety of Army (DoD) related software needs were identified, and a range of training materials on the cognitive technology were developed. These materials included slide presentations covering software development, maintenance, reengineering, conversion (e.g., to Ada) and the integration of independently developed software components and computerized tutors.

A variety of software development, maintenance, reengineering and conversion projects, which both SCANDURA and project leaders felt could benefit from the cognitive technology, were selected for inclusion in the Phase II research. Project personnel were trained in use of the cognitive technology and supporting tool sets. Consultation also was provided. Progress was documented with particular attention to similarities and differences between projects depending on how familiar project personnel were with the cognitive methodologies and supporting tool sets. Two extreme cases were tested: Several projects were conducted by personnel who had at most three days of training (and frequently no more than one day) on the methods and tools. Most importantly, three large projects were conducted with continuing and intensive involvement of SCANDURA personnel who were expert in both the methods and tools.

Due to the rapid downsizing in the U.S. military that was going on when the Phase II effort began, considerably more effort than expected was required to identify projects with the personnel stability needed for successful performance. Considerable attention, for example, was given
to the large SBIS project in which millions of lines of C code were to be rebuilt and/or converted into Ada. SCANDURA had strong support for this work from the Army at all levels. In preparation for training, SCANDURA used its "off the shelf" technology (without any customization) to automatically convert about 85% of the DAMIS code (well over 1 M LOC) into Ada. Nonetheless, personnel shortages and other non-technical factors caused cancellation, literally the night before scheduled training of the Army's SBIS staff was to begin.

Several projects involved conversions from older languages into C/ C++ and Ada. Redstone Arsenal received three days training on our technologies and was successful in converting a complex FORTRAN simulation program into a "good" Ada. Preliminary explorations involving C to Ada (based on less training) also were done by individuals at Ft. Monmouth and the APL Lab at Johns Hopkins. These were less successful due both to limited personnel background and training (for what is necessarily a complex job) and to the low priority given the projects. It is not possible, for example, even with the best automated tools, for a person to convert a complex language, like C, into Ada without good familiarity with both languages.

In one major project, SCANDURA personnel used these same conversion technologies to convert about 600K lines of highly optimized satellite imaging software in VAX Pascal (from MacDonald Dettwiler Associates — MDA) into C/C++ on UNIX. Although cognitive methodology was highly relevant in this case, our Flexsys automation tools had to first be customized to support VAX Pascal. It took about four weeks to support reverse engineering and another six to support conversion to C/C++. Overall, the project was of extremely high priority and had an extremely aggressive schedule of six months — later reduced to five. After customization, actual reverse engineering was done automatically on a low-end DEC Alpha in under three hours. The actual conversion took about five and a half hours, including a substantial amount of time for communication between SCANDURA and MDA, the entire project was completed five weeks ahead of schedule.

A second major reengineering project also was undertaken at Ft. Monmouth. This project was very intensive but of short duration. The goal was to analyze over 400 K LOC poorly organized C code to determine the feasibility and desirability of extracting portions of the code ("golden nuggets") for reuse and incorporation into the Army Common Operating Environment (ACOE). In less than three weeks, we: (a) reverse
engineered the code, (b) identified and classified all global identifiers, (c) constructed call and unit (derived requirement) hierarchies (showing all relationships among various modules and compilation units) in the system, (d) obtained various metrics and (e) developed a reuse strategy. This strategy includes extracting potentially reusable components and semantically wrapping those components along with ACOE components so they all interoperate — not only with each other but with any other semantically wrapped components.

The final major project involved refinement of the cognitive development methodology and its application in software re-engineering. Development-wise, cognitive technology is a highly intuitive methodology that includes abstract "real world" operations as well as objects. By analogy, notice that expressing any complete thought as a sentence requires both a subject [object] and a verb [action]. (In Appendix G [Cognitive Analysis, Design and Programming], cognitive technology is contrasted with the object oriented paradigm in which design is based solely on objects.) Cognitive technology also provides an explicit method for developing software systems that are correct by design. This is an important goal shared with "clean room engineering," a method pioneered by IBM and supported by ARPA as part of the STARS program. Cognitive technology, however, simplifies the "correctness" process, both conceptually and as a result of automation. Whereas the latter is based on a hardware metaphor (e.g., state-transition diagrams), cognitive technology has an intuitive cognitive foundation more suitable for the human processes required in software design.

Cognitive methodology and supporting reengineering technology was used to completely redesign "top level" code in the PRODOC software engineering system. In the process, a large number of C components were extracted (from a system of 500K LOC), semantically wrapped and made interoperable. None of this work directly affected functionality of the system. The system did exactly what it had done before. From a maintenance (and enhancement potential) standpoint, however, the differences were dramatic.

Before redesign the highest level compilation unit in the system consisted of 72 FLOWform modules. After the redesign, all of the design

\(^{1}\)PRODOC software has undergone major revision and extension over the past three years. Special attention has been given to meta-components which greatly facilitate supporting application-specific conversions as well as new languages.

\(^{2}\)FLOWforms have been extended and are now called Flexforms.
logic is contained in only five FLOWforms. These FLOWforms call about 65 semantically wrapped FLOWforms comprising the bulk of the lower level code. The latter are referenced resources but otherwise may be viewed as "black boxes". As a result of the top level redesign, the five design FLOWforms are the only ones which need to be changed. Moreover, all of the semantically wrapped components are interoperable, not only with each other but with every other semantically wrapped component. As a result, we have observed a dramatic reduction in the effort required to maintain, enhance and otherwise modify overall system logic conservatively, by a factor of 50 or more. In this regard, length alone was reduced by a factor of 12 or more. (In terms of maintainability, one can make a case for complexity increasing by the square of the length — i.e., a factor of 144.)

Results of the Phase II research clearly demonstrate the value of cognitive technology in improving software reliability, availability and maintainability. Adjustments and refinements made to the methodology and supporting tools, both during and after the projects, make the technology even more appealing. Important advances have been made in: (a) methodology, (b) supporting technology and (c) the conceptualization of software design, development and maintenance processes. Overall, the potential benefits are enormous. The value of cognitive, of course, will depend directly on the extent to which these advances are implemented.

Consequently, dissemination was an integral part of the Phase II research. Preliminary results were reported at the Software Technology Conference in Salt Lake City on 12 April 1995. Full results were presented to a working group held at DISA on 11-12 July 1995. This working group was attended by software experts in the Army, Marines, Air Force, Navy, the Pentagon, DISA, GAO, ARPA, Unisys, AT&T, several universities and other organizations. An abridged transcript follows this section.

Judging from comments made throughout the working group, most were intrigued with the possibilities cognitive technology offers. Many, including technical experts representing senior Army personnel, expressed a strong interest in pursuing opportunities. Given this strong interest, we currently are pursuing various opportunities both in the military and the commercial world. Among possibilities discussed at the working group are the following:

(a) The efficacy and efficiency of cognitive technology in relation to clean room engineering should be determined. Availability of
automated processes for guaranteeing correctness are especially appealing. The advantages of developing software which is correct by design, minimizing the need for implementation testing, is hard to overestimate. (Oddly enough, another group at Picatinny Arsenal was sponsoring a field test of traditional clean room engineering during the Phase II research. Unfortunately, neither we nor our technical sponsor at Picatinny, Dr. Mort Hyman, were able to solicit clean room involvement.) In view of the unusually positive results obtained in this research, it would seem that such an effort should be given very high priority.

(b) High priority in the Army also should be given to taking advantage of the semantic extraction and wrapping technologies successfully used in this research. The advantages of having Army COE components interoperate with each other as well as with components extracted from legacy code is too important to ignore.

(c) In addition, priority should be given to exploring ways in which cognitive technology could add to the I-CASE effort. (Indeed, SCAN DURA was part of and contributed to the original winning I-CASE effort.) In this regard, Flexsys’ customizable and automated re-engineering and conversion technologies would seem a natural fit, particularly in view of the desire to move mission critical software into Ada. These technologies have proven to be a very cost effective way to achieve high quality results.

TRANSCRIPT: WORKING GROUP ON "IMPROVING RELIABILITY, AVAILABILITY AND MAINTAINABILITY OF LARGE SOFTWARE SYSTEMS"
(HELD 11-12 JULY 1995 AT DISA HEADQUARTERS)

AM Session

Arrival


Participants also received an overview of the Flexsys Re-engineering, Conversion and Integration Factories.

Introductions

Col. Gross opened the Working Group Session for Ms. Diann McCoy, welcomed the group and introduced Dr. Mort Hyman (MH) of Picatinny Arsenal, Technical Director for the SBIR Phase I and II Projects.

MH presented his views of the project and management issues involved in introducing new technologies into the existing government information systems structure. Parts of his remarks were from a paper he prepared for the STC’95 Panel.

MH introduced Dr. Alice B. Scandura (ABS), the Principal Investigator on the SBIR on Improving Reliability, Availability and Maintainability of Large Software Systems.

ABS introduced Bruce Lewis (BL) or Redstone Arsenal and Craig Chris (CC) of Ft. Monmouth who directed SBIR related projects and presented DoD Needs and SBIR Goals.

Joseph M. Scandura (JMS) indicated that his background is in the area of cognition and SCANDURA has come at the issues and problems associated with software from this perspective. The morning session provided a foundation for the remainder of the workshop. JMS described a unified approach based on cognitive principles. Solid methodologies and technologies have been developed and proven in practice. Enhancements are continuing both as add-ons and as a result of reengineering. Several years ago, for example, Pascal underpinnings were converted largely automatically into the more widely used and better supported C language. Major parts of the system have recently been componentized and SCANDURA is in the process of adding a GUI. JMS emphasized that the most important point is that the components are independent and interoperable. Maintenance is greatly simplified and is an evolutionary process. JMS also promised to demonstrate recent work illustrating how existing semantic facilities could be used to ensure that software developed using the cognitive technology could be guaranteed to be correct by design.
JMS presented a slide (Fig. 1) summarizing the cognitive methodology and indicated that it applies equally well whether one is building a new system or renewing an existing one.

**STANDARD VIEW**

![Standard View Diagram]

**NEXT GENERATION**

![Next Generation Diagram]
His next slide (Fig. 2) contrasted the smooth transitions characteristic of the cognitive approach to software development and maintenance with the standard staged view where planning/analysis/design/implementation are viewed as separate processes.

Re-engineering from a Cognitive Perspective -- with optional Conversion

1. "Spaghetti" Code: Most software systems come to look like this as a result of on-going maintenance. Notice that most of the individual modules are fine, and can be reused. The whole system, however, is rather disorganized. (Most new design ideas also start like this.)

2A. Re-engineer Code: Reverse engineer code from the existing system into modular FLOWforms. You easily understand module structure and conveniently make needed repairs. (This step is unnecessary in developing new systems.)

29. Redesign System: Redesign the desired system at a high level, using PRODOC's universal 1-11.0 language. Test your design for logical errors, simulating process and data. At this point you should have a hierarchically structured system. (Step 2B is optional where the original structure is acceptable.)

3. Rouse Modules: Map reusable modules from your old system into the new high level system design. Finally, use PRODOC to design and develop missing low level routines. Experience shows that from 50% to as much as 95% of existing modules are reusable.

4. Conversion Options: Convert FLOWforms to a more modern language using PRODOC VANtage Translator and Semantic Postprocessor options.

FIGURE 3

The next slide (Fig. 3) showed reengineering from a cognitive perspective. Starting with spaghetti code, he showed how redesign can be combined with reengineering to extract "golden nuggets" from legacy code for possible reuse and/or conversion into Ada.
Next slide (Fig. 4) presented an overview of the Flexsys semantic integration factory. The starting point is legacy code and other software components (i.e., C++ libraries). JMS showed how "golden nuggets" might be extracted from legacy code, and made to work together. One way to accomplish the latter is to be a Microsoft — define a standard and make others follow. Standards are critical, but most standards tend to be designed by committees, with many compromises. JMS argued that we should ask what should an interface look like, and why it should look that way. Following the slide, JMS indicated that SCANDURA had automated technologies for wrapping certain kinds of software components, and had developed a general purpose engine to work with wrapped components. Together, the engine and wrapped components form a domain specific development system (DSDS), optimized for a particular domain.

DSDS's, then, can be used to efficiently create and maintain new applications in that domain. JMS suggested this schematic as a resource to guide subsequent discussions.
The next slide (Fig. 5) summarized the issues to be discussed at the workshop and (at the bottom) the topic for the first session. JMS then discussed HANDOUT 5 (a schematic) summarizing components currently available in Flexsys Factories. He pointed out that the simulator can take any system made up of virtual (i.e., abstract and/or actual) wrapped components and simulate it. Moreover, the simulator component can execute itself. JMS indicated that SCANDURA has only begun to take advantage of the possibilities but this new technology makes it possible for the first time to simulate higher order thinking, and automated software construction.

**BL:** Will we discuss more about component wrapping and decisions about what goes into the wrapper?
JMS: We plan to demonstrate that tomorrow morning, and hope to get to some detailed discussion of that topic.

Returning to design, JMS pointed out that the traditional design, implement, test paradigm can be very inefficient. Even in a small system containing 100 binary decisions, for example, one would have to test \(2^{100}\) paths (big bang example). It is totally impossible to test each path of such a large system. JMS proposed an alternative. Start testing from the top, and continue testing as components are refined. Testing in the latter case only goes up additively with the complexity of the system rather than exponentially -- a dramatic difference.

JMS emphasized that the technology do this — to create designs and debug those designs as they evolve — exists now. During implementation, one needs only to ask whether each component does what it's advertised to do. If it does, the system will work without further testing.

CA: You seem to be implying that how much testing needs done depends on what stage in development you're doing the testing.

JMS: It's more a matter of testing systems as you go. I believe this will become clear during the demonstration, but the basic idea is you test your design as you make that design more and more precise.

"Clean room engineering" coming from STARS and IBM is attempting to do something analogous. But they don't have a technology to support the process. In "clean room", the goal also is to make programs correct by design, but it's done sitting around a table, by analyzing the design process. Clean room also has some scalability limitations. (Discussion returned later to the question of why these limitations exist.)

**PROCESS REFINEMENT**

Abstract operation can be refined into:

- a sequence of operations, a selection or an iteration.

- **sequence**: intermediate objects introduced; input/output unchanged

- **selection**: "extra domain" condition partitions domain into (sub)objects; selected operations produce different outputs

- **iteration**: variation on binary selection — LOOP
  body operation modifies outputs systematically insuring transition between (sub)operations (one exits loop)

**FIGURE 6**
Any system you can imagine has a DOMAIN, something going in, a RANGE, something coming out, and an OPERATION mapping the first into the second. If we’re talking about software engineering systems, for example, we would have software engineering events coming in, software engineering results coming out, and software engineering processes making that happen. None of this says very much, but it focuses attention on what’s important. With further levels of refinement, things get more interesting. The important point at each stage of refinement is to ensure that the design is logically consistent (Fig. 6).

SYNCHRONIZING INPUT/OUTPUT OBJECTS WITH PROCESS REFINEMENT

(Abstract input/output objects must also be refined)

- partitioning of input/output objects introduces new (sub)objects
- abstract input is a function of input (sub)objects -- classic case of "evaluation"
- output (sub)objects make abstract output more concrete -- add detail to output abstraction by "inheritance"

FIGURE 7

The process is not like structured analysis, which involves refining a process, and dealing with data later. Or, as in information engineering, breaking down data and filling in processes later. To ensure testability and consistency at each level of refinement, data and process must be refined in parallel (Fig. 7).

A discussion of O0 oriented design followed. JMS pointed out that while the real world is made up of objects, it also involves operations and that O0 design relegates operations to a subordinate position. He pointed out that requiring both objects and actions on objects is analogous to written language. One cannot write a sentence without both a subject and a verb. (Operations within objects are modifiers, much like adjectives.)

JMS presented slides (Figs. 8 and 9) illustrating the process with a simple example involving a system for cleaning a room.
The common starting point for modeling any system.

A top level model of "clean room".

Second level of refinement of the clean system.

BL: The slide says "clean room". Is this representing how you represent the clean room process?

JMS: Actually, this is just an example that refers to a design for cleaning a room. The double meaning is just a historical accident.

BL: You said that some clean room users have reported a limit of 10—15 K LOC. Does this mean you are tying to find a way to go beyond this?
JMS: Yes. Wayne Scheerer reported that limit in a talk about a year ago.

By way of contrast, the cognitive approach appears to scale indefinitely. Returning to the problem at hand, notice that the domain is the room. Range is also the room. In general, domain and range may be the same. The operation "clean" operates on "room" and changes its value - e.g., from "unpresentable" to "presentable". Clearly, this program is internally consistent. But to actually clean the room, we obviously would need more detail.

JL: You could put "house" in the domain and still perform the operation clean (room).

JMS: Possibly, but you would first have to redefine the DOMAIN.

Suppose you presented a child with this information. That is, just tell the child to clean the room. If the child understands, the room might be clean. But, then, it might not be clean, in which case you would need to be more specific. You might say to the child: What I mean is 'Make your bed then vacuum the carpet.'

In Fig. 9, notice that room is refined into bed and carpet. From the standpoint of this program, only the carpet and bed are important. There may be other features of the room but only carpet and bed will be considered because they are the only features specified.

Aside: Notice the @ sign in the FLOWform (FF). This is the FF way of representing fanning in. Nodes with @ are "clones."

The demonstration next moved to an actual demonstration of the clean room example on an LCD panel display.

JMS: Here is what you saw in Fig. 8 "clean (room)". "Room" is the domain, "room" is the range. What we're going to do is execute this design - simulate it as if it were executing - that is test the design. Step 1 is clean your room. It says your room is currently "dirty." It says perform operation "clean" and update the value. Let's say it is clean. We can look at the results. The room is now clean. Now let's refine "clean (room)" into "make (bed)" and "vacuum (carpet)." Initially, bed is "unmade," and carpet is "dirty." Now we're going to execute it. First step is: Make your bed. Result is going to be? ("made") Next step is vacuum. Carpet is dirty. Vacuum it, result is. . . "clean." Let's look at the results again - the bed is made and the carpet is clean.

What happens in a loop refinement? The essence of a loop is the following. You want to perform the same operation on an indefinite number of data elements. To iterate you need a pointer or alias.
The demonstration continued with successive refinements. In each case, results were shown both during and after the simulation. The final refinement involved a combination of steps, some of which executed automatically and some requiring human input. In all cases the results were interoperable making it possible to empirically verify the design.

EW: I'm still not sure why domain cannot be house. (Fig. 10)
JMS: From a common sense standpoint, from what you already know and from what's up there, that would be a valid statement. But the program does not know that.
EW: How? Does it know that carpet is part of room?
JMS: Because it was defined explicitly.
MH: You could embed this program in a superprogram the object of which would be to clean a house. Then, you could have inheritance and all that jazz. What JMS is describing is a hierarchy of
operations. You could start higher, at a house, but he's developing a program to clean a room.

EW: But it already has a presupposition that it is a bedroom.

JMS: You could infer that, but it is important to separate what you know from what you specify explicitly. Bugs come from inferring meanings that aren't there.

BL: What is the impact it you add objects that aren't in that room? What parameters need to be changed or added?

JMS: A good programmer for example might say, "I don't need that." It's redundant. Worse, the design could be inconsistent. The more complex the system, the more likely those problems will come up during debugging. In a trivial system like this, you could build the program without going through a detailed cognitive analysis. But, as systems get more complex, the more efficient this process becomes because of the sheer amount of testing required (if one waits until after implementation).

CA: I appreciate the example you gave from the standpoint of programming and also from the standpoint of human relationships. It's good as a real life situation and I appreciate that you want to train the child to clean the room. You can write a program that way too. You start with a subtotal of the whole program, then using $00$ you want to define a superobject of the room, with its own characteristics, which the room can inherit.

JMS: From a cognitive perspective you would start with the super object and a super operation acting on that super object. Refinement is always relative to some starting point. A careful child might not need more explanation. A less careful child might need more explanation. (For further discussion of $00$ in this content, see Scandura, J.M. "Cognitive approach to software engineering: Generalization of the $00$ paradigm," TIHS JOURNAL, 1997.)

HP: On the other hand, in life, the room may never be clean. In programming, no matter how much you refine, you may never get what you want.

JMS: This comes about when a component is missing or defective. The overall design is only as strong as its weakest link (i.e., component). So, if you use buggy components you'll have trouble. It is important to test components thoroughly before they are used.
SWP: What if you were building a piece of hardware with a specific amount of space with no room to grow, and you're trying to put a program on that hardware? If you were defining your domain and range using this, you could rapidly exceed your hardware resources.

JMS: I would agree that any piece of hardware will have limitations. However, the limitations don't have anything to do with the methodology you use.

SWP: Wouldn't that lead you to think you're a little more safe than you normally would? You refine range and keep refining it. Then, you ask it to do more and more. It is growing. What kind of a monitoring connector do you have with that?

JMS: Think of it as writing documentation for the system. You first start out saying "clean your room." Then you refine it. You are effectively writing specs for the system.

MH: I think she asks, this is a way of programming, and there are other ways of programming. If you use the cognitive way, will you run into memory problems?

JMS: There is nothing inherent in this method that takes more memory than any other method. The particular design you choose always reflects the designer's intuition about how to attack a problem. That's where cognition comes in.

BL: Often you work with limitations and systems imposed by others you are working with. How compatible would this technology be with this problem? How do you deal with distributed systems?

JMS: (Fig. 4, Integration Factory) Here is where the components we're talking about come from. Ideally, they would work on whatever machine they are (stored) on.

MH: Suppose you have two kids. One is doing the floor and one is doing the walls. You have two computers and they may be operative simultaneously. That's an implementation detail. We're talking about programming methodology, how you put the program together logically.

JMS: The methodology explicitly supports parallel refinements. In fact, clean room is refined into parallel operations of make (bed) and vacuum (room). Incidentally, how many of you are familiar with 00 (most hands)? How may believe that objects are the solution to everything? (no hands)

JMS went through the slide (Fig. 11) tracing the history of programming paradigms. In procedural programming, functions can use any global
variable in the entire system. As a result, as systems get big, we get more and more problems, more and more interaction.

The idea behind modular programming is that we can minimize problems if we modularize. One can still use global variables, but (ideally) only within given module and similarly for this module.

That’s still not good enough. Some said we ought to group things together on some conceptual basis. What is that conceptual basis? How about types? We don’t put things together arbitrarily, we group them (according to the types used).
OO folks came along and said we can do better. We don't just want to
group resources; we want to be able to build on (inherit) from what we
have. Think about what inheritance really means. You can have a variable
of type B, but a function fnA1 that come from module A. A is like a car and
B like the sportscar subclass. If fnA1 is basically drive car, you can also
drive sportscar (B). That's basically what we mean by inheritance. This
has certain advantages. Most people who've gotten into OO recognize
there is a big difference in the way you think about programming objects
with messages between them. The bigger the system gets, the more com-
plicated things become. Putting software in classes does not solve the
reusability problem, interface issues, etc.

The GOO system schematic (bottom SLIDE 18) captures only some of
the difference (discussed above) with OO. It does not, for example, show a
representation of successive refinement (discussed above). The GOO
schematic indicates that A and B are objects (overloaded) and fnA1 and
fnB1 are abstract operations defined on those objects. In OO, you'll have
a car object with functions in it, all grouped together in the car object. But
we have all kinds of things that can be driven — cars, trains, planes, horses.
In OO, you'd put the drive train operation in the train object, and drive
plane in the plane object, and drive horse in the horse object. And C++
would allow for that. You'd overload the function. Each particular
"drive" operates on objects according to type. In GOO, "drive" is an
abstract operation on distinct objects. In effect, you can have abstract
operations just as well as abstract objects.

Cognitive methodology deals in parallel with both abstract operations
and abstract data. If you just take operations and make them primary,
that's structured analysis. But that it isn't GOO. In GOO, Types dis-
appear. Indeed their disappearance is a solution to one of the key prob-
lems. GOO is typeless because types are reflected in the real world
structure being modeled. You can even have operations on other
operations.

You can also have data execution — dynamic definition of variable.
Object B is defined as a function of A. One object is defined dynamically
from the attributes of the other.

MH: You could say that current OO subordinates operations to data
structures. What JMS wants to do is have functions and data on the
same level, like nouns and verbs. You have a piece of data. You
perform an operation on it, and out comes another piece of data.
The idea is not to subordinate operations to data or vice versa, but treat them on the same level.

JMS: That's a good way to summarize it. Let's turn now to "clean room engineering" as defined by IBM and STARS. Then, I'd like to show you an example I designed recently to illustrate automatic program correction.

JMS summarized key ideas underlying clean room engineering. It shares the goal of creating programs that are correct by design but uses a hardware (state transition) metaphor rather than a cognitive metaphor. As you create a design, it would be nice to press a button - and we're not quite there yet -- to find out if you did it right or made a mistake, and ideally have the system tell you what to do to fix any problems. Using the clean room example, JMS showed how Flexsys could be used to check the "clean (room)" refinement into "vacuum" and "make bed." Though the refinement looks correct, in fact, it is not a sequence. It looks like a sequence and you probably wouldn't find an error unless you are in a parallel processing environment. A consistency check revealed that the refinement was, in fact, inconsistent.

Changing the refinement to a parallel refinement was sufficient in this case. Other checks revealed the use of inappropriate variables in a loop refinement.

CA: It also seems that the error message could have been a little better.

JMS: Absolutely. This is what we put together in just one day. It's definitely not a finished program.

FK: Do you distinguish between operations that can take place in parallel and those which must be performed in parallel?

JMS: At this point, we aren't making such a distinction. What we are not doing is dealing with communicating parallel processes.

CA: What then is an operation that must take place in parallel?

JMS: This is something the designer must decide.

MH: You could also have two independent processes that needed to come up with a result and they don't take the same time. They don't have to start the same time, and don't have to finish the same time, but when completed you've done what you set out to do.

BL: At lower levels, do you implement some kind of cpm module to implement parallelism?

JMS: Not being a cpm expert, I can't respond specifically to that. It's important to understand that the cognitive methodology
and supporting Flexsys tools are independent of how things are implemented.

CA: What is that little dotted line? (Fig. 10)

JMS: In displaying a FLOWform, you can collapse or expand elements at will. You can show higher levels. You can also show the higher levels and terminal levels in the same FLOWform directly in context. The dotted lines distinguish higher levels of abstraction.

The FLOWform is a unique representation in several respects. You can find a FLOWform representation that is equivalent to any number of other representations. FLOWforms also support indefinite refinement — all in context. You cannot do that with a bubble chart for example. Any time you open a bubble you open a window and after three or four levels it is easy to get lost. FLOWforms are contextual and allow one to go from high level design down to actual code.

CA: You showed how you can check the sequence of operators to see if it needs to be sequential. Do you have a similar device to check the data?

In this example, the carpet can be messy, dirty & messy, dirty, etc.

JMS: In principle, it can be dealt with. You need to define your domain of application: What is legal, etc., but this is not yet implemented. (Further discussion ensued about loop refinements and the need for aliases (indirect references).)

Vacuum relates to carpet, but when you refine into a loop, you're talking about a partitioning of carpet into a set of equivalence classes which make up the carpet: seg1, seg2, seg3. Moreover, the body of that has to address each of those segments. But how? You can't say do seg1, seg2, etc. but must do it via an indirect reference. How? We invent something, an alias called "current_segment", which stands for a different segment each time through the loop.

One thing that has to be asked is whether the loop terminates. Put another way, does it guarantee a transition between elements into which the original data element was partitioned. one of which gets you out (of the loop)? That must be guaranteed.

CA: There is another thing that must be guaranteed. when you divide the carpet among the segments, the segments equal the carpet.

JMS: Yes, that is implicit in the definition of a partition: mutually exclusive and exhaustive.

BL: You have the function clean room operating on a room with a bed and carpet. It can take any kind of consistently stated segment.
Is that what you were talking about as far as having a free floating function that different objects can use? Or, do you refine the functions first and the objects are really data relationships?

JMS: GOO refinements refer to both process and data. When you refine a process at the same time, you must perform parallel data refinement. They go together and you can't do one without the other. For example, if you had something referring to segments before you refined carpet into segments, it would be meaningless.

BL: When you actual program it, does clean (room) become the free floating function, or is it encapsulated in room, or does it make a difference?

JMS: Think of it this way. Clean is the operation. Room is the abstract object on which clean is operating. After refinement, you are operating on a child element of carpet with a different operation. C++ and Ada 95 allow objects with inheritance and also overloaded functions. From a conceptual standpoint, it makes sense to pull out those overloaded functions and coordinate their refinement with refinement of the data. Doing that solves the problem in 00 of not knowing what functions to keep separate and what to put in objects. Often, the urge is to put all sorts of things into objects which is not usually a good idea. This is not a good idea for program understanding or debugging.

MH: I can think of a way to choose between sequential/parallel. You're trading time for memory. If you have time, do it in sequence. If you have memory, do it in parallel.

3MS: Seems reasonable. Let me emphasize that cognitive technology does not require a new language. Everything we've talked about can be implemented in C, C++ or Ada 95.

SAMPLE COMPUTER SCREENS

Discussion turned next to a demonstration of clean room engineering. Specifically, GOO refinements as FLOWforms were automatically converted into the state-transition diagrams used in clean room. By way of contrast, clean room practitioners create state transition diagrams by hand, refine them and then sit around a table to decide if the system is correct.

Box structures or state transition diagrams have a history (Fig. 12). Perhaps the biggest difference (other than use of a different diagram) is
SOFTWARE ENGINEERING AND RE-ENGINEERING: THE NEXT GENERATION

BOX-STRUCTURES: STARS program puts the focus on behavior, requiring very precise specification of S's, R's and functions connecting them. System design by box-structure expansion provides a provably complete and systematic top-down approach to analysis, design and implementation. According to Mills et al (1987) "complete description of ... data operations leads to a data jungle even more tangled and arcane than the control flow jungle of software."

Focus initially is on defining the problem in terms of S's and R's. Tendency is to seek full specification of S's and R's at a relatively low level before attempting to define the first black box. No commercially available tools support Mill's specific notation, although FLOWforms used in Scandura's re/ NuSys Workbench approach represent the equivalent.

COGNITIVE APPROACH: Based on FLOWforms and "structural analysis", which derives originally from Scandura's Set-Function Language (e.g., 1969) and Structural Learning Theory (e.g., 1971, 1973, 1977). Emphasis here is on abstract data (i.e., S's and R's) as well as abstract processes (i.e., functions). Like system design by box-structure, cognitive approach is focuses on inputs, outputs and functions connecting them. It is different in that step-wise functional expansion may begin with more abstract S's and R's.

FIGURE 12

they must specify ahead of time all of the stimuli and responses in the same level of detail as will be used in the final system. If you have a big system, this will take a lot of work. In GOO you specify the in/outs only to the level required by the current refinement. One never includes more detail at any one level than necessary.

In clean room, notice that there are three steps. The first two correspond to the original GOO representation. The third level in clean room corresponds to the refinement. Further details were provided during the demonstration:

Stimulus histories and specific responses are at the black box level. Particular stimuli and states (the history up to the time just before the stimulus) are shown at the state box level and refinement is shown at the clear box level.
JMS: Notice you can't really fit everything in these boxes. This is one of the FLOWform advantages I was alluding to before.

CA: In the case before where you had a sequence and decided it didn't need to be sequential, it could be parallel. Sometimes it doesn't pay to bother setting up parallel processes (Fig. 12).

JMS: Sure. What we're talking about is something to guarantee correctness. You could get away with having this be a sequence as well as being a parallel process. It all depends on what the designer wants to do.

MH: Sometimes it's advantageous to do it one way. Sometimes it's advantageous to do it another way.

JMS: Distinguishing parallel processes, even where they can be implemented in sequential fashion, does facilitate comprehension just as in data flow diagrams.

MW: When you talk about proving correctness you have said nothing about performance.

JMS: That's true but there's no reason one couldn't introduce constraints such as time into the process. In addition to specifying the process, make an estimate of how long it will take to do execute. Then introduce goals or requirements indicating how much time may be allotted. This is more complicated and we haven't done it, but the necessary mechanisms and technologies are available.

In this regard, one thing that has come out of this SBIR is the importance of customizability. It is impossible to anticipate solutions for every problem people come up with. What are needed are generic capabilities that are easy to use and understand. This involves an emphasis on semantics — which is what cognition is all about.

MH: I hear there is no universal Scandura machine, like the universal machine which can do anything. Rather, you use general machinery to deal with things such as time constraints.

BL: Eventually, you want to take this and convert it into C++ code?

JMS: You don't have to. In a way, that is the beauty of the approach. We don't care where the components come from. Even if they were FORTRAN, you could put a semantic wrapper around it, and use the results. You don't care what they're implemented in. Or you could use the conversion machinery and actually convert the legacy code into Ada.

BL: So you have a choice of language implementation.
JMS: You can get the code from anywhere. They'd better be worth the effort of course. If not, even in a logically correct system, you'll have problems. If you buy components, you want to make sure they're good. If the components are to come from legacy code, you need to make sure they are extracted, and componentized and wrapped correctly.

MH: If you use components that are proven but from "foreign" languages, how do you do the wrapping? If sounds like you’re going to set up emulation processes so you can use things quickly in different languages in the same program. Is the program you’re putting together going to have to do some emulation depending on the language in which one of those components is written? The other idea of putting all the components into the same language might be desirable if you're worried about speed or memory for the final program.

SWP: Would you also have to make sure the hardware it was originally compiled on is equivalent to the hardware you'll eventually compile on. All compilers are not equivalent. If you had code in assembler for a Motorola chip and you put a C wrapper on it and tried to run it on a pentium chip it wouldn’t work.

JMS: Of course there certainly are architectural constraints you need to face. Similar issues need to be considered in distributed systems, for example. And these are not trivial problems, but we think they have methodological solutions. Our approach always has been first and foremost to get a sound generic, conceptual solution first. The big problems we've seen over and over are when you try to deal with a technical problem and lose the "big picture". We try to solve the big picture first and add technical details as needed to fill in the picture. This is not to say, of course, that we have filled in all the holes.

PM Session

Figures 13 and 14 provide a common terminology for reverse engineering, reengineering, conversion (to Ada), renewal and conversion and renewal. How do you define a repository of a system or a system?

Figure 15 shows the structure of an arbitrary software system organized into modules (e.g., functions), relationships between modules (e.g., call hierarchies) and relationships between relationships (e.g., file/compilation unit relationships).
MODERNIZATION

Reverse Engineering

Re-Engineering

Conversions

Renewal

Conversion & Renewal

FIGURE 13

FIGURE 14
JMS: Ideally, a repository containing information about a system should have the same structure as the system. The modular FLOWform repository has this characteristic. Changes to the system are changes to the repository and vice versa. This approach is much more efficient in maintaining code than the traditional central repository approach where fundamentally different kinds of files (data bases and text source code) must be kept in sync — usually as a result of manual effort.

MW: There is a difference between the way you design software and the way you design something that involves hardware, software and people. There is much that you do in hardware design that you don’t do in software design.

JMS: What is the essence of that difference?

MW: Manufacturing.

JMS: What kinds of implications does that have in your design?

MH: It restricts the design because materials have to be available. Software is logical and the more you conceive, the more you can do.
With the hardware you have to wait until someone comes up with a better metal etc.

JMS: Yes. In hardware you're restricted to available components.

MW: You may or may not be. You can use components to fabricate your own components. The main difference is who you hire.

JMS: The important point is that the basic principles of design are the same. The only difference is in the nature of the components.

BL: If I have a scheme in engineering it can be easily tested by others. The basic unit is a module and there are different ways of describing a module. Some of the rules of engineering really don't apply. The question is which do you apply and which do you not apply?

JMS: You can build any software components you want and are willing to pay for. In cognitive design, you have the freedom to describe real world systems as you wish. You can invent whatever virtual components you need as you need them. So you have much greater flexibility.

JMS pointed out that from a cognitive perspective the only difference between a system and a family of systems is in the level of abstraction. At a high level of abstraction, you have an incompletely specified system. You've defined a family of systems (Fig. 15).

Question: Has anyone ever used cognitive methodology in developing a large system?

JMS: In part, yes but not taking full advantage of what the cognitive approach offers. But we have used the technology in reengineering. The Flexsys system, for example, was organized in the usual way. The top level consisted of about 120 modules and 100K lines of code. As a result of a top down cognitive analysis, using GOO methodology, we ended up with only 5 high level modules that needed to be maintained. As a result, maintainability has gone up dramatically — we think conservatively by a factor of 50 or more. Originally, for example, it was difficult to simply add a new menu.

Now, this can be done by adding a single statement.

Question: What do you mean by a semantic interface?

JMS: You must have an interface of some sort. But the interface need not be arbitrary. In cognitive modeling, the model essentially is the interface. To accomplish this, there is a need for generic
components (you must be able to maneuver in any model), on th( one hand, and domain specific components, on the other hand.
CA: About the project you just described, you said you reengineere the 100+ modules.
JMS: Reorganized and wrapped the code.
CA: What’s the difference in size?
JMS: A bit smaller but size was not the important difference. It is the difference in maintainability. We no longer need to worry about different functions interacting.
Question: Can you say more about systems and cognitive repositories’
JMS: Again, recall how we view a system (Fig. 15). Every CASE too has a repository, a place where information about the system is kept. Basically it’s a database. You have access to all information in the database. But, changing the database and changing the software are two different things. In our cognitive repository, the: are one and the same. They never get out of sync.
MW: The update problem(!) There’s the update problem, keepinl everything in sync.
CA: There’s also an abstraction problem.
MH: In the repository, you have a lot of data. You also have program for accessing that data. All that stuff is stored in the form of data which is a powerful device. You store instructions in the same form as data. It has certain drawbacks, however, using exactly the sam. form for data and system software and application programs.
JMS: In a central repository you can get information about the system but you cannot change the system by changing the repository. In Flexsys cognitive repository, if you change the system, you change the repository.
MH: What does that mean though? If you have all the information about the system, and you modify the system, you modify what you stor on the server. I'm not clear as a mathematician, that there's logical distinction between the two. But, then, you're emhasizin modification.
JMS: It's not the logical aspect as much as the way in which it is done. You have files representing the system and files representing th repository.
JMS demonstrated these issues using the Flexsys system itself. He showe. examples of Flexsys modules, call hierarchies and unit hierarchies.
containing embedded functions but without a permanent memory. It is very fast. We can reverse engineer 600K lines of code in about 3 h on a low end DEC Alpha.

More important, we also have a higher order design (I-ILD) language making it possible to do whatever you want. It can be used to create custom tools to do pretty much anything you can describe in English. For example, we recently made a custom code slicer to search COBOL code for those statements which use given variables.

(This was demonstrated using the Flexsys system. In response to questions, the demonstration also included a review of the way systems are represented and documented in the cognitive-based FLOWform repository.)

MW: That separation of code and abstraction kind of implies you don't like embedded comments. You shouldn't have code and comments mixed at the same level of presentation.

JMS: There are two common styles in commenting. Some people comment individual lines of code. I prefer to keep that to a minimum. Such comments should relate specifically to the statement (in question) and not to anything around it. The latter kind of information you want at higher levels. In general, in commenting, say as much at each abstraction level as necessary but no more. We've found that doing so gives a tremendous advantage in maintainability.

MW: It seems like this would enable or at least not conflict with the use of compiler directives to maintain multiple interleaved baselines.

JMS: In fact, we can simplify compiler directives (in FLOWforms) by attaching simple labels to anything from an entire module, or complex structure down to an individual statement.

In the latter case (Fig. 18) SCANDURA initially didn't support VAX Pascal, a very rich language, or conversion to C/C++ — just basic Pascal support. The time to perform various functions is listed in Fig. 18.

BL: Regarding the MacDonald Dettwiler project, do you have man hours? Time doesn't exactly tell you how many people worked on it.

JMS: It was much less than you might expect. Imagine if a Unisys or Loral were doing this kind of job. We're talking a factor of 5-50 less. Initially, we thought it (the Fort Monmouth job) was about 200k lines but it turned out to be over 400 (Fig. 19).

MW: In the previous example, you changed OS as well as languages.

JMS: There were some issues that could not be handled statically.
Application of Methods and Tools:

Projects Undertaken Informally with Limited Amounts of Training

- **Redstone Arsenal**
  - Conversion of FORTRAN Simulation Program to "good" Ada

- **Johns Hopkins University/APL**
  - Real-time C system to be integrated into Cooperative Engagement Environment in Ada

- **Ft. Monmouth**
  - Conversion of C graphics simulation system into Ada

**FIGURE 17**

Application of Methods and Tools:

Major projects where SCANDURA worked closely with Project Personnel

- **MacDonald Dettwiler Associates**
  - SYSTEM - large imaging software
    - (600K LOC in VAX VMS Pascal)
  - GOAL - convert to C++ on UNIX
  - INITIAL TOOL STATUS
    - generic support only
    - no VAX Pascai support
  - SCHEDULE
    - completion in 6 Months
  - RESULTS
    - customization:
      - reverse engineering - 6 weeks
      - conversion to C++ - 10 weeks
    - automated reverse engineering - 3 hours
    - 99+% accurate automated conversion - 6 hours
    - on budget 5 weeks ahead of schedule

**FIGURE 18**
Application of Methods and Tools:

Major projects where SCANDURA worked closely with Project Personnel (cont.)

- Ft. Monmouth

**SYSTEM** - simulation code

(400K LOC in C on UNIX)

**GOAL** - analysis to determine desirability of integration with Army COE

**SCHEDULE**

- 2 1/2 weeks

**RESULTS**

- full assessment of simulation code including recommendations and projected schedules for integration

FIGURE 19

MW: Some applications are full of OS dependent utilities.

JMS: We introduced abstractions wherever possible to provide equivalent functionality.

MW: Was this automated?

JMS: The abstraction code had to be written, but the conversion itself was done automatically.

MH: What did you actually do?

JMS: The goal was to determine the feasibility of extracting "golden nuggets"? We first reverse engineered 400K lines of code. Using the reverse engineered code modules, we got call hierarchies, metrics, uses relationships, etc. We also performed some custom analyses. Custom tools were actually built to get specific kinds of desired information.

JB: Were you able to look at the API's?

JMS: We only did some preliminary work here due to the time constraints. The bottom line is, if you can specify what you're looking for, a custom tool can be constructed to do it. The hard
part is figuring out what you're looking for. The implementation is easy.

BL: About MacDonald Dettwiler (MDA’s), you said you took so much time for reverse engineering and so much towards compiling. What level of testing and assurance of correctness was done on that problem?

JMS: Again, our job was to get it to 99%+ level. MDA’s job was to finish the compile and testing. We could have done the compiling. This was mission-critical software in the extreme so they wanted to deal with run-time issues because this requires application-specific knowledge.

BL: After compilations, did they have tests run for comparison?
JMS: I understand they had satellite imaging test suites.
This slide (Fig. 20) refers to reengineering PRODOC into Flexsys.

**Application of Methods and Tools:**

**Major projects where SCANDURA worked closely with Project Personnel (cont.)**

- PRODOC re/NuSys Workbench

**SYSTEM** - 500K highly interdependent LOC in C with many supporting "meta" utilities

**GOALS**
- modularization of key reengineering and conversion functionalities and meta utilities
- integration of modules into semantic environment
- new modules for semantic wrapping and integration
- GOO methodology and supporting design/development environment

**RESULTS**
- semantic "plug and play" capability
- ease of maintenance
- integration layer improved by factor of 50+
INITIAL STATUS OF TECHNOLOGY

- Visual FLOWforms easily modifiable
- Full source code automatically generated from visual FLOWforms
- Change FLOWform in one window; compile in another

OBSERVATIONS

- Visual maintenance routinely saves 25-50% in time and costs
- Generation of source code from FLOWforms originally not sufficient where repeated compiling and numerous "quick fixes" are needed
- Users unfamiliar with visual environment resort to old habits of making "quick changes" directly in source code
- Code changes get out of sync with visual representation
- Reverse engineering can be repeated but non-code additions can be lost

FIGURE 21

BL: You say semantic plug and play. Does the semantic environment provide this? I'm talking about a code interface and you're talking about the Flexsys interface.

JMS: I'm talking about the Flexsys components. They all work independently. They can even be plugged into other tools.

We did learn some lessons (Fig. 21). Because of where the tool was at the time and because they had a limited amount of training, those with limited training weren't able to take full advantage of what was available. In conversions, for example, there was as strong tendency to make "quick fixes" in the code. Consequently, FLOWforms get out of date.

Ideally, one wants to compile then pop back into the FLOWform at the point of error for editing. In effect, you have a visual integrated compile and edit environment.
BL: That was our problem before. We would essentially have to get out of the tool, then get back into the tool, and put the code back and re-comment, etc.

MW: The annotations in the FLOWform don't get reflected in the code?

JMS: You have your choice. You can generate comments with the code or leave it out. We generally leave them out because all editing is done in the FLOWforms.

This kind of visual maintenance, documented pseudocode in FLOWforms gives dramatic improvements in maintainability. Programmers can get into unfamiliar code much more easily. Even the original programmers warm up to it more quickly.

MW: Can you put constraints on FF, to enforce coding style?

JMS: To date, we only check for valid structures, syntax and unused variables. One could do more, however, and that is a very interesting area. Check logical consistency (discussed in AM) during the design and implementation process could be invaluable for any supervisor.

JB: Do you store this kind of information in an internal repository?

JMS: FLOWforms are both the system and the repository. One doesn't need another repository. One could, of course, also put information in a central repository of the traditional sort, but here you only get access to information. You must update both the code and the repository to keep them in sync.

MW: How sharable is the FLOWform repository? If XXX wants to work on one line and I want to work on another part, can we do simultaneous updates? How configuration manageable is it?

JMS: We don't have what you'd call a configuration management tool. Inherently, Flexsys does not require configuration management as much as ordinary code does. On a network, for example, one person can load a FF that someone else is working on it but cannot change it. One would (almost) never want two people working on different parts of the same FLOWform module. Because each function, etc. is a separate file, it is easy to maintain totally different versions by judicious use of directories.

BL: One thing I'd suggest on that chart (Fig. 22): another thing we had to do is testing. It was more than translation. We were modifying the architecture as well. We had to test very carefully at every major level of change. This would be another helpful thing if you could compare executions of an old version and a new version.
RESPONSES TO PROBLEM 1

- Completed Link between FLOWform visual representation and source code: automatic return to point of error in visual FLOWform
- Direct link to "grep" and other utilities
- Easier to make "quick fixes" in FLOWforms than in code using grep, etc.
- Code changes automatically propagate to system-wide information

FIGURE 22

JMS: That requires work at the source code level per se not just at an abstract level. It's an important problem, but we've made the decision to avoid run time analysis, leaving that for other tools. We also decided we will not try to make GUI tools. Other people already do such work, and do it well. What we do want to do is wrap components (and/or tools) and make them easy to use. We did this with the dialog boxes you saw today. Or, communications software or database access - we'll just wrap it and use it.

Short term training was not always sufficient (Fig. 23). When dealing with odd or unpredictable cases, what do you do? It just helps to have someone with experience when doing that sort of thing. As everyone knows, dealing with difficult problems over a phone line is not always easy.

We've moved into giving the user - the educated user - the ability to make customizations themselves. We've made our powerful meta-tools available to partners. The best use is where you've got people doing similar things over and over, in large service projects for example, where you get the benefit of automation, but the learning curve of customization pays off as you get more and more similar projects (Fig. 24).

There are all kinds of different projects out there --- in reengineering, conversion and integration. One cannot prebuild solutions for all variations. New problems or issues are going to come up. You need to be able to build custom tools as you need them. That is what our high level semantic facilities are for.

BL: I would like to say that that's real important. Joe was really busy, probably with that (MDA) conversion project. We found a lot of
INITIAL STATUS

- Short term custom training using project code
- Macros run in automatic mode for performing simple jobs
- Macros help to learn key strokes

INSIGHT

- Modularity and semantic integration open possibilities for FlexTutors
- FlexTutors can easily be created to:
  - perform complex tasks automatically
  - provide arbitrary levels of training
  (FlexTutors build on existing components -- Reverse Engineer, Code Generator, Translator, Reorg, etc.)

RESULTS

- FlexTutors interactively tutor or automatically perform reengineering, conversion and integration tasks
- SCANDURA provides services on one-time projects
- Strategic Partners license technology on continuing basis

FIGURE 23

ways we would have liked to customize the tools if we had had a vehicle to do it. We felt 99% could have been automated.

JMS: Regarding these new metatools we have learned that it's a mistake to get people in over their heads. One day of training is enough for simple editing and code generation, but not for real complicated reengineering or conversion projects. Certainly, learning to use metatools, etc. takes time.

Next, a reverse engineering and conversion demonstration was done using an automated tutor. This tutor was just another application constructed with Flexsys high level semantic facilities.

It was written in Flexsys and executed in Flexsys. Flexsys can be used to generate compilable code from the tutor design. After compiling it can run as a separate application.

Major steps in the demonstration are described in the handout.
OBSERVATIONS
• High degree of variability in legacy code makes customization of reengineering and conversion solutions essential
  Language dialects and vintages, application specific features, goals and constraints make each project different

INSIGHT
• Don’t just keep "meta" tools in-house
• Package for easier use and make available to strategic partners

RESPONSES
• Flexsys Reengineering Factory
  - Perform enhanced and/or custom reengineering tasks in high level design language -- cover increasingly wider areas

• Flexsys Conversion Factory - Modify grammars/processing in easy-to-use high level design language
  Support new languages/variations on existing ones

• Flexsys Semantic Integration Factory
  - Use automated semantic wrapper, semantic "glue", semantic integration technologies and cognitive GOO 'development environment to integrate existing applications and components (support via Flexsys Reengineering and Conversion Factories )

Create: custom domain-specific application development systems to increase productivity and maintainability by factor of 50 or more

FIGURE 24

CA: Can some items be in the call hierarchy in several locations?
JMS: Sure.
CA: Is there a way to flag that?
JMS: They are flagged as clones and a blinking cursor.

Some people make almost a fetish out of computing metrics for programs (Fig. 25). Some metrics tools sell at a fairly high price, even though most of what they measure can be obtained rather easily: Cyclomatic complexity for example. In fact, with the simple graphics library we wrapped recently these things could be represented graphically as well though our QA Measure does not do that yet.
Reengineering

**Issues**
- How to keep Module and System Semantics in Sync
- What Kinds of Information Help Most in Reengineering Software
- Which Reengineering Tasks can be Automated

**flexsys Technology**
- Builds on Semantics-Base Obtained via Reverse Engineering
- Automated Tools for Modifying Software Systems
  - Update System Repository
  - Make C Header Files
- Higher Order Semantic Language for Building Custom Automation Tools
  - Reorganize System (Eliminate Cyclic Relationships)

FIGURE 25

Conversion: No nontrivial conversion is 100%. Customization makes it possible to get what you want. Flexsys conversion machinery works both from the bottom up and the top down level. Bottom up works at the syntactic, token level. Top-down works at the semantic level — adding Ada packages, for example.

Each year we keep finding new uses for Flexsys Implications of the simulator being able to call the simulator are especially interesting to me because rules and higher order rules are crucial to cognition. They also make it possible to envisage systems which create needed programs automatically.

HP: I'm getting the impression that Flexsys is a run time environment in addition to a reengineering environment. Have you been looking at CORBA which has some of the same features without the composition facilities? In other words, instead of building your own run time environment, build on top of another run time environment, though CORBA is not yet there.

JMS: Yes, that's exactly what we want to do. The difficult decision is not what can be done but where to put one's limited resources. Do you spend time trying to encapsulate OLE objects? Do you do it with CORBA? What we've done and we will talk about this tomorrow is to automate the encapsulation of basic C libraries.

HP: The problem with this kind of approach (simulation) is you're asking the customer to buy into your architecture unless you
generate code and compile it separately. For single systems that makes sense, but if you’re going to be doing this across organizations with multiple customers or over a distributed environment, then you get real resistance.

JMS: We’re definitely not trying to impose a standard. What we want to do is work with and build on what these other folks are doing. What we’re talking about is independent of whether you’re buying into CORBA or OLE or any other standards vying for control. Whatever the standard, we want to make it possible to build models in an intuitive way using your own names for processes data structures etc. We’re proposing a cognitive-based, semantic environment which really doesn’t care what’s underneath.

HP: CORBA already doesn’t care what’s on top.

JMS: But neither OLE or CORBA provide an easy to understand, let alone semantically meaningful interface. We can accomplish this by adding a very simple and meaningful abstraction layer. This allows one to build and maintain system at a very high intuitive level. By integrating general purpose and domain specific component with the Flexsys engine you can do even better. You can actually execute fully implemented applications as well as design in the Flexsys engine before generating code for a separate application. The same is true of RPC’s, etc. We see tremendous potential, we also know that developing the full potential will not be trivial. The possibilities are enormous, however. Suppose one were to semantically encapsulate the Army COE. Other interfaces can have arbitrary characteristics. Our interface is nothing more than a map directly into the real world. That’s what makes it so revolutionary; and so useful.

HP: The big problem I always have is people who come at it from a technical attributes of the implementation design perspective instead of the system as a representation of some business process or model. If you go into reengineering a system without looking at that part first, to me you're going to be spending a lot of resources for little gain. More often than not, since the last time you reengineered your setup, technology has advanced sufficiently that it’s time to revisit why you built these systems in the first place.

JMS: How do you coordinate with what you have?

HP: You have an “island of automation” and what you really want is something more integrated or tied into other “islands” — but just tying those things together opens up new ways of operating.
OVERALL PROCESS:

The overall process includes the following steps:

1. Create/view a sample library specification.
2. Semantically wrap specified components.
3. Select and integrate wrapped components; generate C source for new library and compile.
4. Create Domain Specific Development System (DSDS) by linking the .ctexsysw semantic shell with component library.
5. Use DSDS to create/view custom application — a "real world" model referencing domain-specific library components.
6. Test custom application.
7. Generate code for custom application; compile and link.

FIGURE 26

JMS: We've been guided by a cognitive perspective throughout our development. One of the strengths of this has been to keep things in focus — in one coherent picture. One has many choices to make in building a system. The cognitive constraints we are talking about can save considerable time, not only during development but subsequent maintenance.

CA: It seems to me the most significant contribution you've made is to save the implementor from having to deal a lot with the nitty gritty — so he can devote more cognitive skills to the higher level stuff that's more important.

HP: If I've got a new model of how I want a system architecture to look — a template or model of where I want to go and I've now looked at the existing set of components, I can guide the existing structure towards the model.

JMS: That is basically right on where we want to go.

(Figure 4 cool car picture again.)

HP: There's a tool called Capture by CTA (in Rockville) developed for NASA. Basically the focus there was on allowing a comprehensive/contrast of alternative designs, capturing the rationale of why one design was preferred over another in different circumstances.
JMS: ... taking the available components into account?
HP: They allow you to stack windows on top of each other and
basically compare them along with annotations that the developers
prepared at the time. You might want to look at integrating that
kind of tool into your environment. They've thought about com-
mercializing it but decided not to do so.
FK: I'm interested in how do you translate from an old legacy
language to a new language. I assume now you do it mainly on a
syntactical level; you identify certain constructs in an old language
and transfer them. I'm interested in how you construct those that
don't exist in an old program, or vice versa. Can you use the context
maybe comments, etc.?
JMS: We do, in fact, work at a semantic level as well as the syntactic.
It's not just low level implementation issues but also the overall
architecture of the system (i.e. Ada packages). Also, there's no one
answer. One cannot come up with a generic solution to how best to
translate C code into Ada. So the more you can take advantage of
special characteristics of the application, the better the conversion.
FK: In the process of doing a conversion, I imagine there are
situations where an experienced user would recognize familiar
situations and use corresponding solutions. I would like to capture
that using neural networks to automate solutions.
JMS: Neural networks work from examples. But they take time. It's
interesting to consider how practical that approach will be.
HP: When people talk about representational systems, it's really a
knowledge representation system. How do we do that?
JMS: Interestingly, there's a whole methodology called structural
analysis used in the study of cognition that goes back into the 60's.
It has been used to identify cognitive content, including higher
order knowledge. It involves analyzing what a human would need
to know in order to perform successfully on a given class of tasks,
including creating new things. The current technology now makes it
possible to think about implementing these ideas. Twenty years ago
we couldn't implement any of those things. It's mind boggling when
you envision what could be done — creating programs automati-
cally for example. There are many things requiring a lot of heavy
duty human activity. I believe we are on the threshold of being able
to automate many of those activities.
HP: The real point we made in the NIST proposal is we're still not at that point, but what you can do is rapidly generate specific knowledge representation languages for specific domains. So you could have languages for many domains of knowledge. That of course, isn't terribly useful if it takes years to produce each language. But, if you can produce meta-technology to produce the languages, then it becomes more useful.

JMS: I think that's quite right. When you get into a different domain you need a language tailor-made for the domain you're talking about. The nice thing about cognitive modeling and programming is that you don't need a language. You invent whatever you need whenever you need it.

FK: Can you deal with tricky stuff like self modifying code?
JMS: Self-modifying code is code that modifies itself. That is exactly the higher order programming we've talked about does. Conceptually the idea is definitely not strange, and can be accommodated by the Flexsys simulator calling the simulator.

CA: Are the tools adaptable to use with a language like LISP?
JMS: It hasn't been a high priority. The environment captures a lot of what you would do in LISP but it's easier to use since LISP is heavily recursion dependent. Some things lend themselves to recursion, but there are other things which do not. It is better to view them as iterative or whatever else is required.

HP: Do you have a demo disk?
JMS: We're looking at the best way to both disseminate our ideas and do so on a self-funding basis. Obviously, you can't give away core technologies.

HP: If you give out executable software, you aren't giving away the ideas.
JMS: On the other hand, if you give away how something's done, and that something is relatively unique, you've already given away something.

HP: DoD contractors probably have shelves full of interesting software that never went anywhere because they aren't in the commercial software business. But every one of those companies will fight anyone who suggests giving that stuff away. So I'm suggesting a more dynamic approach to preserving your intellectual property — namely staying ahead of the market. I'm not talking
about giving stuff away either but what you're selling is you]\nconcept of how to do this. There aren't many people who would
pick that up and run with it without your involvement. The risk is
low in my assessment that someone would run off with it. The rea
question comes down to whether you want to be a niche player or
whether you want to affect the general approach of how people do
things.

JMS: We've been toying with a lot of different distribution models
They all revolve about the idea of partnering with various pro-
tections for certain key technologies. Service vendors are lookin
for ways to reduce costs and we can save them money. The
typically have specific tools (e.g., a translator) in-house. What the
don't have are generalizable solutions that allow them to deal in
generic way with an ever widening array of design development
reengineering and conversion solutions.

(HP mentions Template, creator of "Snap")

MH: In my experience, three people three instances: One is Harlan Mill:
who is a salesman but can talk to scientists. He picked up ideas here
and there. I mean structured programming is due to Dykstra. Man:
people think that Harlan Mills invented it. The walk-through was
due to Jerry Steinberg or whatever his name is. Other component
also were compiled from various sources of good ideas which Mill
put together and sold. The crucial event was when he got IBM to
adopt it. The beginning was the use of structured programming to
do an event for the NY Times. It was actually done by Mills and
another experienced programmer. Mills built on that. The fact that
he got IBM behind him was his key to success.

Second, is Raymond Woolridge. Here were a couple of academics who
were well thought of in the aerospace industry and got themselves a
advisors to the Air Force — in a position where they were reviewing
lot of the AF research contracts and telling the Air Force what was goo'
and what was bad. They were not doing the work themselves but the
were skillful monitors. On this basis they founded Raymond Woolridge
They became millionaires; sold Raymond Woolridge to Thompson
products and both went back to being professors. As with Harlan Mill
you sell your ideas to somebody with a lot of clout.

Third instance, of course, Gates. Gates paid a pittance for DOS. Whe
IBM needed an OS for their PC, Gates resurrected the Palo Alto researc
and sold it to **IBM**. Now 85% of OS for PCs in the world use DOS. Later developments were built on top of it. Again somebody with an idea found a big buddy to associate with. **HP** itself was started by very practical and innovative engineers who put together a company and got supported. Both started out more or less as professors of engineering but decided to become businessmen at an important moment. **HP** is now one of the most successful companies in the computer business. The key is to find somebody that needs you and has a lot of clout.

**JMS:** We are now working with a company that is very good at setting these things up. They've had a lot of experience internationally. We want it to stay in the U.S., so we're looking there first. Of course, timing is important so potential partners must act in a timely fashion.

**JMS:** Now that we have had an opportunity to present our ideas, I would like to open the discussion more broadly. What we've been talking about are extremely powerful ideas, and they could really change software. We cannot continue spending the amount of money going into maintenance and for minimal effect. We need fundamental improvements both in technology and the way we do things.

**CA:** I'd like to suggest we take some actual code and take a look and see how you work through it. Take a module, reverse engineer it. I'm not clear on how a module or a set of modules is reverse engineered, then is translated into something else that carries out the same function. Even though you demonstrated the process in some detail, I probably missed some things. Things happened so fast it was not always easy to figure out what was going on.

**JMS:** To recapitulate, we first reverse engineered about 400 lines of **FORTRAN**. Then we got some information about it: use hierarchies, metrics etc. Next, we took the same code and ran it through the translator. The translator first did the syntactic stuff, then came semantic postprocessing, creating packages and things like that. After looking at the results, we generated actual code. Finally, we compiled and ran it. It did go by fast, but that's the nature of automation. One thing we could have done is shown some of the meta debugging tools in operation, showing you production by production as it goes through the code.

**HP:** What if you have access to only part of the code? You want to componentize the legacy system, some of which are commercial components for which you don't have the source code.
CA: There are legal questions if you want to do that.

JMS: Legal questions are not an issue. We don't reverse engineer commercial components. Semantically wrapping pre-compiled legacy code is fairly easy. You need a published interface interface, of course.

CA: One kind of problem your tools might be able to help with is that sometimes the DoD pays for code and then the contractor refuses to deliver the source code. There's a lot of object code that doesn't include source code. If you allowed the reengineering of that -- where the vender hasn't supplied the source code.

JMS: We do not reverse engineer object files. That is not our business. There are tools that attempt this, though I don't know how effective they are. What I'm talking about is wrapping compiled code, then using it.

CM: I'm not sure where we're going but we're on a fast track to get there. We're a Navy support design agency and provide ADP systems for wholesale supply, procurement, payroll systems, accounting financial type systems. We're very wrapped up with the whole CIM issue with DoD competing to have some of our systems become the migratory CIM system. In some cases the decision has already been made, and our systems will become a part of some other system or replaced by another system. In most cases parts of what we have will go away. We have a whole spectrum of issues — were trying to compete for the CIM work trying to protect our customers (Navy supply system). We're looking at how do we encapsulate those core business functions that will not be replaced so when the standard system does come in we can take our encapsulated portion and still have it functioning in whatever environment the CIMs systems run in. Our software for the most part — I haven't seen a count lately-is probably upwards of 20 million lines of code mostly IBM big Iron, MVS. Some of the design could trace back to the late 60's and has not undergone a major reengineering effort in 20 years+ . We're looking at ways to bring new technology to a number of those areas addressed, legacy systems as well as migrating those things we need to encapsulate and get those golden nuggets to the right side of the chart.

HP: Are you talking about still running them on the MVS environment or reengineering to run on UNIX server or NT?
CM: Ideally we'd like to encapsulate them and bring them across.
JMS: There is not likely to be one objective that would be the best in all cases. Goals will depend on the status of the code. For example, you probably want to throw out anything I/O related at the first possible opportunity. Other things may have been performing some task ably for 20 years. Just keeping it running may be all you need to do. One thing we've found in any big project, is to split the difference. Break the problem into isolatable pieces and keep the system working at each stage. If you cut the problems in half you've made it only as difficult to debug and fix things at each stage.
CM: I think funding will ensure we do it in that manner.
MH: DoD is peculiar in that for years cost was no object. That's how a lot of things were started. Now there's an effort to downsize, terminate, etc. It seems to be one extreme or another. It isn't easy however. For example they're still paying off Sgt. York which was terminated 10 years ago. There's a lot of good stuff which could be salvaged and useful. Just because it was written 20 years ago doesn't necessarily mean it can't be useful in a modern age — It is just going to be lost, put in a warehouse somewhere, and no effort will be made to use it.
HP: I think it's not so much the code being good stuff but it has domain knowledge about your business that has been lost.
MH: There are some programs that are stable (airline schedules, etc.). They've moved platforms or added front ends to be more user friendly, but the applications have not changed very much.
JMS: If it's not broke, don't fix it. If it's doing what you want, you can just isolate the stuff. Even if it's the lousiest code in the world, if it does what it's supposed to do, keep it. If it has general value, you may want to semantically wrap it so it can interoperate with other components.
AC: How will this apply to software testing. You said earlier you could reduce the number of test cases from $2^{10^3}$ to 300.
JMS: In one sense, the idea is not terribly profound. The problem is that people don't know how to take advantage of it. If you test and debug your system as it is being developed, this will dramatically reduce testing. The disadvantage (of clean room) in this regard is that you have to specify all inputs and outputs beforehand, which is infeasible.
One key to cognitive design is making much more precise what has been implicit in structured walk-throughs and the use of stub programming. Guaranteeing logical consistency as you create a design is a big step. The second key is automation. You build on previous work. You don't have to throw designs away as is the case with code stubs. Consequently, there are no limits as to the scalability. Regardless of the size of the system, you proceed the same way. The question is whether you have the necessary knowledge, discipline and tools. Nevertheless even support for dynamic testing and not (like clean room) having to throw away code may not be sufficient because some programmers like to break the rules. Managers would have a lot more control if they can impose or have the option of imposing discipline on the design and implementation progress. Let your good programmers have the flexibility they need. Constrain the ones you're not so sure about.

AC: So you're saying it only works if you start from the very beginning. After you have the software, you can't do it.

JMS: Not necessarily. Suppose you have a lot of software and you componentize that software. The components may or may not be CORBA, OLE or ACOE compliant. We have developed methods that can be based to extract components, to make them accessible, and provide them with a uniform semantic interface. Currently the last step can be done automatically with standard C libraries and C++ classes. You can also wrap other languages in C. We have not done it yet with CORBA or OLE.

AC: Suppose we have C API's and we need to write Ada API's for response to the C API's. Can you use this tool to translate the C API's to the Ada API's, or is that a different translation?

JMS: I think what you want is a binding, which is a little different. It could probably be automated. Details would have to be analyzed, then we could come up with some solution techniques. When you work at the lowest code levels, however, efficiency must be taken into account. Automation may not be the best way to go. Semantic wrapping works best with utility libraries, ACOE, OLE or other components. Components wrapped in this way are interoperable, not only with each other but with abstract/virtual components as well. This is what makes them so useful for analysis and design purposes (as well as implementation).
END OF FIRST DAY

Second Morning

In many ways, abstraction is one of the most important things we are talking about. It constitutes the core of what we do. So far we've talked about creating logically consistent designs, reengineering and working with legacy code to some extent. We've talked about reverse engineering and reengineering. We haven't talked much about wrapping, or how to put wrapped components together to create Domain Specific Development Systems (DSDS). Any observations anyone wants to make at this point?

Alex raised the question yesterday about testing. How could you use these ideas in testing existing code? If you have a system, you can't possibly test everything that's there. On the other hand, if you have a legacy system and you know the functionality of it, you can create a logically correct design that models the functionality. That's important because you can use that design to select test items that will get at the particular functionality you're interested in. It provides an overview or roadmap as to what you should test.

MH: It's true that the worst errors are the ones at the high level. Design errors which eventually get incorporated in code. It's not so hard to correct bugs at a low level, but when you have to go back and find and change the program at a high level; that's expensive, and something to be avoided. That has often been a prime reason for delays in developing systems. People write software then find for some reason it doesn't work. Then, they work frantically to find out why. They sometimes have to validate both hardware and software, causing greater slippage. The customer was the government in the old days. The government needed the weapons systems or the aerospace systems and tolerated the slippage in time and cost. That was a big problem and it's gotten much bigger. Anything that helps to improve design early in the game is very worthwhile.

JMS: If cognitive methods are used systematically there's no question they can help dramatically.

. . . as regards testing, you can certainly get information about code and focus your testing.
FK: That raises a crucial question — namely metrics. A lot of people need to use metrics. How do you use metrics?

JMS: We view metrics as just one of the many kinds of information you can get from a system using our semantic tools, custom or otherwise. The most interesting part, I think, is being able to improve the system automatically.

JS: In terms of economic models, the Air Force is building a reengineering model. Given they don't have a lot of data to run the model, it's difficult to tell how valuable the model will be. Looking at QA Measures perhaps you can talk to those people.

MH: Or, for example, Motorola who works for the DoD has about 12 different metrics. There's no metric that dominates. The famous cyclomatic metric which tells you the complexity of the code doesn't take account of the fact that some convoluted paths are used very rarely. What you really want to know is when the code is running which parts of the code are the ones most used. The parts that are infrequently used are less important.

HP: It's back to whether you have a tech focus — looking at code attributes from a maintainability viewpoint or whether you're really trying to map your software components against business processes. What you want to invest in is what will persist over time. Focusing on technical attributes is useful for some purposes, but that has limited benefits for an organization.

... critically important are extraction and reuse issues. Clearly that's a very important topic.

(going around the room)

MH: I read an article this morning saying wrapping is not necessarily a panacea. Problem focused on was updating legacy code mostly for mainframe computation into the 00 age we're moving into. One of the difficulties is finding good modules, wrapping them as 00 objects and sending messages to them. They say its very difficult to sort out from mainframe legacy code what are good chunks to wrap.

HP: My experience has been that the main criteria is whether the existing software has some well defined functionality with low bandwidth interaction with its surrounding environment. The more interaction the harder it is to encapsulate and use in other contexts. Other issues are that components need to be as sound as possible.
Ideally, it's self contained and doesn't make a lot of assumptions about its execution environment. Unfortunately most software doesn't fit that. Development practices in the past often had a good sense of an architecture and implemented software according to an architecture. Often, if you are into one of those architectures you cannot extract at a subsystem level and use them in a new context — though it's a crap shoot.

FK: I'd like to comment on use of current software. I have a lot of pieces of software written by students/colleagues but my working environment changes often enough that I'd have to spend a lot of time to get these systems running. Worse, some of the software is written in different paradigms. One parameter for example is based on symbolic programming, others on numeric programming, vision algorithms and neural networks. I'd like to use the vision program and integrate it with some other programs. Even if I have the syntactical interface, it is difficult because the semantics don't fit.

JB: 60 million lines of code have to be moved by SOMEONE. I have to counter the Ada thing every time I talk with clients at the DoD and I have to explain the risks to them. And software is mostly not documented. I'm an old programmer, so when they talk about structured programming, I think I've been there. How big is a module? No more than two pages we used to say. This plays right into that. Go in, analyze, get rid of what isn't doing anything, encapsulate it and reuse it. It works, don't change it. I'm trying to convince Lloyd to take COBOL mainframe and move it to Ada 95. I come from AT&T where 60 million times of code have been sitting there for a long time and they aren't reengineering it because it doesn't work.

AC: I'm currently working on common software for the Army and it's being extracted from different Army systems. We had some problems getting the right software before, and we had a major architectural change just a year and half ago because the code we choose as a COE component is not independent. It depends on its own database which conflicts with our database. So we have to throw out their code and look for other software which is more modular and independent with a well defined interface. I think the key is to choose software that is independent, not have too many dependencies and have a well defined interface.
JB: The taflam is a good document with good ideas on how to get independent components. The DoD takes that document and shoves it in people's desks and don't use it. If they did what they said they'd do, they wouldn't get into these things — and the Oracles and Sybases would come around and you wouldn't have these proprietary kinds of tools. We're doing a project to prove that, moving software from one platform to another environment.

HP: The taflam just gives you connectivity. It doesn't give you semantics. You don't have application level architectures in the taflam. It doesn't buy you the kind of application interoperability that we're talking about here.

CC: I have a small lab and part of our mission is to extract "golden nuggets" from legacy systems and prototype systems with a view toward putting them into the ACOE and/or using them as a foundation for other applications. As a personal comment, on reuse, if you start with an architecture and build your software for reuse I think you've got a chance of reusing it. Some of our leadership believes we can extract golden nuggets from any system (and build a new application). But that doesn't work. It's easier just to recode. Nevertheless, here's how we do it. First we characterize the software as best we can; build a functional matrix describing what it does. Then use various tools to try to collect metrics to figure out how the thing is built. We do a lot of code reading to try to figure out how it works. We also try to isolate those features that we're told will be beneficial for other applications. Then, we make a decision as to how to do that — either to extract the code literally or just take the idea and rebuild it. So far most of the time, it turns out to be easier to recode the feature in the other application. That's because a lot of these prototypes are prototypes and they don't work well. In this situation, it's very hard to save the code.

CA: I'm fascinated with this approach but think we need to get away from looking at the trees and look at the forest too. The overall problem is we need to generate within the next two years vast quantities of software and we can't do it like we've done in the past. We need a way of converting our needs into code, quickly and that means not having to redo anything we've already done. We also need to generate a lot of
new software, quickly and reliably, and get it out in high quality and in
higher quantity than we've ever done before.

CM: In looking at the problem of reusing existing software. When you
talk about legacy code that could be something you did two days
ago or two years ago. For some of us legacy software fits into the
category of being quite dated not only is the software quite dated
but also the business processes it was originally designed for are
dated. I think you need to relook at the components and see if the
cOMPonent has the right type of functionality for today's type of
business operation. If we were truly honest with ourselves we'd say
we need to do the functional analysis again. That doesn't mean
there aren't some reusable components in what we have. The
difficulty is trying to isolate from those things so we can expect a
reasonable return in getting them into a new environment. The ideal
situation would be where components are process oriented or
computation oriented as opposed to doing a lot of I/O. In our own
situation, the things we have that would be most easily reused are
black box math models used to predict needs of various categories
of suppliers throughout the Navy. Those types of things where we
feed parameters in and they spit the answer out, with no heavy I/O
functions — lots of FORTRAN computational code fit this bill and
are probably the most easily extracted functions.

JL: I work with Charlie and what he says is obviously true. Those
kinds of reusable modules are the things to reuse. However, we're
faced with another problem. Perhaps it's not a reengineering
problem *per se*, but we have standard systems coming down the
road. We have old code which should be reengineered but some
shouldn't be reengineered because it will be replaced. The problem
is trying to find what needs to stay and what can go. In the CIM
world business processes and business functionality were defined
based on a consensus of all the services. We ended up with a lot of
code, much of which is throwaway and much of which is not. In
that universe the trick is to turn yellow nuggets into golden nuggets
because we have to capture and maintain them.

JE: We're doing a lot of related work. The business case for
reengineering is an economic thing. You are either deferring costs
or reducing costs. Somewhere over 20 million lines exist out there.
New systems are not in Ada because they hadn't picked the
hardware. Managers wanted to defer costs, not knowing if it would be used. But it was sitting on old machines and we wanted it to run the new machines. I think you usually get better economic benefit rewriting small components rather than wrapping large components. When one talks about redoing a system, it's not a one time thing. The system evolves and will continue to evolve. Given that the system is going to evolve, one thing you need to look at in terms of tools used and technical analyses is what do I do now that will improve things in 5 to 10 years. The things we do today should reduce maintenance costs and allow us to move hardware but also reduce future maintenance costs. We should insert standard interfaces or db architectures. We don't do this but we should. This is why reengineering tools are useful. Also, data must evolve as process evolves. This is a major problem for DISA and people I deal with in other engineering environments. The data stay around forever. As you change the processes, you want to evolve the schema.

AS: I have a kind of "big picture" cognitive approach/point of view, so I'm finding this stuff fascinating from an academic point of view. Practically, I think we're putting the cart before the horse generally in the DoD. I think we have a bunch of shotgun efforts that are uncoordinated. We have not agreed on a definition of what domain means or what architecture means. We don't know if the two classes related to each other. As far as I know no one is even attempting to make that happen except for these scattered approaches. If we did have definitions we still wouldn't have sponsors and owners of them. Our budget process requires end product tangible systems, not infrastructures, architectures, and stuff that might make those end products cheaper in the long run.

MH: I think you should do the best you can to prepare for the future but you can't be too detailed. You have to employ a sort of fuzzy logic as you move toward the future. The problem I find with DoD is that they make things very precise before they really have thought through what the problem is and how to attack it.

JS: Also, the people who are building a system are responsible for rebuilding that system. In their view adopting someone else's design and code, given the tools we have now, is a risk. There have been movements in DoD to set up software czars or whatever and services beat it down.
MH: If you're talking about a howitzer it should be clear what it does. If you're talking about software, I find too much precision at this point somewhat objectionable and pointless. Software engineering is evolving and I think what finally registered was that you should give an outline of what is current best practice but allow people to tailor it to the software tastes they have.

HP: I think the DoD and government in general is too much in love with dictating standards and mandates. We won't get reuse of software components across DoD until contractors can sell and advertise those components to other contractors and DoD agencies. The whole idea of government ownership of software is a risk and I think we are seeing that with the new focus. Until you free up the DoD contract community to invest in their own resources and DSDC to address government needs and market to different services, you won't meet these needs.

JMS: The Flexsys system existed and was doing a lot of the things we were talking about — not everything but a significant number of these things. A big problem was that the system was getting hard to maintain. What do you do in that case? What we did was to create a new design for the top based on cognitive analysis. Then, we extracted — isolated all of the key Flexsys components. At the present time, not only are these components interoperable in our system but they are independent of each other and can be called from any other system. Moreover, they can interoperate with any other components or programs that have been semantically wrapped. They are truly independent components. Our final step, of course, was to map these components into the new design. The entire process was greatly facilitated by our semantic tools, some "off the shelf" and others custom.

I'd like to show you some specifics with regard to the technology and then return to the discussion.

One of the things we didn't talk about is slicing code. It might be of some interest to see one kind of custom application we developed to do this. Suppose we want to identify all the code that does something specific. For example, let's find all the statements which use the variable AMOUNT (in a COBOL FLOWform). Our slice variable utility will go through and change the color of all statements using AMOUNT. It also could extract the statements if we wanted to do that. You can see here all the statements are identified.
That's a very simple example of the kind of thing you can do. The utility could easily be refined and/or extended to do various kinds of things — such as extracting the code and putting it into a new FF, etc.

The Flextutor itself can also be extended by simply adding new components. As components improve, and as people develop more and better components — all we have to do is wrap and incorporate the new components. These processes are largely automated so you can easily get access to new facilities. For example ACOL components could be wrapped and made interoperable with an other components. And, of course, another key source is basically to get the stuff ("golden nuggets") from existing applications.

JMS explained that Flextutors are tutorial/demonstration programs built using semantically wrapped components most notably key Flexsys components such as code generation, reverse engineer, etc. Flextutors can be executed interpretively within Flexsys itself. Or, one can automatically generate code, and compile them as separate applications.

The Flextutor demonstrated most of the basic steps involved in Fig. 4: Reverse engineering, semantic wrapping, integration to form a new Domain Specific Development (DSDS) system, use of that DSDS to create a new application (drawing a structure chart corresponding to a call hierarchy FLOWform) and finally executing that new application.

1. LIBRARY SPECIFICATION:

   Select and/or create library components to be wrapped.

   - Up to 90% of legacy code can be wrapped into library components.

   - Third party libraries may be wrapped as well.

   - Create a library specification FLOWform (*.spc) listing all components to be wrapped.

   - Each item in the *.spc FLOWform includes the names and parameters of a to-be-wrapped function with optional parameters and default values.

   - Illustration is based on specifications for routines in a graphics library.

FIGURE 27
2. WRAP COMPONENTS:

- Semantically wrap selected library components.
- Wrapping eliminates implementation details making components interoperable with semantic core.

3. LIBRARY INTEGRATION:

- Select from inventories of wrapped components taken from any number of different libraries and/or reverse engineered legacy code.
- Generate code needed for integration into a Custom .clexsysT. Domain Specific Development System (DSDS) optimized for a given problem domain and user population.
- You may directly compile from .clexsyw- and conveniently view errors directly in FLOWforms.

4. LINK CUSTOM .clexsysm DSDS:

- Link compiled library with the .clexsys semantic shell.
- The result is a customized j."cxsus". DSDS.

5. DEVELOP "REAL WORLD" MODELS:

- Complex, large scale applications in domain can easily be developed using Custom .clexsusTh DSDS.
- Applications created with Custom .Clexsys DSDS may be tested from highest levels of design.
- Designs may be simulated dynamically and checked for logical consistency.
- The sample application displays FLOWforms graphically.
6. TEST CUSTOM APPLICATION:

- During simulation, the Custom...c(exssisni DSDS automatically executes all "known" semantic components.
- Other components are manually simulated; application designs may include both executable and virtual components.

7. GENERATE SEPARATE APPLICATION:

- Once tested via simulation, source code can be automatically generated.
- Compile generated code and link with referenced components. Compiling and editing can be done directly from "Clexsus" DSDS.
- The result is a tested, custom application ready for use.

FIGURE 30

The agenda summarized in slides (Figs. 27-30) was briefly reviewed with a request for comments.

HP: I still have a question about components in terms of the physical distribution of components on the net. Does the wrapping process allow them to be addressed across the network?

JMS: The wrapping process currently deals with C libraries on the same machine. Our approach is designed to work with other standards. We want to semantically wrap them so they interoperate. We cannot support everything immediately. We don't want to compete with OLE components, CORBA components, Army COE, etc. We will select those to support and wrap based largely on needs and opportunities. Insofar as RPC's we're going to make use of existing low level technologies. We're going to wrap them and make them interoperate on a much broader scale. The single most important thing this approach provides is it does not depend on an arbitrary standard. Cognitive wrapping makes components inter-operable not just with each other but with make believe (virtual)
ones, abstract ones that you invent as needed. This allows you to build arbitrary models and make sure all the pieces fit together.

AS: I understand that from a cognitive point of view. But practically, the government puts out a contract to the vendor, specifying what standards need to be followed. How do you implement that higher level of abstraction in practical terms?

JMS: In practical terms, it means you'd have to know what those specs were and your wrapping would have to reflect that specification. If the specs called for ordinary C lib routines, we would probably have that covered with automated wrapping support. Introducing another spec would require additional work. It's a matter of how much is supported automatically. The same general philosophy should work.

FK: Are there requirements on the libraries or can you wrap anything?

JMS: Pretty much any C library. Right now it works with simple types. If you have new or more complex types, one needs to write appropriate type conversions. It doesn't write the type conversion automatically. We have some ideas on how that could be attacked. Whether it would be practical to do that, as opposed to just writing needed conversion routines manually is problematical right now. The point is that these routines are usually very simple and you only have to do it once. The rest is automatic.

MH: The basic problem is still deciding what you want to wrap, and extracting it from its environment.

JMS: There's definitely a small overhead every time you wrap a component. If you're talking about wrapping a C operator (e.g., +, in), there's obviously going to be significant overhead in calculations. If you're talking about something of substance, the extra overhead is no big deal.

JB: If I took these wrapped components which were done on an ANSI C platform, and moved it to a K&R platform, would I have to do anything?

JMS: You can reverse engineer ANSI C into FLOWforms and then generate K&R C.

HP: If you wrapped a library of programs are you doing it also to make it easier to use?

JMS: Absolutely! You look at most libraries, and they're weird and idiosyncratic. Why did they use this as a parameter or why use this at all — that kind of thing? In wrapping, we try to get at the
semantic essence. How should parameters be named to have maximal meaning. The point is you make up whatever semantic and naming scheme you like. Concerns about types are eliminated, having been replaced by semantically meaningful typeless structures.

JB: You say up to 90% of legacy code can be wrapped. Is that 90% based on your experience? I find your number intriguing.

JMS: In our code, we reused 99% of the legacy code. We didn't throw much code away at all. What we did was reorganize it in a fundamentally better way. It depends on the code, of course, and how much you think is worth salvaging.

There are a lot of exciting possibilities — limited only by your imagination. What fascinates me more than anything is that it's now possible for the first time to do things I've always wanted to do. To be able to develop software at a high level of abstraction by describing the real world. Cognitive methodology gets away from all the low level "junky stuff." On the other hand, I don't want to underestimate the importance of and the work involved in making legacy code interoperable. Our automated tools help but there is no panacea.

MH: You can't get away from human judgment involved in isolating golden nuggets and wrapping them for future use. You have to examine what you have extracted and decide if it is still functionally as good as well as computationally secure. I went to a conference of multiprocessing and they were talking about getting different machines to work together. To me, the most significant kind of multiprocessing couples machines with humans. You must decide how to divide the work.

JMS: Right! We're talking about possibilities for automation. Human judgment is essential.

HP: To automate, one of the things you can do is try to capture as much as possible about human judgment and make it available to others — instead of just throwing up our hands and saying get good people. To what extent can we capture understanding and convey that to people?

CM: How do you determine good judgment from higher level judgment?

JMS: The top of this slide characterizes the standard approach to software services. You take on projects and you do them individually. Each dot represents a project. You may use tools or not.
The fundamental problem, whether tools are used or not is that each project has its own schedules and priorities. What one does or learns on one project has little impact on other projects. One company is trying to improve on this model. They will have a bunch of tool experts who know how to use the tools sitting in a laboratory. The idea is to have these experts serve as a resource to personnel responsible for the project code.

The problem we find with this approach is that solving any given real world service problem requires that you know what the problem is and what the tools can do. Only then can you map the two together. Often the tool doesn’t do what needs to be done, but if you don’t know the tool and the problem intimately there’s no way you’ll find a solution based on the tool to solve the problem.

The approach we take is as follows. This is what can be automated now. When you add a new project, we don’t just devise a new solution to that one problem, but we generalize the tool set. You now have a solution to a broader universe. Anything that falls inside (the current scope of automation), represents a high profit margin due to automation. Essentially, not only are you completing projects using this amoeba-like approach but you are expanding the scope of what can be automated in the future.
How many of you would view what we've been talking about as a natural evolution of current thinking and how many would view it as a different paradigm?

(most hands for natural evolution)

JB: What I'd like to see is how your set of tools would play with other tools to see how the stuff might be brought together.

SWP: I think it wraps at least several old paradigms and, as you've said, I think the potential has barely begun to be scratched - particularly taking things to a higher level of abstraction and going on from there. I think traditional CASE tools are a different paradigm. They don't go to that extra higher level of abstraction.

JMS: It's interesting to consider who wrote most of the old structured analysis textbooks. Most were very different from people actually in software engineering. They are technical reporters. They rarely invent anything themselves but they are very good at describing things and getting it across to a mass audience.

MH: Some are more than reporters. They are salesmen.

CA: One thing these people do that's critical is to decide what to report about.

JMS: But they can retard progress by missing essentials. For example, in one book in talking about constructing data flow diagrams, the author points out that most people start in the middle. Consequently, he elaborates on quasi-methods for abstracting (upward as well as refining. To be sure most people start in the middle because they start out using the terminology most familiar to people in that domain. Those terms in general will be somewhere in between the most abstract and most concrete. But, having started there, at an intermediate state, there's no hope really of getting systematic correctness by design.

If you take clean room engineering as an example, then my argument would be a little different. In clean room the goals are much the same as ours. I think the failure in that approach is that they need to be too precise too soon. They need to know about everything that's coming in and out from the beginning. If you have all that information and systematically refine the process, it will work. But that approach just won't scale. What's different in cognitive analysis is that data and process are refined in parallel.
MH: I think it's the way you look at problems. You've described a
different way of looking at the universe we're dealing with. There
are different approaches and your approach is new, at least for
software engineers.
JMS: This is true even in artificial intelligence. AI has been charac-
terized-as building a bridge across a river on a foggy day where you
can't see the other side. It deals with the unanticipated in an
idiosyncratic unstructured way. In building expert systems, you try
to codify knowledge as a list of productions. Unfortunately, if you
change the order of those productions even a little, you typically
change what the productions can generate. Consequently, you can't
just go out and get some experts and codify their knowledge. What
you get may depend on the order you ask the questions.

In structural or cognitive analysis you start with a domain and ask how
you can characterize the knowledge necessary for success in that domain.
It's very easy to come up with domains where it's not possible to find a
single algorithmic solution — the domain of mathematical proofs, for
example. What you're talking about there is a domain where there are
going to be subdomains for which you do have solutions. Then the
question is how do you fill the gaps? How do you allow for the unanti-
cipated? Structural analysis takes the subdomains, builds solutions
for them, and then looks for relationships between the solutions. These
can be characterized as higher order solutions, ways of generating
new solutions (from old ones) for unanticipated problems. The major
difference between structural (cognitive) analysis (SA) and the tradi-
tional AI approach is SA offers a systematic approach to dealing with
the unanticipated.

Let's turn to some general questions. I'd like each of you to comment
on them one at a time:

1) *What is the single most important idea we've discussed?*
HP: Dealing with systems at the component level.
AC: Conversion between different languages, for example, C to Ada
right now, especially in the DoD community.
CA: Flexibility and Adaptability.
ABS: The cognitive approach generally. Code is becoming more
complex and is getting beyond people's cognitive capacity to deal
with it. It's necessary to have an approach that addresses these basic
cognitive limits. Cognitive methodology deals with this problem by
offering high level abstractions both for pure abstractions and abstractions linked to actual code.

GF: A very important issue is generating exposure to this kind of capability.

CM: The idea that perhaps we're working in an environment in which we don't yet know all the laws. We're still searching for the underpinnings, the laws regarding what we're trying to do.

JL: The evolution of development methodologies is at the point where we're no longer throwing away everything we have. We now have a methodology that allows us to take what we build today and start from that and begin to build systems from components so we don't have to sacrifice all the legacies.

CS: You can represent a program in a hierarchy and in doing so you can more easily replace parts. Part of the debate is whether you should represent a program as a hierarchy or as a flat source file.

2) How might you take advantage of that idea?

MH: Personally, I'm going to retire in a few months and by encouraging this work I've ended my active job related career on a high note.

HP: From my perspective I'm looking at a number of opportunities in which we'll try to take the CIM initiatives and make sense of them from an architectural perspective. That is, I want to figure out a way to get those existing systems into that architecture.

FK: One might be as a user. It would be great to have a tool to help me recover legacy code. Also, as a researcher, it gave me a confirmation that some of the kinds of issues we've been working on are going in the right direction.

JB: I think there are going to be some great opportunities. I'm talking about 60 million COBOL lines of code waiting to be translated. My objective is to help my clients make money and to meet user needs in the USA.

AC: In the program I work on we have components that run on different platforms using different OS's. I'd like to see it we can wrap and let them work together or move them to a single platform or a single language. Right now, we're trying to make them interoperable.

CA: I think the best way to take advantage is to put it to maximum use; get a lot done and minimize costs.
CM: The best way to take advantage of the method is to use it to better understand our own systems and the systems that we're going to have to interface with. There are a lot of unknowns on both sides. We have to get our own house in order and I can see this may be a methodology to do that.

SWP: I'm not sure that this is the methodology we'll use, but it certainly is a methodology that might help.

AS: Regarding the paradigm shift, dealing with designing systems from the top that are correct by design from a higher level of abstraction: How might I take advantage? I don't know.

3) Key question not answered?

HP: The whole issue of runtime, the execution environment. Actually, applying the architecture on some new set of hardware. Given a network and a bunch of servers, how would I implement this on that environment? How do I go about deploying a solution in a real environment?

JMS: We didn't talk much specifically about that. People clearly are and have developed mechanisms for RPC, etc. across networks. That work is ongoing. It will continue to happen. My feeling is that that work is going to be done by big players. We're going to take advantage of it. Our abstraction layers will be above those. Just like they would be over OLE., CORBA and ACOE.

FK: I think it's also important to have some methodology to integrate non-technical constraints into your system or non-software technical constraints; aspects of the hardware, or aspects of the business environment.

JMS: Embedded systems is one area of concern. We think of hardware components as jagged chunks of capability. They are not semantically wrapped. However, software can be used to round them off. Wrapping can be used to make them interoperable just like anything else. What you have in this case is a combination of software and hardware that is interoperable with the rest of your system. Ideally, what you want is to use low level software to round the edges (of the hardware), to form components that have semantic meaning.

TB: The ability to work with other tools and to meet the standards that are out there.

AC: Much of what we discussed has a lot to do with requirements. When you do reengineering you're satisfying new requirements.