BEHAVIOUR

McLeod: In this session on behaviour I would like to raise, at least the following issues:

- What is the distinction between data and process?
- How are process and data related?
- How is behaviour, or system dynamics, modelled and specified?
- What is the impact of behavioural specifications on system evolvability?

Rick Cattell, Peter Buneman and I will start by commenting on these issues with respect to database, AI, programming languages. Rich Cattell will attempt to relate evolvability to behaviour and discuss behaviour modelling in terms of different forms of invocation. Peter Buneman will talk about how we model processes in programming languages. I am going to talk about tradeoffs between modelling things as data or as process.

Lundberg: What do you mean by process when you say that we will discuss how a process is related to data? Is a process a transition between states?

McLeod: That's an important aspect. Another aspect concerns declarative versus procedural specifications. Do you have a set of primitives that relate objects, or do you have a general purpose procedural language for describing them? There's a continuum. You might put database query languages somewhere in the middle, and the general purpose programming languages on the highly procedural oriented end of the continuum.

PROCEDURAL INFORMATION

Cattell: I interpret behaviour as procedural information and intend to address how it is emphasized in the different areas, how it is modelled or abstracted, how it is invoked and how it relates to evolvability.

Relative Performance

One major difference in the viewpoints of database versus programming languages people is that in programming languages they're primarily concerned with maintaining large systems of programs and less interested in long term data while in database systems data is a valuable, persistent (long-term) resource. It might be hard for a programming language person to change point of view. It seems that in AI everything is throw-away since there are seldom hundreds of users. For AI systems, user groups are small enough you can afford to throw it away and write a new program.

Rich: There is a trend in AI and programming languages (particularly in LISP- and Smalltalk-like environments) where your fix-it procedure is laid out in database fashion. That is your priority resource. The whole thing is evolving. That sort of brings the two points of view together.

Cattell: That's a good point. These things are being put into one environment to support development and evolution. So far this is being done only in small projects; (less than 10 people). In such environments it's not a major problem to separate data from process in such a way that you won't have to rewrite all your programs when you change representations of data or vice versa.

Mark: I don't think you ought to get esoteric. My project is certainly an AI project, but it's very much the way you characterized a database project. We are really concerned with a very large chunk of long term data.

Cattell: Well aren't you saying you have a database project instead of an AI project?

King: AI really does include long term data and long term procedures. Look at Mycin. In one direction physicians are building up a set of rules. The evolution is a little different from standard AI in that the data is a bit more valuable. People talk about their rules and heuristics independently of the program. On the process side, the inference engine has been taken out and used with different rule bases. So both data and process are being developed and kept, not thrown away.

Distinctions Between Process and Data

Cattell: In a lot of AI systems the distinction between data and process is not blurred.

Brodie: I suspect that in AI, as in databases, the distinction may have been clear some time ago but that the distinction is becoming less and less clear. In the past, database design dealt almost exclusively with data; however, it was found that the resulting schemas were not adequate for describing all constraints, particularly state transitions. As a result, database researchers began to consider data types, that is, structures together with their operations. Now, in my work, I am concerned with the design of process (the behaviour properties of database applications) just as much as with the design of data (structural properties).

Your characterization of database, programming language and AI concern for data and process is, I think, generally true but rapidly changing.

Christian: I suspect that the manner of structuring behaviour in databases is to put operations

together around the data types. I think that data and behaviour are so interrelated that it is pure abstraction to make them disjoint.

Brodie: I agree. In databases, there has been a lot of work done from the data point of view. Now, I and others are looking at the behavioural properties of databases. This naturally comes around to data abstraction, which can be used to integrate the two. However, you want to be able to change data structure--both its abstract logical properties and its representation--without affecting process and vice versa.

Let me clarify that a little. There is an abstract notion of data, which I call structure, just as there is an abstract notion of behaviour. The abstract properties which are given in structure and operation specifications are distinct from representation. In programming language research what I call structure is not distinguished from representation but in database research it is, e.g., the distinction between internal and conceptual schemas. For example, it seems natural to view an employee as an entity with certain data or information properties (name, age, salary, etc.) rather than the result of a sequence of operations promote (promote (hire (emp), trucker_2) trucker_1). In programming languages structure is treated as representation and as such it is not abstract. These structural abstractions are at the heart of database design.

Balzer: People are talking about separating concerns, such as data from process, but more generally these systems are going to evolve, and we want mechanisms to help protect assumptions that get built up on various parts of the system. The data-process distinction is only one bifurcation; there are lots of others.

Lundberg: One point missed is that you have to consider the evolution of the content of the database. How to model the evolution of the information content of the database. We have just been discussing the evolution of structure and behaviour but not the information content.

Procedure Invocation and Demons

Cattell: I can think of four ways in which procedures get invoked. In one way, which I call views, a procedure gets invoked in what, to the user, looks like fetching and storing information. That mechanism is used in both the database and AI worlds and little bit in the programming world. The converse of that is demons, which are sort of pull instead of push. They sit there and wait until something happens, and then wake up, and do their thing.

Balzer: Are you saying that the difference between views and demons is that a filter is inside a view, whereas it's external in a demon?

Cattell: What does filter inside of a view mean?

Balzer: The view is interested in only some of the things that are going on with related data.

Cattell: Yeah, if there's any filter at all, it would be inside the view.

Buneman: Surely that's an implementation issue. A demon is simply a rather complicated view.

Deutsch: A simple way of characterizing the distinction is that demons are triggered by conditions occurring in the data which are defined by predicates, whereas what Rick is calling views are triggered by events. A view is triggered by an event, which is something in the process world, whereas a demon is triggered by a condition in the data world.

Hayes: I don't understand the distinction between them. Surely a change in truth value is of importance.

Rowe: A demon is a process, something active; an agent sitting there waiting for things to happen. A view is somewhat passive until it is called. If you say that activity invokes a demon, then it sounds very similar to a view.

Rich: A demon is more general. It acts on more than fetch and store.

Cattell: Let me give you an example of a view. Consider the employee relation EMPLOYEE (Name, Date-of-Birth,...) and EMPVIEW (Name, Age) which filters and throws away the other attributes of employee. It also does a computation, and to convert date of birth into age, but only when you fetch or store. It doesn't sit there waiting for the current date to change.

Rich: Isn't that called a derived relation?

Buneman: This is again a matter of implementation. You've presented one way of implementing views. Another way of implementing views is to have a set of virtual relations which are evaluated when you access them.

Hitchcock: There is a distinction between a derived relation and a copy. In one you see recent changes and in the other you don't.

Balzer: I think I misunderstood you originally. Perhaps the distinction is that a view is invoked and updates itself based on the current state of the world only when the user invokes it, whereas the demon is involved externally by some predicate becoming true.

Hayes: That's a very important point. The importance of demons in AI is precisely that they do things that weren't specifically done by the user. A demon could fire a condition then another demon fires, and that can go on in a chain of computation not foreseen by the user.

Cattell: Right, these are definitely different from the users point of view. An example of a demon in a speech understanding system is one that is invoked on finding three lexemes on which it does some action like conjecture that some word was uttered. The flow of control is a lot more like a procedure call than like a concurrent process sending messages.

Hayes: Am I right in thinking that a view has to be called by its name being mentioned in a piece of text written by the user?

Cattell: Yes.

Hayes: OK then an implicit procedure call is not a view mechanism.

Sibley: Databases were demon-oriented in 1969.

Hendrix: Can we look at views as being some kind of special translation procedures that happens in the I/O channels, (You're either putting something in or pulling something out.) and that demons are internal to the system. Hence views get used to transform from one language or one set of predicates to another? (Cattell) Yes.

Explicit Process Invocation

Cattell: There are two other ways of invoking processes which are more explicit than views. (Views are implicit in the sense that the user didn't know whether the thing invoked was a view or whether it was stored.)

The third and fourth types of process invocation are applications of built-in operators and procedure calls respectively. Built-in operators are distinguished from procedure calls primarily because the set of built-in operators are known to language implementor and can, therefore, have an optimizer designed to exploit their known algebraic properties of combination. This point is not terribly critical, however.

The fourth kind of invocation operation is a procedure call which in an abstract data type scheme would be separated according to type on which they operate. Procedure calls are used in programming languages and AI. In database the operations are generally independent of types.

Pattern Directed Invocation

Deutsch: You might want to mention in that connection what's called pattern directed invocation. It is like a procedure call but may involve a certain amount of inference or searching of the database to find a concrete procedure corresponding to a pattern.

Rich: Are you talking about demons, Peter?

Deutsch: No. I think that that belongs then under what you call procedural. However, you can't put these things in separate categories anyway. There's a continuum and you may as well put it down there under procedures.

Balzer: The only distinction is that the name of the procedure is not known. You just give a description of what you want, but you know you are doing a procedure call somehow.

Carbonell: Suppose you have a language with an internal evaluator, like LISP or APL. You can create an expression on the fly and then execute it. Now which one of these categories does it fit into? Does compile time versus run time create any problem?

Cattell: Yeah, I think that's what we're talking about.

Call by Description

Rich: Rick, David McAllester has a way of talking about these different types of pattern directed invocation and explicit calls. He talks about call by name, which is obviously exactly the name, and call by description. You don't have the name of a procedure but you have some description of it. With that you get into all different kinds of how much of a description, what the form of it is. It is assumed that there is some database and some process that can match your description against the descriptions of all the available procedures.

Cattell: Is it called by description when I index into an array of procedures?

Rich: Yes, that's a trivial kind. If you say get me the procedure which operates on the color yellow, or operates on the color red, it's definitely a call by description. However, the indexing thing is an implementation issue.

Hendrix: In the discussion on types, types were fairly closely tied to the notion of description or a set of conditions. Call by pattern, call by name, call by any one of these things where there is a description or constraint that is automatically resolved, involves automatic type resolution on the procedures that are to be called.

Deutsch: You can also think of it as a generic procedure mechanism.

Data Flow Invocation

Balzer: One type of procedure mechanism we haven't talked about is the notion of dataflow languages in which a procedure is invoked when the data arrives.

Cattell: That's a simplified form of demon where the demon is only allowed to wait on a simple type of input.

Hayes: Your definition of demon is so general that it encompasses just about any form of invocation mechanism other than the explicit ones.

Cattell: Well let me call a demon more specifically something that has a boolean expression that's evaluated continuously.

Hayes: I was going to suggest an alternative, which may be the same as dataflow and which is used in PROLOG, that is, invocation when a binding is made to a variable. That results in a procedure which refers to that procedure being invoked to pay some attention to the new binding. It has the same flavor as dataflow, but it's in the context of logic programming.

Message Sending

Rowe: You haven't mentioned sending messages to a process to cause some operation to be invoked, either in the Smalltalk sense or when you write demon, nor of demon processes which have nothing to do with predicates.

Deutsch: Rick isn't concerning himself with whether the thing being invoked executes concurrently or sequentially, or exactly how it's made. In that regard, the right way to think of the Smalltalk discrimination mechanism is simply as a generic procedure mechanism.

Cattell: A Smalltalk message is a bad name for what they really are.

Deutsch: Yes, it is a bad name.

Now Peter Buneman will discuss programming as behaviour modelling.

PROGRAMMING IN THE END USER INTERFACE

Buneman: I confess to being somewhat confused by the terms behaviour and process. There are some questions that I would like answered. The fist question is "what kind of programming languages are needed for database practitioners" practitioners are people called designers, application programmers, end users, and administrators. I want to talk about end users quite a bit. There is incredible amount of stupidity in designing simpler and simpler languages that turn out often to be less and less what users want or need.

The next question is, "Is there something special about database design and use that distinguishes it from other forms of programming?" The problem I see here is with the evolution of a database; this is what I properly consider behaviour. The problem arises because one person designs a database in terms of a set of types then somebody else uses them. Should these two people necessarily use even remotely related languages to talk about the database?

Hitchcock: Well, there's not just one type of database user. There's others in the future you don't know about.

Buneman: I would actually like to blur these distinctions between types of users, designers, etc. A person should both be able to design a database and to write applications programs if he feels like it.

Buneman: The last question is directed to programming language people, "Can we obtain the benefits of a 'type discipline' in an interactive programming system?" I happen to like nice programming languages like LISP and APL. They lack the sort of type discipline I'd like to see. Why do I think that's important to databases? Well again, I want to blur the distinction that somebody creates a database (which as I said is a set of data types) and fills it up, and somebody else wants to extend it. Can extension be done on the fly? Can it be done directly, or do I have to go back and recompile the whole program, which is the current state of the art. In System R there's a certain amount of extension possible. System R is a relational database management system built by IBM which allows interactive creation and modification of the structure of relations.

Balzer: With regard to interactive extension, the language EL1 or ECL, at Harvard, is strongly typed and is highly interactive in the LISP style.

Buneman: There's also Robin Milne's language LCF, which has some of that capability.

Database Administration

Sibley: I must take exception to your view that database design and use should be blurred. It's a good idea for only a small percentage, less than 5 percent, of the normal user community of database systems. Merging these groups generally means a loss of control of database integrity and security. The whole purpose of the initial introduction of the concept of the database was to enable administrators to control the data.

Buneman: There's something in what you say, which bears out my initial contention that we hold database end users in very low esteem. I've seen many cases of database end users who wanted something much more powerful than they get, and could use it. For example, these users are using APL to write data processing programs not provided with the database system.

Hendrix: Well they are essentially using data for their own uses. They aren't sharing it with anyone else.

Buneman: That's all right. But this kind of usage will not preserve constraints defined across collections of data.

Sibley: Now that's 99 percent of databases.

Rowe: I've heard these concerns from computer center directors who run large databases. I've also heard them talking about this large number of people who are asking them to give for interactive database facilities for their own purposes.

Sibley: Sure, but these are toy systems.

Rowe: Those needs are now being met by creating separate, as you call them toy, systems for small amounts of data. In the future those people are going to discover "gee, here is this neat tool I can do all these marvelous things with, but I don't have access to the real data." They're going to demand interaction and access to the real data. The real organizational and technical challenge is going to be now to provide the protection for those users.

Deutsch: There is a tremendous difference between the responsibility of a database administrator to decide what constitutes a condition necessary for the integrity of the database, and the activity of the database administrator to basically define the roles of the people who use it. In one fairly large commercial system I'm familiar with there's a continual flow of demands from the users for new

applications. This creates a tremendous demand on the company itself to write these application programs precisely because that system, like most other systems doesn't distinguish the integrity mechanisms from the programming mechanisms. It seems to me that there's a tremendous amount to be gained by giving the end users capabilities to do their own applications programming within the integrity constraints defined by the database administrator.

Type Generators

Buneman: I want to consider primitives for defining data types, and I want to draw a parallel with databases. One of the least understood things in data types is the notion of type generators which are also called parameterized or generic types. My view of the world is that a database management system, or shall we call it a data model (depending on whether we are talking about the implementation or the abstract model) is very much a set of type generators. For example, a relational database might have type generators that construct domains and that construct relations out of domains. A CODAYSL system essentially gives a set of type generators like area and set-of.

A database schema results from an instantiation of a type generator. The database designer makes a set of type generators and creates a set of types. People who design database schemas are really instantiating types, people who use databases are examining the objects. Do these people need the same kind of programming language? These people are exporting type downward. This is a distinct contrast to a programming language for writing a compiler. If I'm using your operating system or Pascal compiler, I don't care tuppence about what data types you used in building your Pascal compiler. To summarize this issue: along with a data model are a set of generators or functionals which are used to create new instances of types.

Query Languages

Let me just take a couple of cheap shots at end user query languages. Consider adding two and two in your favorite language. 2 + 2 is of course APL, (plus 2 2) is LISP, "TWO PLUS TWO" is at least one macro language system. Doing this using Pascal (in our environment) involves

```
MAKE TEST.PAS

IPROGRAM TEST (OUTPUT):

BEGIN

    WRITELN (2 + 2)

END.

$EX$$

EXTEST
```

In SQL it is

```
SELECT UNIQUE 2 + 2 FROM EMPLOYEE
```

which you can do if you happen to have an employee around. "2 + 2 FROM EMPLOYEE" is really pretty poor. The end user will not accept the awkward ways in which arithmetic is done in certain database query languages. The point is that he needs a more complete programming environment to be able to process his data.

Well let's consider a database query to get employee names. SQL does very nicely. SELECT NAME FROM EMPLOYEE. Now consider the solution in the LISP environment. The natural approach is to use the functional, "mapcar", assuming that there is a list of EMPLOYEES.

```
(mapcar (selection function) EMPLOYEE)
```

The original query was set based, however, and this approach returns a list. One attempt to fix this problem would be to add a uniqueness qualifier to the operation. This, however, will not solve the real underlying problem that the data is not a list (in core as LISP understands it) but is instead a large collection of records on a disk somewhere. Hence, it is necessary to rewrite the query in the following form (not my favorite introductory LISP program):

```
(PROG NIL

    (FINDFIRST EMPLOYEE)

    (COND ((EQ ERRSTAT XYZ) RETURN))

  TAG

    (GET)

    (PRINT NAME)

    (FIND NEXT EMPLOYEE

    (COND ((EQ ERRSTAT pqr) RETURN)

    (GO TAG))
```

Finally, here is the query in PASCAL/R

```
(type declaration)

Var EF:  EMPLOYEE Type

      :

Begin

    Foreach E in EF

        WRITELN (E.NAME)

END
```

Lazy Evaluation

Buneman: Rather than use this iterative program, we would like to view the data base as a sequence to which functionals, such as mapcar or selection, can be applied. This can be implemented by delaying the realization of the output list until elements are actually used. This approach is normally called "lazy evaluation." Although lazy

evaluation is generally up to an order of magnitude slower than normal evaluation, some of this time can be hidden by the time spent waiting for I/Os in the database system.

Deutsch: Friedman and Wise at Indiana State University have done a lot with lazy evaluation, especially with respect to CONS.

Buneman: We handle lazy evaluation quite differently.

Balzer: It would be perfectly reasonable to have, let me say, a generic mapcar which, based on the type of its left argument, decided whether it was a disk list or an in core one, and did the appropriate thing, but you don't need lazy evaluation to do that. You can get by with much simpler mechanisms.

A Simple Database Query Language

Buneman: We've been constructing a language in which you can talk about types in a fashion completely independent of all objects that might exist in the system. It has four functions; composition of functions; extension, which is the same as mapcar; restriction, filtering something on a predicate; and construction, which takes a set of functions and creates a tuple. This specifies a very simple database model. It happens to be rather nice in that you can map CODASYL and things like that into it. It turns out to be a programming language at least as powerful as pure LISP. It does account for one of the problems I have mentioned, the end user is not necessarily limited to rather feeble query languages. Now you don't have to write these awful programs. It's being used now in several places for a natural language interface. One question is how far can it be extended? Implementation and efficiency are of course standard problems.

The next thing is incremental definitions of types. One can argue whether this is desirable or not. Can the end user having been given this world in which the types are now all defined, implement and define new types, e.g., a special set of people to be dealt with from day to day whose CODASYL numbers you have and for whom you must add integrity constraints. To what extent can this be done without recourse to recompiling the whole database? To what extent can we perform incremental type checking and verification? I hate the words compile and runtime, but I much prefer to do is at runtime.

The next problem is the definition of functionals and operators. Now this is a general problem. Take for example, a standard database bill of materials processor. A standard recursive problem. It's so complicated in COBOL for FORTRAN that you'll go and buy a special purpose application. If you have a nice Pascal interface, it would be maybe two to three pages. You would say, "gee that's two or three pages of code, that's getting along. The user is likely to make a mistake. Should we organize it and have a well structured program?" On the other hand, you could write it in a line using a LISP program. All you need to do is compute the transitive closure. Transitive closure is in fact a functional, or relation, and produces a new function.

The final problem is to what extent can we convey in these languages, information about types? Well these are questions that I want to pose for the programming language people more than for the database.

McLeod: Thank you Peter. Are there any questions?

Codd: Would you sum up in a few words the superiority of this approach over, say, PASCAL/R, which addresses the same problem? Put aside all the remarks about end users, which are unnecessary actually, for your whole presentation. There might be a place for what you are doing even if everything you said about end users is false, OK?

Buneman: Pascal/R embeds the relational model in Pascal, and supports relational operators. What I would like to do, even in Pascal/R, is to add some of my own new operators. For example, transitive closure.

Codd: I see. In other words, user defined extension to . . .

Buneman: To the types. A second point is that I find adding two plus two, or getting the names of employees cumbersome even in Pascal/R.

Brodie: Both of those things are particularly simple in Pascal/R.

Buneman: Well, I'm conscious that a type declaration is needed.

Hayes: That's a point about the superiority of functional programming to Pascal, period. It has nothing to do with database.

Deutsch: Functional programming and type declaration don't have anything to do with each other. That is, they are orthogonal.

Buneman: Yes, but I hope you're going to see something which incorporates both.

Process and Data

McLeod: I'm going to comment on what I'm calling perspectives on process: tradeoffs between modelling things as data and process. I'm going to do this in the context of an example concerning the type persons, the type homes, and the operation or act of purchasing a home, which involves an object of type person and an object of type home.

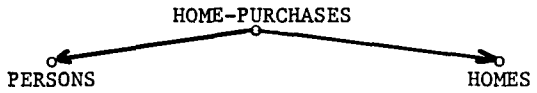
Abstract data types would be a formalism for this kind of problem, but abstract data types require each operation to be associated with a particular type. Here is a very simple example of an operation which really involves two types. It takes as parameters objects from more than one type. So there are really two points I want to

make. One was simply this is an obvious way of modelling the operation, and the second thing is that some formalisms, such as abstract data types, can go part of the way in accommodating this, but have some limitations.

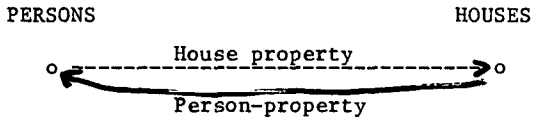
Aggregation versus Attributes

Brodie: What is the association between person and the home? Depending on other characteristics of your application, you could have a data abstraction which is the aggregation relationship between the types.

McLeod: Yes, this is the database approach. You could introduce another type, which I call home-purchases, the instances of which are records of home purchase events.



As Michael indicated this would be viewed as a relationship between person and home. Another way of modelling that same thing is to use a data model with the concepts of object and attributes. You could say, that House is an attribute of objects of type persons.



Similarly we could define person as an attribute of house or we could define both of them. As Ted Codd pointed out and John and Diane Smith have called a lot of attention to, there are different ways of viewing the relationship between persons and homes.

Hitchcock: The semantics of your two examples are different. They don't carry the same information. In the home purchase model, I can distinguish individual purchases.

McLeod: I think what you're getting at is that a person may own more than one home and vice versa.

Deutsch: There's nothing in those diagrams that commits a person to having only one house actively associated with it.

Levesque: The distinction in the semantics can be seen by considering the case where the same person purchases the same home more than once?

Hitchcock: He bought the home, he sold it and then bought it again. We can identify them separately as home purchases because the identifier for each purchase is different, whereas if we are identifying it just by the pair (person, home) we can't distinguish the purchases.

Operations versus Data Objects

McLeod: There's a tradeoff between modelling a fact as either a data object such as home-purchase or as an operation. They guy who records the fact "home is being purchased" may want to say "I want to invoke an operation," and give it a couple of parameters. Someone else can say "what are all the instances of home-purchase that happened over the year?" The tax board may want to view them as entities. There's a degree of relativism involving operations.

Hitchcock: Another aspect is if you want to know how much he paid. It's very easy using the home-purchases type to tack it on.

McLeod: There are many considerations which would say that one of them is better than the other.

Balzer: Well it seems to me that if you use the home-purchase model, you are not recognizing events as objects. You're creating a new type which is divorced from the event.

Hayes: You're right. There's a generic-specific distinction which is closer to the home-purchase model than anything else. This is well-known in AI as well. I don't understand how the purchase-home operation is an alternative to the home purchases type, because surely it's the result of that manipulation operation which records the home-purchase fact.

McLeod: Some people may want to think about it as an operation and others as an entity.

Hayes: What's the result of the purchase-home operation? Presumably it leaves something around that records that the home has been purchased.

McLeod: Yes, the effect of a purchase-home operation may be to create an instance of the type home-purchase.

Hayes: I thought you were putting them forward as alternatives.

Weber: You need them both. They are not alternatives. One is the action the other the result.

McLeod: The programming language, database, and AI communities have different approaches. Programming language people tend to model things as operations, and database people tend to model things as in the attributes or home purchase examples.

Weber: Dennis you didn't get the point. You need to have both the operations and one of your database examples. One describes the action which leads to something you have to record in the database.

McLeod: Tell me a database management system that lets me look at it as an operation, and you look at it as a pair of attributes, and Rick looks at it as a home-purchase instance.

Rich: Aren't transactions in database models?

Deutsch: Yeah, but almost no database models allow you to talk about the transactions themselves as pieces of data.

McLeod: They're unrelated to the structure; they're outside it. That's one of the problems I'm trying to point out. There are tradeoffs between modelling things as process or as data. People choose different tradeoffs which are not well understood.

King: Another way to model your example is to have multiple records coming out of a particular event. You can think of the transaction as an interchange, and one person gets money and the other person gets property. But there may be preconditions on that action that you want to reason about, which limit particular values of the recorded events. In order to understand what those are you have to go back to look at the action.

Weber: What we do right now is we put the transaction into the database management system and not into the model. The alternative is to put the description of the event into the data model. But there is no alternative where one has either the operation or the result. We need to have them both. The question is how do we do that?

Data Dictionaries

Sibley: If you've got a data dictionary you can put the fact of the purchase-home operator in the dictionary and say that it does effect the persons, homes, and home-purchase records, and you record who is allowed to initiate the purchase-home operator.

Hayes: Yes, but the point is that having anything in that dictionary certainly it didn't affect the house, or owner records, or whatever. It just records that this operation has taken place but there was no actual consequence or trace of the operation in the database, that would be ridiculous, right? The fact that some internal piece of machinery has been invoked, and has run, and has done its thing, of itself is not an alternative way of recording a piece of information. That internal process has to result in some recording of the information that it was supposed to record.

McLeod: I think there is some confusion as to what is meant by the term data dictionary.

Sibley: They're mechanisms for control and description of the database, the processes that operate on it, human roles, and also the communication systems. The typical one has something like, say, 120 or 130 records which represent the statements about possible states of the system. So, for example, you can state that you have an operation called purchase-home, that can or cannot access persons, homes, and home purchases. You can't implement the process called purchase-home unless you've made that statement prior to the start of the system. You are then allowed to access persons, homes, and home purchases. So it's a control mechanism in that sense. Equally, it's a repository of information so that you can ask, later on, if I do something to persons records, make them bigger or smaller, add information to it, whatever, what programs can be affected. You usually call it a dictionary because it contains the standard definitions.

Hayes: That doesn't seem to have anything to do with this at all.

Codd: I only wanted to comment that the separation of present dictionary products from the database management system is an historical accident. It turns out that dictionaries have come about in many cases because the database management systems weren't adequate tools. I think that newer systems will incorporate dictionary capabilities within the database management system.

McLeod: The data dictionary as Ed has defined it, is the traditional bridge between the procedures that manipulate the database, which are external application programs, and the database conceptual schema. It contains the external procedures that accomplish a kind of indirect link. The trend, as Ted points out, is that the data dictionary is migrating into the conceptual schema.

Viewing Processes as Data

Feather: The act of purchasing a home is an observable event. We can ask about it later on. You can determine ownership by asking who purchased this home. It's all a matter of what you can observe. So the purchase-home operation (model) is an alternative to the database models.

Meyer: In programming languages there are several examples of systems which you look at a module both as a process and as an encapsulation of data. The best example may be the Simula 67 class, which is both the implementation of an abstract data type and may also be co-routine, that is, a process. Other examples are CSP and also tasks in Ada, which are both modules and process.

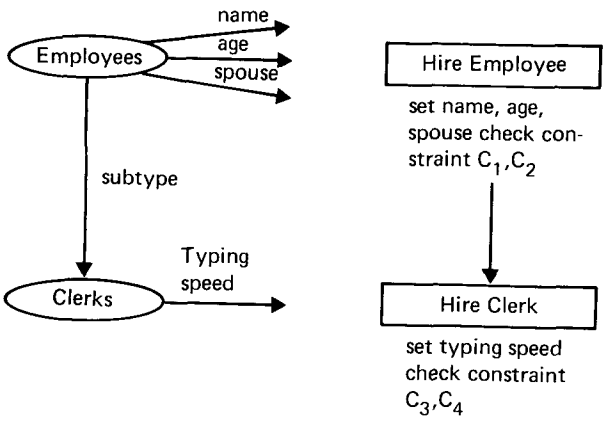
Deutsch: Modula and Mesa have the same property.

Relationships Among Operations

McLeod: One can also talk about structural interconnections among operations. A problem with most approaches is that operations are relatively disjoint. There's some attempt to relate them using the constraints that are involved; that is, eliminating the notion that an operation is not decomposable, and breaking it down into preconditions, postconditions, and actions.

To see this we draw an analogy with subtypes. Suppose we have a type "employees," and we know some things about employees like their name, their age, and spouse. We can have a subtype of employees, say clerks, and we know since all clerks are employees this is an is-a connection. What we know about clerks is their name, age, spouse, and we also know their typing speed.

In terms of operations, suppose we have an operation which hires employees, called hire-employee. It creates a new instance of type employee, it sets his or her name, age and spouse, and checks a bunch of constraints. We might have a more specific operation like hire-clerk, which is a special case of hire-employee. It does all the things that hire-employee does. It accepts the name, age, spouse, checks some constraints, but in addition it sets the typing speed, and checks an additional constraint.



ORGANIZATION OF DATA AND OPERATIONS

Hayes: That seems the same as hire-clerk, simply calling hire-employee as a subroutine and then doing something else as well?

Rich: Another kind of relationship between two operations is "is an extension of." For example, one standard LISP procedure scans a list for the first element of the list which satisfies a predicate that is an argument of the procedure. Another procedure which operated the same way and also returned the test element immediately preceding the one satisfying the predicate would be an extension of the first procedure. It inherits the first proceedings preconditions and the inherited post-condition would be modified to reflect the extra result.

Carbonell: Instead of just adding things to do, you could have a more stringent constraint to hire-clerk then to hire-employee. Instead of C2, you can introduce C5, such that C2 is a subconstraint of C5.

McLeod: Other analogies can be drawn from ways you can interconnect types, say, data, to ways you can interconnect operations. An important point here is what types of structural interconnection primitives will be provided for operations. The primitives you need are determined by what applications you had in mind for the system. What you're going to do with it. A number of research projects including TAXIS have made some progress in that area.

Rich: It might also provide a richer vocabulary of taxonomic relations for the data side. There may be many kinds of subtypes, or ways of stating relations between two different types.

SUMMARY

McLeod: In summary, I think we have addressed six issues concerning behaviour.

First, the distinction between data and process depends on your point of view and it is somewhat muddled. Process concerns behavioural information such as manipulation. Data concerns data structures information.

Second, at least three approaches to the behaviour modelling were mentioned: the database approach which offers generic operations on rich data structures; the behavioural characterization approach in which the meaning of data is expressed in terms of the operations on it and the "database as a collection of facts" approach which uses inference and deduction to model system dynamics.

Third, data can be related to process in several ways. In the past they have not been related in databases except recently in data dictionaries in which the link is weak since it is outside the schema. Secondly, abstract data types can be used to modularize behaviour. Thirdly, we can model operations as derived data. Finally, inference can be used to model system dynamics.

Fourth, the best way to model behaviour depends on the intended use. Abstract data types help to organize complex software systems. Algebraic specifications aid in verifying correctness. Predicate calculus is good for inference and deduction while views might be best for end user convenience.

Fifth, we discussed two mechanisms for organizing behaviour: the same structuring primitives used for data and the decomposition of operations into pre-conditions, post-conditions and a body.

A sixth issue was evolution: what is the impact of the behaviour expression technique on system evolvability. Behaviour expressed in programs is hard to modify. You could define generic operators which are assured to be relatively fixed over the database or system lifetimes. However, they are not expressive in terms of application-oriented manipulation.

In conclusion let me mention some tradeoffs and trends. One tradeoff was changeability versus expressiveness. The more explicit the specification, the harder the change. A second tradeoff is understandability versus formality. In specifying behaviour are we interested in proving things or aiding user understanding. A third tradeoff is scope of applicability versus the cost of system design and maintenance.

Two obvious trends are to put more process specification into databases and more structuring into programming languages.

Rich: The things in your tradeoffs seem orthogoral. For example, changeability seems unrelated to expressibility as do understandability and formality. If anything I think formality improves understandability. Known tradeoffs are useful, as in engineering, I do not think we have those here.

Shaw: Some of the most unintelligable descriptions are wrong.