Mylopoulos: When we talk about a model, it can be either a program snapshot or execution, or a program, a data base, a conceptual schema, or a knowledge base. We can think of a program as consisting of units of some sort, e.g., procedures, assertions, data types; and they are related by relationships of various kinds. Some relationships are user-defined and dependent on the domain the model is dealing with. On the other hand, some of the relationships used to describe the model are primitive, in the sense that their semantics are well-defined and embedded in the modelling framework in terms of which the model has been defined. Some examples from the three areas being represented here are ISA, PART-OF, INSTANCE-OF. Certain other relationships include procedural attachment, used in AI to associate procedures to data classes to specify operations on instances of the classes.

In programming languages, considering statements as the units, statement sequencing is a primitive relationship between these units. Considering ALGOL-like begin blocks as the units, scoping rules are relationships between units (blocks). Procedure activation rules between blocks are another example of a relationship that can be considered as primitive and embedded in the modelling framework. Simula concatenation, which allows the definition of classes to be given in terms of other classes gives a relationship between classes. The association of operations to a data type can also be treated as a relationship that has been used in PLs.

Turning to data bases, we can find some examples of relationships in the relational model. There is a notion of relationship between a relation schema and a domain. Every attribute of a schema can be thought of as defining a relationship between the schema and a domain. There is a relationship between a tuple and a relation, that a tuple is an element of a relation. Relationships such as generalization, aggregation, and instantiation are also relationships used to structure data models.

Cristian: Could you be more precise about the definition of a relationship?

Mylopoulos: I am giving examples, hoping that people will understand enough about units and relationships to see how they are used in the three areas. There are really two ways that the notion of relationship is used. The first use, as illustrated by the above examples, is to combine several units into a single larger unit. This helps to structure models, especially as they become larger.

The second use of the notion of relationships is to relate models to other models. Several models of (possibly overlapping) worlds could exist in a system. A specification and a representation (i.e., realization) can be thought of as two models. The same is true for internal and external schemata and views. Multiple representations and hypothetical worlds have been found useful in AI. In software engineering people have talked about different levels of specification starting with something called requirements specification and moving down to design specifications and implementations. One can think of these as models at different levels of abstraction.

If such models are to usefully coexist in a system, it is important to provide mappings between their parts. That is the second issue to be discussed in this session. I do not think there is a clear-cut line where we define the boundaries of the model. One could say just as easily that these relationships are between units of one model. I do not think we should get into an argument over the difference between what a model is and what the components of a model are.

A third area, related to both relationships within models and mappings between models concerns inheritance rules. For every relationship or mapping R, one can define inheritance rules which, for (a,b) in R, allow you to say things about b based on what you know about a. There are examples of inheritance from all three areas. There is default inheritance, used a lot in AI, and exemplified by scope rules in PLs. One can view the subclass function as providing a sort of default inheritance. There is a strict inheritance, as in the use of generalization in DB, where there is no way of masking out inherited properties. One can also talk about the inheritance of attributes, or of values. For example, since a student is a person and a person has an "age" attribute, a student inherits the right to have the attribute, age.

The issues that I think are important and should be discussed are:

(1) The identity and classification of relationships, mappings and inheritance rules.
(2) Is there a universal set of primitive relationships?
(3) Should a modelling framework offer primitive relationships or should it be representationally neutral?
(4) Design principles for a modelling framework that supports several primitive relationships.

With respect to point (4), most of us are here because we have been involved in designing a modelling framework. It seems that the number of features one could consider for inclusion is larger than ever before, because we are looking for a more "semantic" or "natural" modelling framework. One may be tempted to just include as much as possible. There is something to be said for design criteria, i.e., how does one distinguish a well-designed framework from one that is not well designed? Just as there is something "nice" about Simula's class notion, what are "nice" features of modelling frameworks, and how can such features be integrated in nice ways. I think this issue will also be addressed in Joachim Schmidt's session.

Hayes: Does "primitive" mean the same thing in point (2) as in point (3)? In (2) it looks like you are talking about a universal set of language or syntactic constructors or types in a programming language. In (3) I think of Roger Schank and his 13 primitive actions, which is a different kind of primitive, where you are talking about all of your knowledge reducing definitionally to the particular primitive notions.

You can think of Schank's modelling approach as a kind of graph grammar, where there are, say, three syntactic ways to put graphs together but 13 semantic relationships.

Rich: There is a tradeoff between providing primitive relationships versus remaining neutral. The advantage of providing more primitives is that the designer of a system can provide a lot of built-in machinery for manipulating the system in terms of those primitives. As a user of a system, I would prefer having a lot of built-in relationships.

Hitchcock: If there are too many built-in features, in some sense things may be overspecified. You have to beware of that.

Mylopoulos: The remainder of the session will be devoted to five short presentations by Bertrand Meyer, Diane Smith, Gary Hendrix, Paolo Paolini, and Martin Feather.

THE SPECIFICATION LANGUAGE Z

Meyer: Z is a specification language that is based on logic and set theory. It was designed mostly by J. R. Abrial, with help from S.A. Schuman and myself. Z is not a programming language. It is a problem description language whose design has been influenced by programming languages.

Basic Building Blocks

There are three main kinds of modules.

(1) One program module is a description of one object (process, data element), e.g., FORTRAN or ALGOL subprogram.

(2) One program module is a pattern for a class of potential objects, which must be individually declared and instantiated, e.g., Simula class.

(3) One program module is a collection of (hopefully related) object descriptions, e.g., ADA package, FORTRAN common.

In Z, these functions are met by chapters and classes. A chapter has definitions, which define functions, sets, classes and theorems. A chapter also has a "use list" which gives a list of other chapters that can be referenced in the present chapter.

Classes are used to describe structures; for example, mathematical structures, the structure of an operating system, or an abstract data type. Classes have primary attributes, derived attributes, subclasses, objects, and function concepts such as injection and projection.

The basic form of a class definition is illustrated by the following definition of a set of tables T whose components are drawn from a set X.

```
TABLE[X,T] =
    class with
        emptytable: T;
        insert: X x T -> T;
        has: T -> subset(T)
    where
        a: has(emptytable) = null
        b: has•insert = op(U) • (single[X] #has)
    end
```

The first line defines the generic parameters, the sets X and T, of the TABLE definition. The binding (instantiation) of the parameters is shown below. The generic parameters will always be sets. The second section (following with) defines the primary attributes which, in this case, are operations that yield, respectively, a given element of T; a new table given an existing table and an element of X; and the set of elements of X in an existing table. The third section (following where) specifies the constraints. These state (a) that there are no elements of X in the emptytable and (b) that has of insert gives the union of the previous elements of the table with the new element of X. [The last constraint is expressed in a variable free, combinator form similar to that used by John Backus in his FFP system. (ed.) It is equivalent to

```
    forall x,t for x:X, t:T,
        has(insert(x,t)) = {x} U has(T) ]
```

Thatcher: You called T a generic parameter, when it seems like it is the value being returned. It looks like X is the parameter and T is the value.

Meyer: No. They have exactly the same status.

Instantiation

Meyer: The class may be instantiated via the cons construct. The following code shows how tables might be constructed from sequences, which are predefined in Z.

```
SEQTABLE[X] =
    cons TABLE[X, SEQ[X]] with
        emptytable = null;
        insert = make;
        has = range[NAT, X]
    end
```

Here the primary attributes are bound to the predefined operations null, make and range for the set of SEQuences. The set of REALSEQTABLEs could be similarly defined by instantiating the X of SEQTABLE with the set of REALs.

Hayes: TABLE seems to be just a definition with three parameters, emptytable, insert, and has. Basically, it's a functional with those three parameters. This seems like the lambda calculus to me.

Meyer: So far that is true, but that will not be true for subclasses.

Hayes: There is nothing like the encapsulation idea here, because null, make and range are defined outside of TABLE. You could supply some parameters there. There is nothing private to TABLE that can not be accessed from outside. (Meyer) That emptytable, insert, and has are bound to null, make, and range is private.

Meyer: The validity of the construction of SEQTABLE can be expressed in a proof clause.

Rich: Could you say something about the purpose of your language? Is it for verification, refinement, or what? And who is the audience?

Meyer: The purpose is basically to specify. We want to give a formal description of a problem. The audience is the professional program designer. I am not suggesting that this is going to be in the hands of the accountants. Our aim is to give a precise definition of the problem at hand, with of course the hope that getting insight into those problems will make the solution process much easier and give us a precise basis for checking our final programs.

Subclasses and Class Combinations

Subclasses can be defined either by extension (enrichment) of classes, and/or by contraction (restriction) of the properties. For example we can define a subclass of TABLE with a new operator, delete, as follows:

```
DTABLE[X,T] =
    class TABLE[X,T] with
        delete: X × T -> T
    where
        has•delete = op(-) • (single[X]#has)
    end
```

In the general case, we can have new attributes, new constraints, or both. This way, we can work top-down according to a technique of successive refinements. In addition, derived attributes can be defined in terms of primary attributes. This is especially important where there is a relationship between the attributes that must be preserved, such as with the son-of and father-of attributes.

We could also work bottom-up by combining existing classes. We have a "join" operation. If class X is the join of B,C,D, then it provides all the attributes of B,C,D. We also have a union operation. If class Y is the union of U,V,W, then the elements of Y may have the attributes of either U,V, or W.

Deutsch: What does a join mean if some attributes of B,C, and D have the same name?

Meyer: There is no overloading. Attributes with the same name have to be compatible. Suppose class A has attributes X and Y, and the class B has the attributes X and Z; then, A join B would have the attributes X, Y and Z, assuming X had the same definition in A and B.

Codd: Is the union permitted over arbitrary sets of classes?

Meyer: Yes, provided there is no conflicting use of the same name. Using classes A and B above, the elements of A union B would have an X attribute and either a Y attribute or a Z attribute.

Usage

Rowe: What is the largest thing you have specified in this language? (Meyer) The largest thing I have specified is a nuclear plant protection system, which is reasonably complicated and has high reliability demands. It took about 15 pages of Z. (Rowe) What use have you made of the specification. (Meyer) We are not doing anything now. (Rowe) Do you have some idea what the actual size of the program would be? (Meyer) It would be a factor of two or three bigger than the specification, I would guess.

For more information, see the paper on Z in the Proc. Summer School on Program Construction, Belfast, 1979, Cambridge University Press, 1980.

PRIMITIVE RELATIONSHIPS IN DATA BASES

D.Smith: John and I are interested in semantic models. We begin by first considering a model with uninterpreted relationships. For example, the relational model is relatively uninterpreted, i.e., can be interpreted in any way the modeler finds useful. Then we add a small number of interpreted relationships. The emphasis in our research is on which relationship to add. These simplify application programming by moving more semantics out of the programs and into the responsibility of the data base management system.

Balzer: When you say "interpreted" relationships, do you mean the system understands some of the semantics of those relationships?

D. Smith: Yes, these are relationships the application programmer can use as building blocks. One example is the relationship called specialization/generalizition. It is similar to ISA in AI.

Our model is called the Semantic Hierarchy Model. Like most other data models, it consists of 5 parts:

- (i) data space (individuals and relationships)
- (ii) type definition language (constraints)
- (iii) manipulation primitives (create, destroy, modify)
- (iv) predicate language (first order logic)
- (v) control structure (iteration, recursion)

We have decided to include the notion of an individual in our framework. An individual maintains its individuality during its existence in the system and can participate in relationships.

Rich: What would a system be like without the notion of an individual?

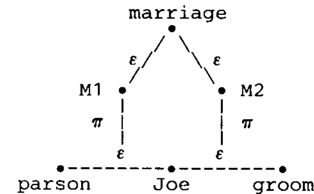
D. Smith: Some systems have a notion of record, but that does not necessarily model an individual. A record may not correspond to an individual. A real-world individual may map into several records, and that is detail the application programmer must deal with.

Our data space is a 7-tuple defined as follows:

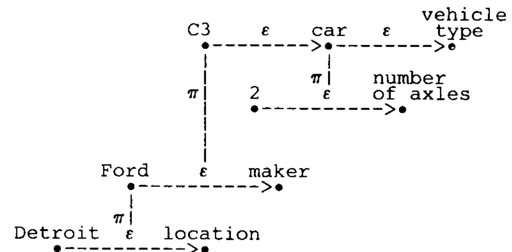
$$<I_p, I_t, N, \phi, \varepsilon, \equiv, \pi>$$

I_t - set of type individuals
I_p - set of primitive individuals
$I = I_t \cup I_t$
N - set of individual names
ϕ: N -> I - "name" map (is 1-1, into)
$\varepsilon \subseteq I \times I_t$ - "instance-of" relationship
$\equiv \subseteq I \times I$ - "identity" relationship
$\pi \subseteq \varepsilon \times I$ - "attribute-of" relationship

We have a pictorial notation. Individuals are nodes and the arcs represent the primitive relationships.



On this next diagram we see that types can have types and attributes.



Vehicle type is a higher order type. All individuals are treated uniformly, except that primitives may not have instances.

Hayes: Does component mean physical part? (D. Smith) No. (Hayes) It sounds as though your relationship is

uninterpreted. (D. Smith) It is interpreted at one level just as a cross-product.

Reiter: I would say it has a logical representation. Surely this all maps into logic. It looks like second-order, because you are allowing a type "car" to be an instance of another type "vehicle type". But you could, nevertheless, give a mapping from these diagrams into logic. Then there would be no further discussion of what they mean.

D. Smith: That might be a mapping worth doing, for clarification, but the logical representation would not be used as a modelling tool.

Zilles: The point seems to be that we should separate the semantics the system provides from the semantics that the problem domain has to add. I think the choice of system semantics is within the category of "universal primitives" that John Mylopoulos was talking about earlier, in the sense that these are built-in notions that the user does not have to define, but they do not provide all of the interpretation.

Hayes: I am still unclear on the interpretation of "component-of". (D. Smith) The discussion of type definitions may clear this up.

Type Definitions

D. Smith: A type definition (schema) is a 3-tuple, consisting of a set of type names, a "subtype-of" metarelation, and a "component-of" metarelation. For example, body, shaft, wheels, etc. are components of a car, and land or air vehicles are subclasses of car. This is the kind of information that goes into a type definition.

The primitive relationships can be used as predicates in the query language. So we can ask about the structure of the data base as well as the content. For example, we can ask about the type of an individual. Or, we can ask if there is some relationship between two individuals.

Mayr: Doesn't Thompson's REL/REALM have nearly the same expressive power?

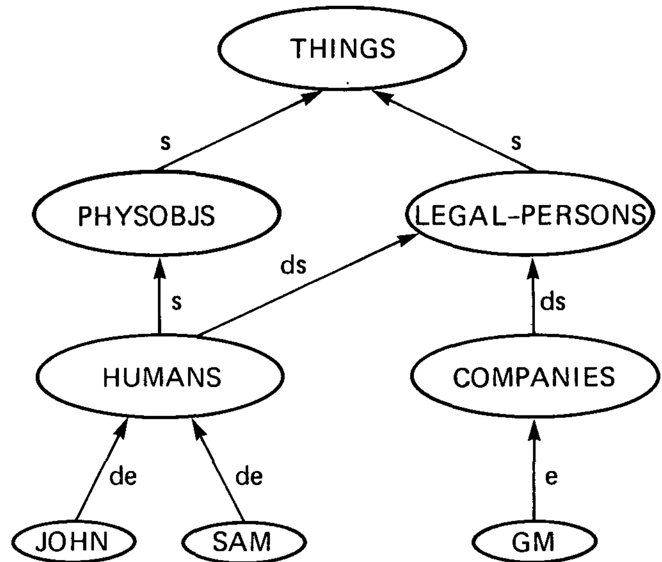
Buneman: I couldn't find an obvious way to ask questions about the structure of the data base in REL.

SEMANTIC NETS AND FIRST ORDER SYSTEMS

Hendrix: I have two main points. First, I want to talk about how relations are used in abstraction. Second, I want to talk about doing abstraction with a two-level system of first-order languages. Everything said so far at this workshop implies that the same language is being used for both the abstraction and the thing being abstracted. This can cause all kinds of problems that we can get around that by using multiple languages, each language based on a first-order system.

We have experimented with semantic nets, which are equivalent to predicate calculus. The semantic net form is just syntactic sugar, but sometimes a spoonful of sugar helps the medicine go down.

The two figures, Hendrix-1 and Hendrix-2 are equivalent statements. I think for ordinary folks, Hendrix-1 is slightly easier to read. But when you try to put the semantics into the computer, you've got to go to a linear notation. They are not really equivalent, because Hendrix-1 is based on sets Hendrix-2 is based on propositions. I think Hendrix-2 is superior.



s subject
ds disjoint subset
e element
de distinct element

Hendrix-1

$$
\begin{array}{l}
\forall x\ \mathrm{thing}(x) \\
\forall x\ \mathrm{physobj}(x) \Rightarrow \mathrm{thing}(x) \\
\forall x\ \mathrm{legal_person}(x) \Rightarrow \mathrm{thing}(x) \\
\forall x\ \mathrm{human}(x) \Rightarrow \mathrm{physobj}(x) \wedge \mathrm{legal_person}(x) \\
\forall x\ \mathrm{company}(x) \Rightarrow \mathrm{legal_person}(x) \\
\forall x\ \neg \mathrm{human}(x) \wedge \mathrm{company}(x) \\
\mathrm{human}(JOHN) \\
\mathrm{human}(SAM) \\
\mathrm{company}(GM) \\
JOHN \neq SAM \\
\forall x\ \mathrm{legal_person}(x) \Rightarrow \\
\qquad \mathrm{human}(x) \vee \mathrm{company}(x)
\end{array}
$$

Hendrix-2

Hardgrave: Dont you mean that they are equivalent under one interpretation of the graph in predicate calculus. Several interpretations are possible. (Hendrix) Yes.

Hendrix: The semantic net shows the structural relationships among the sets mentioned. In Hendrix-1, THINGS is the most general set, PHYSOBJS is a subset of THINGS shown by the S-link, and so on. The ds-link shows disjoint subsets. So HUMANS and COMPANIES are disjoint subsets of LEGAL_PERSONS. GM is an element of COMPANIES, shown by the e-link. And JOHN and SAM are distinct elements of HUMANS, shown by the de-links. By distinct we mean we guarantee that JOHN and SAM refer to different objects.

Sibley: If you had "x element of HUMANS" would you know x and JOHN are distinct elements?

Hendrix: No. You only know that the elements connected up by de-links are distinct from each other. In Hendrix-2 this shows up easily as JOHN ≠ SAM. So you can see predicate calculus wins some time in terms of the notation.

But, when you get right down to it, just because you express something this way does not mean you have to implement it this way. There are some interesting internal schemes. I think one of the things Ray Reiter was saying

has been misinterpreted. I do not think he means everyone has to use predicate calculus. I believe he means that it would be a good idea to interpret your notation in predicate calculus so everyone would know what you are talking about.

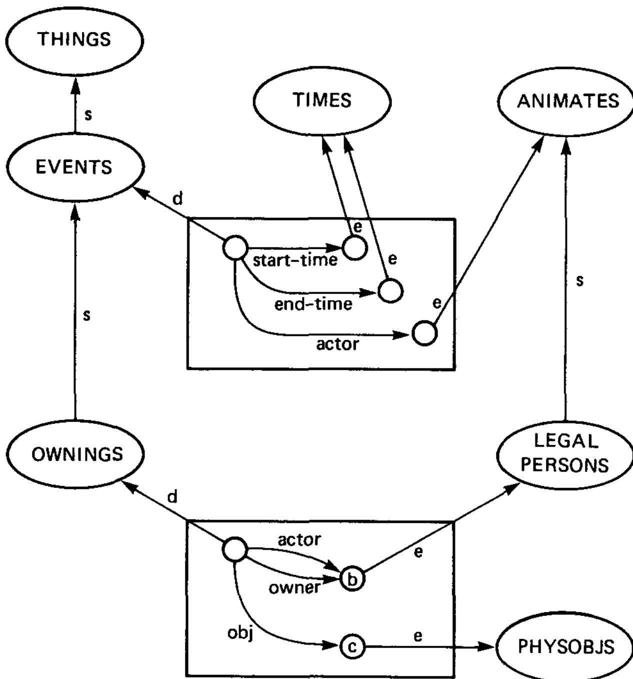
Hayes: Including the system that manipulates it.

Hendrix: Exactly.

So, in Hendrix-2, THING is a predicate, or a type. A subtype is a simple implication. We have said that HUMANS and COMPANIES are disjoint subsets of LEGAL_PERSONS, but we haven't said that they cover all LEGAL_PERSONS. You can do that in your notation any way you want, but in predicate calculus you can fall back on well known techniques. For example,

$P(x)$	a type predicate
$Q(x) \Rightarrow P(x)$	a subtype
$\neg(P(x) \wedge Q(x))$	a disjunctive union type
$P(x) \Rightarrow \exists y\, Q(x,y)$	inference

Now we can discuss properties that are meaningful for a particular type. As an example, we use the type EVENT, which is a subtype of the type THINGS. Now consider Hendrix-3, which is a syntactically sugared version of Hendrix-4.



Hendrix-3

$\forall x\, event(x) \Rightarrow [\exists st, et, a,$
$time(st),\ start_time(x)=st$
$time(et),\ end_time(x)=et$
$animate(a), actor(x)=a$
$before(st,et)\ ]$

$\forall x\, owning(x) \Rightarrow [\exists b, c$
$legal_person(b),$
$b=owner(x)=actor(x)$
$physobj(c),\ obj(x)=c\ ]$

Hendrix-4

The box in Hendrix-3 shows some of the components of an EVENT. Every instance of the type will have a start-time, an end-time, and an actor. The start-time and end-time have to be elements of TIMES, and the actor has to be an element of the set of ANIMATES. Now, in the predicate calculus formulas (Hendrix-4) this is straightforward. They say, whenever you have an event x, x is going to have these attributes. At the bottom of the diagram we see what OWNING is.

My point is that whatever we want to say here can be said quite straightforwardly in predicate calculus. Do we really use a sugared form, as exemplified by my notation? I think we do, but maybe not exactly this sugar.

My second point is that we want two levels of data base language: (1) a low-level data language which allows the content of the data base to be queried, and (2) a meta-language. These would both be first-order languages. The idea is that the metalanguage is used to talk about sentences in the data language. We talk about whether a sentence in the lower level language is going to be true or false. A sentence is an object in the meta-language. We want to relate the sentences of the data language to their use. So we would be able to say in the metalanguage: if you use this sentence in the data language, it is going to answer this kind of question from the user model.

SCHEMA EQUIVALENCE AND VIEWS

Paolini: I want to talk about equivalence of schemas. When do we say two schemas are equivalent? It is important to answer this question. One possible answer is that they are equivalent when they represent the same piece of the real-world. I think this is the kind of answer the AI people would produce to this question. I do not think this is very useful.

Another answer could be the following. Two schemas are equivalent when the two sets of legal sets which can be derived from the schemas are set-isomorphic. This is not very useful because it does not take into account any constraints on the data. A better definition is that two schemas are equivalent when the two algebras defined by the sets of legal states and operations on them are (or can be made) algebraically isomorphic. Typically, a lot of work is required to show two schemas are equivalent according to this definition. A better way is to define an algebraic equivalence that is closer to the semantics of the data base. Rather than using general purpose database operations, such as fetch or update, the applications programmer provides specialized applications-oriented operators (e.g., store-supplier, get-supply, etc.). He would also provide a formal specification for these operations. Then, if these operations can be correctly implemented in terms of the database operations over the two schemas, the two schemas are equivalent.

The Implementability of Views

The question of equivalence is particularly important with respect to views. For example, when is a view V implementable over a schema A? For the same reasons given above, implementability cannot be defined in terms of the real world being modeled, nor in terms of a function which maps the set of legal states of A into the set of legal states of V. A suitable definition is that a view V is implementable over a schema A if there exists a homomorphism which maps the states of V to A and makes the diagram of the operations commute. That is, all operations (e.g., store-supplier) have the same result when applied to the view V as they do when applied to the homomorphic image of the state of V in A.

A Data Definition Language

The algebraic approach to schema equivalence and view implementability suggests the structure of a language for database definition. [This structure is very similar to that of programming languages with abstract data type definition facilities. ed.] In the definition below, the definer specifies the application dependent operations: new_supplier, get_supplier, etc.. He then specifies the representation and gives the implementation of the operations.

```
database db
  is new_supplier(s#, sname)
     get_supplier(s#)
  representation
    type srec = rec s#:___; sname:___ end;
         srel = relation <s#> of srec;
    var  supplier: srec;
         sp: srel
  operations
    procedure   get_supplier(s:snumber,   var
                     n:name)
      begin
        for each t in sp: t.s# = s
          do
            n:=t.sname
          end
      end
```

Then a view can be defined in terms of the operations defined on the database. For example,

```
view b
  is get_supplier
  representation
    use db: database
    var buffer:____
  operations
    <procedures as above but for view form of
     operations>
  end
```

HANDLING TIME IN SPECIFICATIONS

Feather: We want to be able to give a specification (model) and then by a series of mappings move toward an efficient implementation. Now, in the earlier stages we would like to have such things as constraints, quantification, nondeterminism, inference, future reference, and full historical reference. Later we will want to get rid of some of these things. The problem we are concentrating on here is historical reference.

The problem is that we want to describe things that go from state to state, and then be able to refer to previous states in the description. Associating some kind of "memory" with a state (to refer back to) messes up our simple model. Then, the actions (state transformations) must deal with a memory component. And, all states must potentially be able to refer to all previous states. The solution is to extend our information model to deal explicitly with previous states. Each extraction of information from the system state I(R1,...,RN, X1,...,XM), where the R's are relations and the X's are objects, can be qualified with the time T at which that state holds i.e., I(R1,...,RN,X1,...,XM,T). The intent is to make time a first class citizen and have the specification system deal with managing all this information.

Hendrix: What about having two pieces for time, a start time and an end time, and the relation would hold over that interval. And all things that currently hold would be flagged and carried forward until their relation ceased at which time they would be deleted.

Feather: You might want to do all kinds of things, stating that at some particular time something should happen.

Balzer: Gary, the approach you are talking about is equivalent to the one Martin has described. It is just a different way of conceptualizing it. Our (Martin's and my) conception is that the current state only contains information that is still existent, and you refer to the past by explicitly referencing the time of the state. But there are easy mappings between the two. (Hendrix) I think that what I am suggesting costs much less. (Balzer) But this is only a specification. (Hendrix) Ah. Fine. (Hayes) Can times be intervals? (Feather) Yes.

Feather: A non-contrived example which uses a time reference is: "When given lines of text to insert into a unit of text, if no point of insertion is also provided, insert the lines after the last position of change within that unit". A unit is part of a text file. When no explicit position is given, we want to compare the unit contents now with the contents at the time of last change to determine the position where the insertion should be made. We say in our language that

```
∃ latest changetime:time
   ((unit:unit-contents at changetime) ≠
                         unit:unit-contents)
```

This establishes the time "changetime" as the time at which the latest time occurred.

The next step is to map from a time-based model like this, to a time-free model. We want to remove time from the model by introducing auxiliary memory and processes to remember information which will be required in future states.