Smith, J.: The topic of this session is the consistency of models. (I believe that consistency and integrity are substantially the same.) A model basically contains:

- states consisting of entities and various relationships between them,
- operations (functions) for examining states, and
- operations (procedures) for changing states.

The topic will be discussed in terms of the following, rather controversial issues:

1. Kinds of Consistency
2. Specifying Consistency
3. Detecting Inconsistency
4. Living with Inconsistency
5. Concurrency and Consistency
6. Recovery from Failure
7. Exploiting Constraint Knowledge

For each issue I will present my own view, and I encourage alternative views to be expressed.

1. KINDS OF CONSISTENCY

There are three kinds of constraints. The first and most fundamental kind concerns stating which operations are applicable to which kinds of entities. These constraints delimit what operations make sense and what behaviour must be accounted for. The second kind concerns invariants that must hold in a quiesced state (e.g., credits = debits) or across successive states (e.g., salaries increase). These constraints are expressed as functions and are independent of procedures. They delimit the behavious of procedures in general. All procedures must be defined so as to satisfy these invariants.

Balzar: There are also constraints across multiple, not necessarily successive states. For instance, a person fired from a company can never be rehired.

Smith, J.: The third kind of constraint concerns the pre- and post-conditions which determine the behaviour of each procedure. These constraints delimit the behaviour of each procedure, are used to verify code and are used to ensure state invariants.

Zilles: John, how do you distinguish constraints from operation semantics?

Smith, J.: Constraints are specifications to be fulfilled by the operations; they delimit what the operation is supposed to do.

Zilles: There is a distinction, however, between external constraints--those that arise in the domain being modeled--and internal constraints--those which apply to the representation chosen for the model.

Smith, J.: If the constraints are specified separately from the operations then constraint checking can be done either by checking the procedures which access and modify the data or by checking the data after modifications are performed.

Sibley: For example, in the 1971 CODASYL specifications there is the ability to include data base procedures which check prior to the storage of data elements.

PL Treatment of Invariants

Smith, J.: It seems to me that AI people address all three kinds of consistency, DB people ignore the third kind while PL people ignore the second kind.

Christian: PL people also treat state constraints, the second kind.

Smith, J.: Aren't the invariants treated in PLs specified in terms of the representation?

Christian: The internal or representational invariants correspond to external invariants.

Zilles: Typically in PL the only way to change a state is by an operation. Presumably you take all appropriate invariants as the operation definition. The invariants are not stated independently but are built-in.

Christian: In abstract data types the invariant is implicitly part of the precondition. Proving that it is part of the postcondition ensures that every resulting state satisfies the invariant.

Smith, J.: I agree. It is, however, useful to have a separate statement of the constraints (invariants) so that operations can be added and deleted later.

Balzer: It is important to separately state the external constraints as well as the corresponding internal constraint or invariant. In particular, you do not want the external constraints to be determined by what it is the code happens to do.

2. SPECIFYING CONSTRAINTS

Smith, J.: Typically, operation applicability constraints are specified with type definition languages. However, there is no consensus as to which belong in type declarations and which should be specified separately. These specifications state what operations are applicable to instances of types, what types functions return and type overlaps. A subtype inherits all the operations of its supertype but not vice versa. Disjoint types have no common operations unless they have a common supertype. Overlapping types have elements to which operations of both types apply. Type overlap (hierarchies) is important in DB and AI and is becoming important in PL.

Overlapping Types

Mayr: I suggest you use the "class" which permits overlaps rather than "type" for which it seems odd to attribute two semantics to the same entity.

Zilles: A type can be defined based on a subset of the semantics associated with an entity, therefore, it is perfectly reasonable for something to participate in multiple types. For example, both numbers and characters belong to the type ordered set but they also have many distinct properties as well.

Smith, J.: An example of multiple typing is an individual that is both an instructor and a student. In this case, type overlap would occur if operations of each type were applicable to some students and to some instructors.

Balzer: Type overlap can be defined explicitly a priori or can be permitted a posteriori except when explicitly prohibited.

Smith, J.: An example of a problem with your latter case is when both student and instructor have id_numbers. If a student becomes an instructor is a second id_number required?

Rich: Another approach to type overlap is to treat "point of view relationships" between individuals and their types explicitly. For example, separate types (entities) can be defined for the student and instructor aspects of one individual. Student and instructor point of view relationships are distinct but can refer to the same individual. This permits the roles to be distinguished.

Hayes: If every function of subtypes are applicable to their supertypes you lose expressability. For example, spouse_of applies to females and produces males and vice versa, whereas it does not apply to person, the supertype.

Deutsch: Objects have two aspects: their identity which is time invariant and their state which may vary over time. In programming languages, the notion of type is associated with the identity of the object; it is invariant over time. Talking about determining the subtype or putting an object into a type expresses a very different notion of type.

Specifying Invariants

Smith, J.: Languages for specifying the second kind of consistency, general or state constraints, include the first order predicate calculus, procedural techniques (e.g., demons) and semantic networks. The DB approach is to use predicate calculus and a limited use of semantic networks. I suspect AI uses all three.

Hitchcock: Procedural techniques such as database procedures are used in DB.

Cristian: In PL first order predicate calculus is used to express invariants which are not explicitly checked.

Katz: It is not clear how constraints and behaviour differ.

Brodie: A constraint is any static or dynamic (behavioural) property of the application that must be present in the model for completeness and consistency with respect to the application. Some properties hold because they are implicitly enforced by the data model. These may be specified using the type system. Other properties require explicit checks and are stated in a specification language. This latter class of statements are what is typically called the constraints. However, that use of "constraint" varies from one type system to the next.

Smith, J.: Absolutely. There is a continuum between what you put in type declarations and what you put in general constraints. It is not clear where to draw the line.

Brodie: For each type system the line is clearly drawn between what can and cannot be defined using the types.

Smith, J.: Pre- and post-conditions are generally specified using assertion languages. Typically AI, DB and PL use different languages. There is little controversy here.

3. DETECTING INCONSISTENCY

Smith, J.: The principal concerns in detecting inconsistency are when and how constraints are checked. Operation applicability expressed in type declarations should be maintained true at all times, i.e., inapplicable operations are never allowed. These checks are done at both compile- and run-time. Type overlap tends to force a run-time check.

General invariants should be maintained true at the end of "atomic" procedures, not necessarily in the middle. Checking can be done manually or automatically. Checks for all (necessary) invariants can be embedded manually at the optimum point in the procedure code. Verification can be used to demonstrate that all invariants are maintained. Alternatively, an "integrity subsystem" can be used to deposit checks at the necessary places. The entire subsystem is verified rather than individual procedures. The automatic method is better for handling unanticipated changes to invariants, however, it is limited in the kinds of

procedures and invariants that can be handled effectively. The manual method relies on programmer skill and verification methods which may be difficult. The automatic approach gives a stronger guarantee of consistency at the possible expense of inefficiency.

Sibley: You would not wait until the end to check long, complex atomic procedures. Some finer control is needed.

Weber: Is the end the "optimum" place?

Smith, J.: The invariant must hold at the end of the procedure. It should be checked at a place which is sufficient to ensure it and is efficient in time and space over a large number of instantiations.

Codd: Can you analyze programs to determine where to place checks?

Smith, J.: Some systems have limited success in that regard.

Balzer, Deutsch: The important point is that distributed checks are not optimum but sufficient.

Smith, J.: Software engineering techniques are used to verify and debug procedure code to check pre- and post-conditions. Automatic methods do not seem useful. Most or all checks are incorporated in normal program logic.

Sibley: John, I agree that it has not been implemented but I believe that there are reasonable places to do pre- and post-condition checking, e.g., before and after database storage operations.

4. LIVING WITH INCONSISTENCY

Smith, J.: It is not always desirable to remove all inconsistency from a model, e.g., two databases with overlapping information report conflicting ship positions. One approach to this situation is to change the database to conform to the query language semantics, e.g., distinguish the conflicting ship positions by information such as who sighted the ship. Another approach is to change the query language semantics using non-standard logics for inference over inconsistent facts. AI has apparently not had much success in applying multi-valued and modal logics for this purpose. In any case the objective is that the system, rather than the user, cope with inconsistency.

Levesque: First order logic is used to represent true facts. Whereas we may be willing to live with inconsistencies in a database, true facts cannot be inconsistent.

Hayes: There is a difference between incorrect facts and inconsistency.

Shaw: In discussions of program correctness there is a strong inclination to assume that inconsistencies or errors are bad and must be prevented. But errors, particularly in databases, are sometimes tolerated when for example, the cost of preventing and correcting errors may be greater than living with them. A good example is an airline reservation database.

Deutsch: Errors are failures to enforce specified constraints. This gives inconsistencies between the state of the database and the specifications. These are distinct from incorrect facts which are inconsistencies between the model (the database) and the real world.

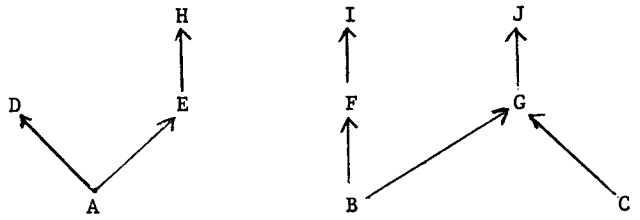
Hayes: The problematical kind of inconsistency is a third kind: inconsistency within the model (database) itself. This kind of inconsistency can propagate in dangerous ways, allowing, in the extreme, any assertion to be proven true.

Using Probabilistic Evidence

Carbonell: Corroborating and disconfirming evidence can be used to resolve inconsistency. Statements can be made as to the confidence in and relevance of the facts or to the sufficiency of premises leading to a conclusion, e.g., if there is gas in the tank (relevance = 1) and oil in the crankcase (relevance = .8) and tread on the tires (relevance = .3) then the auto is drivable.

Resolving Inconsistency

Rich: I will discuss the Truth Maintenance System (TMS) of Doyle and McAllester at MIT which addresses error explanation. Its goal is to maintain a consistent set of beliefs during inferencing with changing premises. Beliefs and truths differ. We can believe that something is true without it being true. TMS has applications in problem-solving, incremental system building and explanations of why beliefs are held. The principal idea is to construct a network which uses "is derived from" in the following diagram where J is derived from G which is derived from B or C, etc.



A contradiction triggers the system to backtrack to find the premises involved. In the example, if H and I are inconsistent then the premises A and B cannot be believed at the same time. The human problem solver decides which premises to give up and the system prunes the network, e.g., dropping B causes F and I to be dropped. The notion of "is derived from" can be expanded so that something, say M, is believed as long as something else, say N, is not believed. This is a form of non-monotonic dependency.

5. CONCURRENCY AND CONSISTENCY

Smith, J.: Consistency or non-interference between independent, atomic procedures has been studied more extensively in DB and AI than in PL. Independent atomic procedures have no knowledge of each other but do ensure consistency individually. A general

purpose mechanism, such as the transaction concept in DB and AI, is needed to ensure consistency between concurrent atomic procedures. The transaction concept, implemented by locking, timestamps, etc. requires all database accesses to be within transactions and produces a result equivalent to running all transactions in sequence.

Another problem is concurrency within atomic procedures. Atomic procedures may contain multiple tasks which know of each other and which cooperate in maintaining consistency. The tasks may understand and be willing to live with various kinds of inconsistency. Mechanisms are needed to handle consistency under cooperative as well as independent procedures. There is active research in both problems.

Zilles: Transactions have two different roles: recovery and consistency. Since most PL work has focused on mechanisms with low overhead, general purpose mechanisms such as transactions have not found favor in PLs. In terms of synchronizing concurrent computations (consistency), there is more interest. Looking at operating systems, there is again a rejection of a general purpose mechanism for insuring consistency. Instead special purpose techniques are used throughout the code.

Sibley: Handling concurrent access is more important in the DB case because many people run the same or similar programs over the same data at the same time. In the PL case, one program is not aware of other programs or data.

6. RECOVERY FROM FAILURE

Hitchcock: Since a database is shared by many users, it is vitally important that the contents are accurate lest erroneous values propagate further errors. The first step in error recovery is error detection. Hardware malfunction may be easy to detect; software failures are more difficult. An important part of data modelling is to capture in checkable assertions, rules or constraints that apply to possible database values. The next step is to assess the amount of damage caused by a failure. This step is simplified if the system has been written in a disciplined fashion. The software equivalent to water tight compartments are the spheres of control suggested by Bjork and Davis of IBM and the concept of recovery blocks put forward by Randall's group at Newcastle University. These both generalize the notion of a transaction. A transaction is a unit of work which is either carried out completely or is not carried out at all. The internal states through which the transaction passes must not be visible externally. The generalized notion of transaction allows complete nesting. Transactions at one level become primitives with which to build larger transactions at another level. The scope of errors occurring within a transaction are bounded by that transaction. The final step is to repair the damage caused by a failure and to return the database to a consistent state.

Two kinds of error recovery are necessary; backward error recovery (backing the system up to a known consistent state) and forward error recovery (continue forward trying to patch the damage). The discipline imposed by nesting transactions eases the task of backward error recovery. If a checkpoint is taken whenever a transaction is entered then that checkpoint must be returned to if the transaction fails. The error cannot be transmitted beyond the transaction's boundaries. If this discipline is not imposed the checkpoint itself may be corrupted and the backout process may cascade. Forward error recovery is necessary if the error is detected after it has been committed to the database. The task of determining and correcting the subsequent errors may be eased using a data dictionary system which can indicate the programs that might have accessed this data and the data they might have changed. An audit trail of the access and modification of individual data items can also be used.

7. EXPLOITING CONSTRAINT KNOWLEDGE

Smith, J.: Constraints can be used to optimize data access, refine specifications to automatically delimit non-deterministic choices and can be used to structure the design and proofs of procedures. These topics will be discussed by the following three speakers.

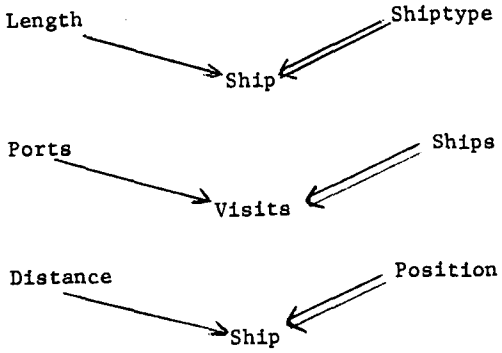
King: Integrity constraints support semantic query optimization which attempts to find queries that are semantically equivalent to and more accessible than the original query. I distinguish value constraints and structural constraints. Value constraints constrain attributes or participants in a relationship. The value constraint that "only tankers are over 500 feet long" applied to the query "ships over 650 feet" produces the query "tankers over 650 feet." Structural constraints constrain relationships, e.g., cardinality. The structural constraint "No more ships at a port than the number of docks" applied to the query "Ports with more than 20 French ships" produces the query "Ports with more than 20 docks and more than 20 French ships." The following example shows how a functional assertion can be used to derive attribute values.

$$\forall x\ f(x) \equiv x \cdot \alpha_1 + x \cdot \alpha_2$$
$$x \cdot \alpha_1 \in [0,10]$$
$$x \cdot \alpha_2 \in [-10,20]$$
$$\{x \mid f(x) > 15\} \rightarrow \{x \mid x \cdot \alpha_2 > 5 \wedge f(x) > 15\}$$

Binding or Accessibility

We have already seen a variety of knowledge represented in the database and used to find semantically equivalent specifications of retrieval requests. Now I want to introduce a bit of terminology to highlight some aspects of physical organization that apply to different kinds of entities represented in the database. "Binding" is a term used by Gio Wiederhold to refer to the degree to which some semantic entity or relationship is reflected in underlying database structures. The stronger the binding, the more accessible is the entity or relationship in terms of ease of retrieval. Of course, there are added costs for maintaining strong bindings. Below we see three examples of different binding strengths. The Length attribute is less strongly bound to each Ship than is the Shiptype attribute. The Ports are less strongly bound to

their Visits than are the Ships. The Distance of a Ship to some location is less strongly bound to the Ship than is its Position.



Adaptive Databases

Bob Balzer stated earlier that one distinguishing characteristic of databases is that they don't change in response to queries. Yet you can imagine putting adaptation to good use. Think of query optimization as an impedance matching problem. If a query is well suited to the existing physical database structures, then efficient retrieval is possible. If not, you can try to alter the query, as I've suggested, or you can try to alter the database structures. This idea is familiar from AI, systems, and programming languages as foregrounding, caching or memo functions. In this case, a good choice of temporary extracts of files may speed retrieval at an acceptable maintenance cost. A related idea is to have redundant or partitioned highlight files, where transformations of certain queries will be permitted. For instance, if there is a very big file with ship position reports over some period of time, and a much smaller highlight file of the current positions of American ships, then it may be worth checking the nationality of the ship whose position we want to find out on the chance that it's American.

Another example might be a database which logs the number of queries made. A query would then cause an update. Some other non-update users of constraints are queries over a given domain, e.g., "where are the British supertankers?" and querying over constraints, e.g., "What nation's ships do you know about?"

Refining Specifications

Balzer: Constraints can be used to refine specifications to be more what and less how. Also, rather than purely checking if an invariant is being violated, a constraint can be used to remove non-determinism from a program by selecting only those paths that do not lead to violations of the constraints. To do this requires an arbitrarily smart choice mechanism with unbounded look ahead or concurrent exploration of alternatives. For example, the program to berth a ship as it enters a port can be specialized by the constraints "only one ship per slip" and "cargo ships require slips with cargo handling mechanisms." Then slips with ships and without cargo handling mechanisms will not be considered as possible berths for cargo ships.

Specialization

Borgida: The generalization or is-a relationship can be used to simplify, by means of specialization, the design and verification of classes and transactions. A subclass must have all the same components and conditions of its superclass. A subclass can be specialized by strengthening the conditions and adding new components and conditions. For example, graduate student can be specialized from student by constraining student properties, e.g., enroll-student, or adding new properties, e.g., supervisor. Every instance of a subclass is automatically an instance of a superclass.

Specialization can be used to simplify verification. Using mathematics you specify the pre- and post-conditions which must be true for transactions operating on a class, e.g., $A\{O_1\}B$. You can specialize operation O_1 to get operation O_2 for a subclass. O_2 must have stronger invariants, e.g., $A \wedge \bar{A}\{O_2\}\ B \wedge \bar{B}$. These specifications can be used to guide implementation using a language like TAXIS. To verify the representation you first verify O_1 and use that verification for O_2 so that only the specialized or addition properties need be proven.