Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis

Brian Nixon, Lawrence Chung, David Lauzon, Alex Borgida,* John Mylopoulos† and Martin Stanley

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

The features of a compiler for the Taxis design language are described and discussed Taxis offers an entity-based framework for designing interactive information systems and supports generalisation, classification and aggregation as abstraction mechanisms Its features include multiple inheritance of attributes, isA hierarchies of transactions, metaclasses, typed attributes, a procedural exception-handling mechanism and an iteration construct based on the abstraction mechanisms supported Developing a compiler for the language involved dealing with the problems of efficiently representing and accessing a large collection of entities, performing (static) type checking and representing isA hierarchies of transactions

1 Introduction

Any database contains information *about something*, i e , it has a *subject matter* (or "world" or "application domain") The overall usefulness of the database is determined partly by its performance characteristics — speed of access, robustness, security and the like — but equally importantly by the degree to which it models naturally, directly, completely and accurately its intended subject matter So-called *semantic data models* were proposed as a means of providing powerful expressive tools for the description of a subject matter Such descriptions, often called *conceptual schemata*, facilitate the logical and/or physical design for a database but also aid in the interpretation of its contents (by designers, programmers and end users) Research on semantic data models has been in full bloom since the mid-seventies ([Abrial, 1974], [Chen, 1976], etc) with a lop-sided emphasis on their expressive power rather than their implementation Most semantic data models adopt an entity-oriented framework (see [Borgida, 1985] for an overview) which includes abstraction mechanisms such as aggregation, generalisation and classification, in addition to rich integrity mechanisms that allow the expression and enforcement of constraints on the contents of a database We assume that the reader has some familiarity with semantic data models and the basic modelling concepts they provide

We are interested in efficient implementations of semantic data models Enhancing the expressive power of any data model obviously affects inversely the computational complexity of its implementation We are interested in the development of a theory that enables us to analyse the tradeoffs involved, and guides us in the design of semantic data models that are expressively adequate for a collection of tasks while being, at the same time, computationally tractable

This paper focuses on a particular class of performance issues for semantic data models, namely the compilation of conceptual schemata The premise is that once a conceptual schema has been defined which includes descriptions of entities that are relevant to the subject matter, as well as procedures that affect the state of those entities, there are implicit and explicit constraints that need to be maintained at all times for the database to be well formed with respect to its semantic data model and to be meaningful with respect to its subject matter For example, an implicit constraint supported by all entity-oriented semantic data models is that referenced entities — e g , the father of John — must exist in the database (*referential integrity*) If an entity is removed, it must first be confirmed that it is not being referenced in the description of any other entity Stating that the manager of an employee must also be an employee and must work in the same department is an example of an explicit constraint that may be included in a conceptual schema Making sure that these or any other constraints are in fact

* Dept of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA

† Senior Fellow, Canadian Institute for Advanced Research

satisfied at all times for a particular database is a computationally expensive proposition Doing this checking once-and-for-all statically at the time the database is under construction (i e , during "compilation" of the conceptual schema) is an obvious, well-established technique for introducing run-time efficiency in the implementation of a database The main purpose of this paper is to outline the issues that were raised during the implementation of a compiler for Taxis [Mylopoulos, 1980], a semantic data model intended for the design of interactive information systems

Obviously, the extent to which static checking is possible depends on the features of the semantic data model used to define the conceptual schema Some semantic data models, notably Galileo [Albano, 1985] start with a strongly typed language (in Galileo's case, ML) and extend it with semantic data modelling features which leave by-and-large strong typing unaffected Static constraint checking here is obviously a priority At the other extreme, an object-oriented language such as SMALLTALK-80 [Goldberg, 1983] relies almost entirely on dynamic constraint checking In terms of its features, Taxis lies somewhere between the two extremes

Like all semantic data models, Taxis offers a variety of mechanisms for the representation of constraints These constraints need to be enforced at all times for a database to be meaningful Because of the features of Taxis, it turns out that we can statically enforce "structural" constraints (for example, constraints on the names of attributes) but are sometimes forced to do "value checking" at run-time (e g , checking an integer sub-range) Thus the answer to "Are things statically correct?" for a particular expression in general will be one of yes, maybe or no

Details about the design of a Taxis compiler can be found in [Nixon, 1983] and [Chung, 1984] Additional questions and potential solutions were addressed during an implementation project for a Taxis compiler carried out at the University of Toronto between 1983 and 1986 and are presented in [Nixon, 1987] The approach chosen is to translate Taxis programmes into a Pascal-like language, augmented with relational database facilities (e g , [Schmidt, 1977]), and to use a very simple relational schema Another approach is to map to files with indexes and to give the designer flexibility in choosing the exact storage mechanisms used (e g , [Chan, 1982])

Section 2 of the paper outlines basic concepts of the Taxis data model and the data structures and algorithms used for their storage and access In section 3 we present some of the problems encountered in compiling expressions and the solutions that were considered for these problems A description of the compiler can be found in section 4, while section 5 presents related research, and summarises our conclusions and contributions

2 Basic Features of Taxis and Their Implementation

This section sketches entities and their associated attributes, and then considers implementation issues

2.1 Entities and Attributes

A fundamental concept for entity-oriented data models is that of an *entity* Entities can be created and destroyed and they have a unique identity, i e , are different from all other past, present and future entities Taxis treats all elements of a Taxis programme as entities, including numbers, strings, exceptions and classes (which are used to represent not only the database schema but also the applications programmes that operate on it)

For example, consider the following class declarations

```
define AnyDataClass Person with
  unchangeable
    name  AnyString
    SIN  SINValue
  changeable
    addr  Address
    age  {| 0  120 |}
  unique
    personID  (SIN)
endAnyDataClass

define AnyDataClass Employee isA Person with
  unchangeable
    emp#  {| 0  9999999 |}
  changeable
    age  {| 18  65 |}
    sal  {| 10000  100000 |}
    pos  {| president, secretary, janitor |}
    dept  Department
  unique
    employeeID  (emp#)
endAnyDataClass

define AnyTransactionClass ReviewSalary
       (e  Employee, incr  Money) with
 prerequisites
   moneyAvailable?
    (e dept salTotal+incr<= e dept maxSalTotal)
    elseRaiseException NoMoneyAvailable(e,incr)
 actions
   changeSal  e sal <- e sal + incr
   report   (e sal) print
   changeTotal
    e dept salTotal <- e dept salTotal + incr
 postrequisites
   tooHigh?  (e sal > 75000)
    elseRaiseException SalaryTooHigh(e,incr)
endAnyTransactionClass
```

```
define AnyExceptionClass NoMoneyAvailable with
  unchangeable
    e  Employee
    m  Money
endAnyExceptionClass
```

The first declares the class Person with unchangeable attributes name and SIN (social insurance number), changeable attributes addr and age, and unique (key) attribute personID consisting of the SIN value [1]

Attributes are typed Class definitions induce factual attribute values on their instances Suppose JohnSmith is an instance of Person Then the *factual attribute value* JohnSmith addr (= '123 Maple Street') must be an instance of the *definitional attribute value* Person addr (= Address) The second declaration defines the class Employee as a *specialisation* of Person Thus Employee inherits all attributes of Person and has additional ones specified by its declaration In addition, for Employee the attribute age is refined by restricting the allowable range of values [2] The third class declaration describes a transaction [3] which reviews an employee's salary and changes it, provided there is money available in the employee's department If there is no money available, an instance of the exception class NoMoneyAvailable (see the fourth declaration) will be raised for the problematic employee and his proposed increase, and then a procedure-oriented exception handling mechanism similar to one first proposed in [Wasserman, 1977] will be invoked Thus transaction definitions are treated in Taxis as class descriptions, transaction invocations are treated as instances of these classes, and most innovatively, transactions are also organised into specialisation hierarchies (e g , reviewing the salary of a manager is a specialisation of reviewing the salary of an employee)

An entity, be it generic (e g , Person, ReviewSalary) or non-generic (e g , JohnSmith, 7, 'John') must be an instance of one or more classes For example, an entity created to be an instance of Employee will also be an instance of Person, while Employee and Person are both instances of the metaclass AnyDataClass Among all the classes a data entity is an instance of, Taxis assumes that there is always a unique *most specialised class*, i e , one that is below all others in the generalisation hierarchy This assumption is a (reasonable) implementation concession knowledge of the most specialised class for an entity provides full information about the classes of which it is an instance

Taxis attributes are grouped into *categories* which further define the semantics of attribute definitions For example, data classes can have unchangeable attributes (whose values cannot change), changeable and computed attributes, in addition, optional unique attributes ensure that only one entity in a class will have a particular combination of (unchangeable) attribute values Likewise, transactions have parameters, locals, prerequisites, actions, postrequisites and return expression as categories Attribute definition and selection are treated uniformly across categories, as far as possible, both in the language definition and in the implementation For example, the address of a particular person, say JohnSmith, can be obtained with the expression JohnSmith addr regardless of the category of the attribute, while the type of the addr attribute of the class Person can be found with the expression Person addr

Unlike other semantic data models such as ADAPLEX [Smith, 1983] [Chan, 1982] and GEM [Tsur, 1984] [Zaniolo, 1983], Taxis does not permit *multiple valued attributes* For example, the class Person can have at most one age attribute and each instance of Person can have at most one attribute value for age On the other hand, Taxis allows generalisation hierarchies to be acyclic graphs rather than just trees, thus introducing *multiple inheritance* of attributes For example the declarations

```
define AnyDataClass Student isA Person with
  changeable
    school  University
    averageGrade  {| 0  100 |}
endAnyDataClass

define AnyDataClass Tutor
      isA Student, Employee with
  changeable
    instructor  Professor
endAnyDataClass
```

make Tutor a specialisation of both Student and Employee and enable it to inherit attributes from both directions

Multiple inheritance is very useful from a modelling viewpoint However, if unconstrained it can cause considerable grief, both to the designer of an application and to the implementor of a semantic data model that supports it Suppose Student and Employee both have a dept attribute, but Person does not This would make both the definition Tutor dept and the expression JohnSmith dept (where JohnSmith is an instance of Tutor) ambiguous — do we want the department where a tutor studies or the one where he works? We therefore impose an additional constraint on multiple inheritance, based on [Schneider, 1978] every attribute that is multiply inherited must have a unique most generalised class where it is declared Thus, for the dept attribute to be associated both with Student and Employee, it must be

[1] Not all class definitions are shown, for example, Address is presumed to be declared as a string class Also note that the unique attribute actually defines attribute personID on SINValue, with value Person (written SINValue personID = Person), ensuring that (non-null) SIN values uniquely identify persons

[2] This is an example of *strict specialisation*, which is enforced for all Taxis attribute values (except for expressions and statements within transactions, as we have not defined an isA hierarchy for expressions)

[3] There is a whole class of important database issues which have not been considered in this research, especially concerning the recovery and transaction management facilities which are expected of a database management system Hence the term "transaction" as used in this document should be viewed only as a synonym for "procedure or function"

declared in **Person** (or at least in one class somewhere higher than **Student** and **Employee**) This constraint makes sure that there will be no ambiguity in interpreting an attribute or an attribute value, since the multiply-inherited values have a common source, and, when coupled with the strict specialisation constraint, ensures consistency between an attribute and its specialisations [4]

More complete accounts of Taxis as a data model and a language can be found in [Mylopoulos, 1980], [Nixon, 1983] and [Chung, 1984]

2.2 Entity and Attribute Storage

For the implementation of entities, all that is needed in principle is the ability to generate internal identifiers for newly created entities These identifiers can then be used in the representation of relationships to other entities [Codd, 1979] provides a clear account of how such mechanisms might be implemented within the relational model

Our compiler assigns unique internal identifiers to all entities Groups of identifiers are reserved for data entities Identifiers for a non-generic entity consist of a concatenation of the identifier of the entity's most specialised class and a sequence number Thus we can very rapidly determine the most specialised class of which an entity is an instance [5] In addition to coding information about an entity's most specialised class in its internal identifier, the implementation maintains a binary relation **MinClass**,

MinClass(*entity*, **class**)

where this information is stored The relation is useful when one needs to retrieve all instances of a given class Likewise, the **Value** relation

Value(*subject*, *attribute*, **value**)

stores a tuple (of "triples") for each **unchangeable** and **changeable** attribute value and for each **unique** attribute entry of *every* entity [6] Performance is discussed in the next sub-section For strings, the implementation stores an internal identifier and the actual string value

String(*stringID*, **stringValue**)

Since strings are ordered, we also place a unique index on **stringValue**

For example, recall the previous definition of class **Person** If we have a person with name 'John', social insurance number 333444555, address '123 Maple Street' and age 25, we could have

Entity	*Denotes*
01	**Person**
02	**AnyString**
03	**name**
04	**SIN**
05	**addr**
06	**age**
07	**personID**
010008	a **Person** entity
020009	a string value
020010	a string value
25	25, an integer value

with the following tuples

```
MinClass(010008, 01)
Value(010008, 03, 020009)
Value(010008, 04, 333444555)
Value(010008, 05, 020010)
Value(010008, 06, 25)
Value(333444555, 07, 010008)
String(020009, 'John')
String(020010, '123 Maple Street')
```

An important integrity constraint for entity-based frameworks is the *deletion constraint*, which permits a data entity to be removed only when it is not being used as an attribute value In order to enforce this constraint and at the same time provide a means for iterating over all attributes that have a particular entity as value, the implementation of attributes includes inverse attribute selection (back pointers) [7] Storage of back pointers requires a one-to-many relationship between entities and lists of (subject, attribute) pairs This roughly doubles storage required for inter-entity relationships

It is interesting to compare this very simple implementation of entities and their attribute values with some of the alternatives that have been used by other implementations of semantic data models One of these alternatives involves using a universal relation which has one tuple per entity, one column per attribute, and null values for attributes not applicable to an entity Obviously, such an alternative would be highly inefficient with respect to space usage A second, more interesting, alternative involves using one relation per class [Smith, 1977], while another uses one relation for each generalisation sub-lattice that is part of the conceptual schema [Zaniolo, 1983] When using one relation per class, one may store all attributes (newly defined or inherited) of a particular class in the corresponding relation (*horizontal splitting*), or only the newly defined attributes, as done in [Smith, 1983] (*vertical splitting*), either way an entity identifier must be included in each tuple Note that if

[4] Violations of the Schneider constraint force the designer to reconsider the level at which multiply-inherited attributes are initially introduced This schema revision may result in an attribute definition being moved up in the hierarchy (possibly to a newly-defined class) The effect of this process will be a semantically unambiguous definition E g , the tutor's department will have a unique value

[5] However, this makes it hard to change the most specialised class of an entity

[6] For compound **unique** entries, additional relations are used E g , for **unique** entries comprised of two attribute values, we use **Value2**(*subject1*, *subject2*, *attribute*, **value**)

[7] A more general approach is to use any combination of attribute values to access entities [Balzer, 1984]

the Schneider constraint for attribute inheritance is not in force, vertical splitting is harder to implement because it may have to maintain two or more attribute values for a single attribute

2.3 Analysis of Storage Alternatives

In our analysis of alternative implementation strategies, it is useful to make assumptions about the characteristics of the run-time environment, the databases that will be constructed, and the patterns of their usage We use the following parameters and expected values, for a (moderately large) conceptual schema and its associated database

Schema Parameter		*Expected Value*
S	Number of data classes (schema size)	1 000
L	Average number of levels of specialisation (depth of the generalisation hierarchy)	5
P	Average number of attributes per class (new, inherited and specialised ones)	10

Database Parameter		*Expected Value*
N	Total number of data entities	1 000 000
R	Average number of entities per class ($R \approx N \ L/S$)	5 000
N/S	Average number of entities with a particular most specialised class	1 000

It should be noted that the analysis which uses the above parameters is approximate, as some assumptions are made about the uniformity of distribution of entities throughout the classes of a schema

The (assumed) large number of entities does not permit the use of main memory exclusively Thus the database is assumed to be stored in secondary memory Since class definitions are frequently examined, on the other hand, they are placed in faster-access main memory (requiring, say, 1 megabyte) Note that the typical parameter values mentioned here are *not* maxima for the implemented compiler

It is also useful to make assumptions about the expected relative usage of the data access routines While this distribution will be dependent on the domain of application, we do expect that attribute value *retrieval* and *update* will be much more frequent than entity *creation* and *removal* Individual run-time operations on a single attribute value or entity are designed to be fairly efficient, especially with respect to the number of secondary storage accesses

Let $K(t,n)$ be the time required to select t tuples, from a relation of size n, each one by selection on a key Observe that creation, removal, retrieval and modification can all be implemented via key accesses to the relations

An analysis of three alternatives in terms of the above parameters and their typical values yields the table at the bottom of this page

The database storage entry gives the total number of field values, and assumes that each field is of equal length (say, 1 word) Since each alternative requires an entity identifier in every tuple, and horizontal splitting uses the fewest tuples in total, it uses the least storage overall

In order to obtain times for the data manipulation operations, we need to state assumptions on how relations are implemented Assume that B-trees are used, with only key information and associated pointers in each node of the tree In a node, there is one more pointer than key-entry, hence the maximum number of key-entries per non-leaf node (block), b, is

$$b = \left\lfloor \frac{BlockSize - PointerSize}{KeyLength + PointerSize} \right\rfloor$$

Since each non-root node will be at least half full, the height of a tree with n entries, h, is at most [Knuth, 1973, p 476]

$$h \leq \left\lfloor 1 + \log_{\lceil \frac{b+1}{2} \rceil} \left(\frac{n+1}{2} \right) \right\rfloor$$

Assume that $K(t,n)$ is proportional to $t \ h$ [8] Assume further that pointers occupy one word, that $BlockSize$

[8] One additional secondary storage access is required to obtain the actual tuple, this is offset by the access saved by keeping the root of the tree in main memory

Attribute Storage						
Alternative	A Triples		B Vertical Splitting		C Horizontal Splitting	
	Parameter Value	*Typical Value*	*Parameter Value*	*Typical Value*	*Parameter Value*	*Typical Value*
Criteria						
Number of relations	1	1	S	1 000	S	1 000
Key length (number of components (fields))	2	2	1	1	1	1
Average tuple size (number of fields)	3	3	$1+(P/L)$	3	$1+P$	11
Average number of tuples per relation	$N \ P$	10 000 000	R	5 000	R/L	1 000
Database storage (words)	$3N \ P$	30 000 000	$N \ (P+L)$	15 000 000	$N \ (P+1)$	11 000 000
Entity creation or removal time (ms)	$K(P,N \ P)$	800	$K(L,R)$	200	$K(1,R/L)$	40
Attribute retrieval time (ms)	$K(1,N \ P)$	80	$K(1,R)$	40	$K(1,R/L)$	40
Attribute modification time (ms)	$K(2,N \ P)$	160	$K(2,R)$	80	$K(2,R/L)$	80

is 1000 words, and that an access to secondary storage (e g , traversing a B-tree link) takes 20 milliseconds

For entity creation and removal, horizontal splitting is best, as only one tuple is used (and accessed) to store all attribute values of an entity, and the relation size is small Vertical splitting requires access to several (larger) relations, while triples require several accesses to one very large relation

The number of database key accesses for attribute retrieval and modification, which will likely comprise the bulk of processing time, are independent of the alternatives A small relation size improves performance, hence horizontal splitting has the best results, followed by vertical splitting In addition, both horizontal and vertical splitting can exploit locality of reference (e g , retrieving several attribute values of one entity) by caching a tuple, reducing secondary storage access time

Using triples, attribute values for *all* classes are stored in only *one* relation, assuming that only simple **unique** (key) attributes are used, otherwise, the number of relations needed is equal to the maximum number of components used in a compound unique attribute For Taxis, triples were used, mainly to simplify the implementation Although its performance is not the best, it allows translation to a schema with a small, fixed number of relations, that requires no refinement or reformatting even when the semantic schema is extended It also allows a small, safe interface to a strongly typed target language It is interesting to note that the analysis for triples is independent of the depth of the isA hierarchy

The table omits the costs for back pointers We expect that the additional time and space requirements to be roughly double those of triples, and more than double for horizontal and vertical splitting

Other implementations give the system designer more flexibility in selecting storage mechanisms to meet more selectively the needs of a particular application ADAPLEX [Chan, 1982] supports a form of both vertical and horizontal partitioning of entities and their attribute values In fact, arbitrary predicates, specifying which entities to include in which partition, are available to the system designer A form of *semantic grouping* is used where non-inherited attributes are stored near the entity record itself Instead of duplicating inherited attributes, the designer is offered the opportunity to use clustering to appropriately juxtapose the information The Taxis implementation does not address these issues, we feel that a system designer should initially separate the concerns of semantic modelling and physical design

Recently [Weddell, 1987], the issue of obtaining efficient access to attribute values contained in a schema with horizontal splitting has been thoroughly studied [9] Suppose that we have the following (physical) record structure

```
Person   with name, SIN, addr, age
Employee with name, SIN, emp#, addr, age, sal, pos, dept
```

In this case, the record structures are not "aligned," as the **addr** and **age** fields occur at different physical offsets Thus a routine which obtained the addresses of all persons (including students) could not statically determine an offset which could be used for retrieving all attribute values Now this can be accomplished, by re-ordering fields, and adding unused fields However, Weddell has shown that finding an optimal alignment, which minimises wasted space, is NP hard This result constitutes one of the first examples of a performance theory for semantic data models

2.4 The Conceptual Schema

First we note that the conceptual schema must be available not only during compilation, for checking definitions and expressions, but also during run-time, when expressions might require retrieval with respect to the conceptual schema rather than the database (e g , **x dept**, where **x** has as value some class, is the **dept** attribute definition for **x**) Most conventional programming languages do not need to retain their symbol table at run-time, but do copy selected parts to the emitted code "in-line," e g , to perform checking on sub-ranges and subscripts As the frequency of schema examination is high, and as the exact parts of the schema needed cannot always be determined statically, it was decided not to copy parts of the schema (possibly several times) "in-line" into the code This reduces the size of emitted code at the expense of table look-ups

If Taxis did not support attribute inheritance, the storage of classes and attribute definitions could clearly be based on compilation techniques for Pascal-like record types, which is the starting point for our data structure design Thus the entry for each class should include its basic kind (integer, data, etc), its extension (e g , a subrange for integers), its external identifier[10], a compact internal identifier, and a group of attribute entries

The entry for each attribute definition must include external and internal identifiers, the attribute category, and the attribute type (an internal class identifier) This structure is essentially the same for attributes of data and transaction classes, except that for actions of transactions, the attribute value is a reference to a code segment As a result, separate schema look-up routines are not needed for different kinds of attribute values

But since Taxis does have inheritance, we need to look at alternatives for storing attribute definitions Starting with the most compact, we could store, in the isA hierarchy, each class with only its new or refined attributes This essentially rearranges the text of the programme into

[9] This applies to memory resident databases, in our case, it would be helpful for *schema* storage

[10] The external identifier (e g , the string **'Person'**) is recorded for classes and attributes, not only for more informative run-time messages, but also to allow flexible run-time schema examination For example, at run-time, a user could enter an arbitrary class name, and the programme would then list all its attributes

an acyclic graph To determine if a class has inherited an attribute, a search up the hierarchy is needed, examining attributes of more general classes The second alternative has each class store *all* its attributes, be they newly declared, inherited, or refined For the second alternative, execution of transactions is faster, as statement attributes (which define the body of the transaction) are not dynamically inherited and linked This choice was implemented both for the compiler and the run-time system

The closure of the binary isA relationship is precomputed The obvious way to represent the isA relationship is in terms of an $S \times S$ Boolean matrix However, in view of the expected sparseness of that matrix (e g , in a schema of $S = 1000$ classes, we would expect most programmer-defined classes to have fewer than $B = 40$ specialisations), we chose a representation requiring less space and more time for its access Each class has four lists associated with it — one for immediate sub-classes, one for non-immediate sub-classes, and similarly two lists for super-classes In many calculations, it is preferable to first examine *immediate* specialisations/generalisations of a class This is permitted by our chosen representation In addition, searches are allowed in both upward and downward directions, which can improve efficiency So time required at execution is $O(B)$, where B is the number of specialisations of a class, and typically $1 \leq B \ll S$

3 Compilation of Expressions

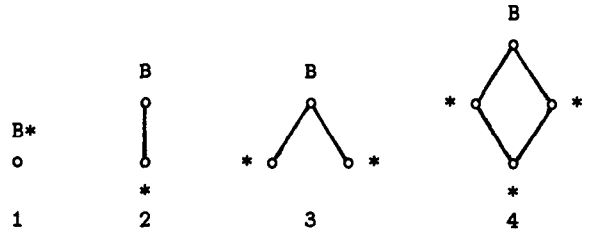
This section examines some of the problems that arise in compiling Taxis expressions and discusses alternatives for their solution

3.1 Type Checking for Expressions

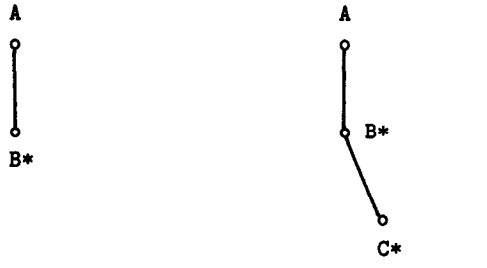
There are three basic steps in checking the type of an attribute selection ensuring that the use of an attribute is legal, determining the resulting value of the expression, and propagating the value to a larger expression This section presents some of the cases that must be considered in performing type checking during compilation A goal of many standard Pascal-like compilers is to statically determine "correct usage" vs "incorrect usage " Due to the richness of the Taxis data model, it is sometimes impossible to statically verify the absence of errors in a programme In fact, a recurring theme throughout expression analysis of the Taxis compiler is the trichotomy of "correct usage," "possibly correct usage," or "incorrect usage " The compiler was written with the view that incorrect usage should be eliminated while possibly correct usage should lead to the generation of run-time checks

Consider an expression of the form `x p` where `x` is a variable of type `B`, and `x p` is a transaction call or within the left- or right- hand side of an assignment There are four basic cases that need to be considered depending on the topology of the isA hierarchy in the neighbourhood of `B` [11]

In the figures throughout this section, an asterisk indicates that the class it is associated with has attribute p, whether newly defined, inherited or specialised



1 *The simple case* Class `B` is said to be simple with respect to `x p` if p is an attribute of class `B` Class `B` may or may not have specialisations The following figures depict the two situations



i) `no-specialisation sub-case` ii) `specialisation sub-case`

i) *no-specialisation sub-case* If there are no specialisations of the class `B` then the simple case is analogous to situations found in standard Pascal-like compilers For example, recall the definition of class `Person` in section 2 1 and suppose there are no specialisations defined for it Now consider the assignment `loc age <- 27`, where `loc` is a variable of such a class, `Person` For complete static checking of the assignment statement, it suffices to check whether `age` is a (`changeable`) attribute of `Person`, and if so, to determine whether the value 27 is an instance of `Person age`

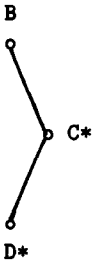
ii) *specialisation sub-case* If there are specialisations of the class `B` then the range of values for an expression `x p` cannot in general be determined at compile time For example, reconsider the above assignment to `loc age` in light of the additional definition

```
define AnyDataClass Child isA Person with
  changeable
    age   {| 0  17 |}
endAnyDataClass
```

[11] So for a general assignment statement of the form `x p <- y q`, there are as many as 4×4 cases to consider

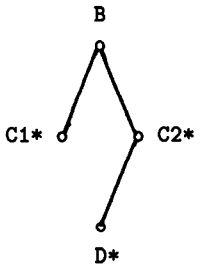
Now the assignment is illegal whenever the variable, loc, refers to an instance of Child In this case, to perform a complete static check would require repeating the analysis for each specialisation of Person, and emitting more complicated code when the check does not succeed for some specialisations [Thomsen, 1984] This costly analysis is not done by the compiler, instead, a simple run-time check is generated

2 *The chain case* Suppose class B does not have attribute p, but some class C does Then a path from C to B forms an *isA chain* with respect to x p if C is a specialisation of B and every other specialisation of B with attribute p is also a specialisation of C The following figure depicts the situation



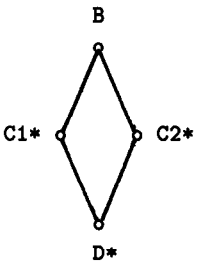
In this case, the expression, x p may actually be undefined at run time (this is referred to as *type uncertainty*), but at least the expression is unambiguous in the sense that if it has a value at all, the value will be an instance of C p So when an attribute p is *not* defined for class B, it is necessary to check the specialisations of B, in this case class C, to determine the type of the expression x p at compile time [12]

3 *The join case* Suppose now that p is not an attribute of B and there are mutually disjoint specialisations of B, say C1, C2, , Cn with attribute p Further assume that any other specialisation of B with attribute p is also a specialisation of one of C1, C2, , Cn The following figure depicts a join of C1 and C2 with respect to B

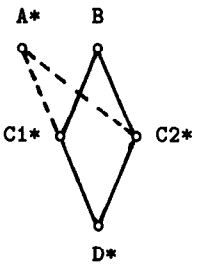


Here x p is clearly ambiguous So in assigning a type to the expression x p at compile time we need to consider the n attribute definitions associated with C1, C2, , Cn respectively Actually, the compiler treats x p as having the type consisting of the union of the instances of (C1 p), (C2 p), , (Cn p) As well, the compiler eliminates isA redundancies from the union type [13]

4 *The meet case* Suppose again that p is not an attribute of B and there are specialisations of B, say C1, C2, , Cn with attribute p, which have a common subclass D The following figures depict a meet having D as a specialisation of C1 and C2



Schneider constraint violated



Schneider constraint satisfied

If there is no generalisation of C1, C2, , Cn with attribute p (left diagram) then the case violates the Schneider constraint for attribute inheritance as described earlier Otherwise there is such a generalisation which is not a specialisation of B (A in the right diagram), we recognise that the Schneider constraint is satisfied, and we treat this as the *join* of C1, C2, , Cn with respect to B

Let's turn to implementation considerations The expression stack machinery found in standard Pascal-like compilers is augmented to handle the propagation of type uncertainty and type ambiguity

Type uncertainty This arises in the non-simple cases (*chain*, *join* and *meet*) when an attribute value selection is undefined at run time If the type of a sub-expression is uncertain then the uncertainty propagates up through an expression For example, consider [x p q, y r] s, which is a compound attribute selection with two subjects (x p q and y r) on attribute s Suppose the type of the expression x p is uncertain, then it follows that the type of x p q is uncertain if it is defined at all On the other hand, the type of y r is unaffected by the type uncertainty of either x p or x p q As for the entire expression, [x p q, y r] s, the type is uncertain (if defined at all) since the type of the sub-expression x p q is uncertain So, to avoid losing information and thus provide more accurate type checking, our data structure design must allow propagation and use of the *certainty* of a nested sub-expression as far up as possible

Type ambiguity Since an expression can be ambiguous (the *join* or *meet* case), the data structures must be extended to handle more than a single type for an

[12] Finding the specialisations of B having attribute p is simplified as we maintain an "inverse attribute table" indexed on attributes which gives us the classes on which any particular attribute is defined

[13] A class T_i (here C_i p = T_i where $1 \leq i \leq n$) is considered *isA redundant* in T1, T2, , Tn if there is some T_j $(1 \leq j \leq n$ and $i \neq j)$ such that T_i is a specialisation of T_j T_i is regarded as *redundant* for the purposes of type checking since all the pertinent information of T_i is conveyed by T_j

expression Consider the expression `x p q` where the sub-expression `x p` is found to be ambiguous Suppose the value range of `x p` is the union type `C1 p, C2 p, Cn p`, then to determine the status of the whole expression, we must consider each case `(C1 p) q, (C2 p) q, , (Cn p) q` individually Note for the expression `x p q` to have a value at all at run-time, the attribute `q` must be defined for at least one of `C1 p, C2 p, , Cn p` If the attribute `q` is defined for *exactly* one then the expression `x p q`, unlike the sub-expression `x p`, is *unambiguous* That is, type ambiguity does not necessarily propagate upwardly through an expression Note further that even if the attribute `q` is defined for all of `C1 p, C2 p, , Cn p`, the whole expression may still not have a value at run-time since the sub-expression `x p` may not have one — that is, type ambiguity is a special case of type uncertainty

3.2 Transaction Hierarchies

Like all classes, transaction classes can be specialised Consider the definition

```
define AnyDataClass Manager isA Employee with
```

and suppose that we want to specialise the `ReviewSalary` transaction so that for managers it increases (decreases) their salary depending on the profit (deficit) of their department in addition to an across-the-board salary increase (decrease)

```
define AnyTransactionClass
    ReviewSalary(e Manager, incr Money)
    isA ReviewSalary(e Employee, incr Money)
 with
  actions
    changeSal  e sal <-
        e sal + incr + (e dept profit / 10)
    changeTotal  e dept salTotal <-
        e dept salTotal+incr+(e dept profit/10)
    stockOption
endAnyTransactionClass
```

An interesting feature of Taxis is the transaction call rule, by which `[x,incr] ReviewSalary` will call the `ReviewSalary` transaction associated with the most specialised classes of the arguments `x` (an employee) and `incr` (a salary increase) If `x` is an employee but not a manager, `ReviewSalary(e Employee, incr Money)` will be called, but `ReviewSalary (e Manager, incr Money)` will be called if `x` is a manager In other words, a transaction call involves the selection of the most specialised transaction that can perform a particular task [14] This is a powerful feature of entity-based frameworks For a system that may include many variants of the same basic operation, say `ReviewSalary`, the selection of the "right" operation for a particular combination of arguments is done at run-time automatically In fact, the programmer *cannot* over-ride the use of the most specialised transaction for a particular transaction call by saying something like "review the salary of `x` considering `x` as *only* an employee, even though `x` is also a manager "

We also note that the Taxis procedural exception handling mechanism uses the same transaction call rule The *caller* of a transaction can specify an exception handler E g

```
[x,incr] ReviewSalary
        exceptionHandler for NoMoneyAvailable
                         is AskDirectors
```

Recall the example in section 2 1 If an exception instance of `NoMoneyAvailable` is raised, execution of the `ReviewSalary` transaction will be terminated, and the handler transaction, `AskDirectors`, will be invoked instead, using as arguments the attribute values of the exception instance In fact, the arguments will determine the most specialised version of the handler transaction to be invoked This introduces a dynamic aspect to selection of an exception handler by combining a simple exception handling model with the Taxis transaction call rule

We now consider the problem of implementing a mechanism to efficiently invoke the most specialised transaction for a particular call This is tied to the more general problem of deciding how to compile transaction hierarchies

The issues considered in evaluating different alternatives for translating transaction hierarchies into a Pascal-like language include the space requirements for compiled code, facility in selecting the right transaction for a particular call, and overall execution speed

Two extreme alternatives present themselves for the compilation of transactions At one extreme, one subroutine is created for every transaction Obviously, execution of a particular transaction is fast, but the space requirements for transaction code are high At the other extreme, transaction bodies are assembled together at run time by splicing `prerequisite`, `action`, `postrequisite` and `return` expressions and statements This simplifies considerably the compiler's job and leads to the generation of very compact code (no expansion of code is required), but execution is slower due to the overhead of assembling a transaction

The Taxis compiler opts for an intermediate solution based on the observation that statement *attributes* are the units of inheritance for actions This suggests that there is no need to duplicate the actual statements, as long as we record which attributes should be executed for a given list of parameters This fact can be exploited in two different ways

1 A decision table can be kept, with attribute versions on one axis, and conditions for execution (i e , the classes of the parameters) on the other One compilation approach attaches "guards" to each version of an attribute

[14] While not shown in the example, any or all of the parameters may be specialised For instance, different constraints could be applied to salary increases of different sizes

For the `ReviewSalary(e Employee, incr Money)` transaction we would have

Guard	*Attribute*	*Code Reference*
`Employee, Money`	`changeSal`	`Stmt01`
`Employee, Money`	`report`	`Stmt02`
`Employee, Money`	`changeTotal`	`Stmt03`

The Guard field means "execute the statement if the parameters are instances of the specified classes " The Code Reference points to the code for the statement, which is stored only once

Now we extend the table to handle the case of `ReviewSalary (e Manager, incr Money)`

Guard	*Attribute*	*Code Reference*
`(Employee,`*not* `Manager), Money`	`changeSal`	`Stmt01`
`Employee, Money`	`report`	`Stmt02`
`(Employee,`*not* `Manager), Money`	`changeTotal`	`Stmt03`
`Manager, Money`	`changeSal`	`Stmt04`
`Manager, Money`	`changeTotal`	`Stmt05`
`Manager, Money`	`stockOption`	`Stmt06`

Now the guard must block execution of the code for certain parameter combinations To execute a transaction, one need only examine each line (entry), and execute the code when the arguments meet the conditions of the guards

Each statement is stored here only once, but the decision table must also be stored, its space requirements grow with the number of specialisations The guards also become more complicated when there are several parameters Furthermore, a linear scan will not obtain the correct order of execution of specialisations (This could of course be achieved by rearranging table entries at compilation) Finally, the number of guard conditions evaluated at each transaction invocation will be far greater than the number of attributes actually executed

2 Another alternative is to determine *at transaction entry* which attributes are to be executed, by considering the arguments of the transaction call with a lookup table such as

Transaction Attribute	*Transaction Arguments*	*Action Attribute*	*Code Reference*
`ReviewSalary`	`Employee, Money`	`changeSal`	`Stmt01`
		`report`	`Stmt02`
		`changeTotal`	`Stmt03`
	`Manager, Money`	`changeSal`	`Stmt04`
		`report`	`Stmt02`
		`changeTotal`	`Stmt05`
		`stockOption`	`Stmt06`
`AskDirectors`			

At execution, the arguments are checked against the table and the most specialised combination is chosen

One method of compiling this is to post-process the list of guards from Alternative 1 Another method is to handle attribute inheritance for transactions in the same way as for data classes and to store a code reference with every `prerequisite`, `action` and `postrequisite` attribute

Alternative 2 is used by the compiler for a number of reasons selection is done only at transaction entry, avoiding repeated evaluation of guards, a minimum number of attribute entries is examined, the decision table requires more storage for the guards (although it has fewer entries), the lookup table provides the desired ordering of attributes, and compilation of transactions using a lookup table is handled by the same compiler routines that handle all other classes

3.3 Iterative Statements

Taxis adapts the iterative statement found in conventional programming languages to exploit the three abstraction mechanisms of the data model (generalisation, classification and aggregation)

An important aim in designing the compiler was to avoid repeated checking of expressions in the body of a loop (e g , while iterating over all specialisations of a class), whenever possible Here are the cases that were treated

1 *Classification* This corresponds to iteration over the elements of a finite set in conventional programming languages For instance, to print the average grade of every student we have

```
for each instance y of Student do
   (y averageGrade) print
```

where `Student` is the "iteration class " This leads to exactly the same type checking as with the declaration

```
y  Student
(y averageGrade) print
```

according to the four basic cases for type checking described previously We can also iterate through instances of metaclasses, and use the same checking techniques For example, to print the number of instances of each class which is itself an instance of the metaclass `PersonClass` we have

```
for each instance cl of PersonClass do
    (cl cardinality) print
```

The case of nested iteration is interesting

```
for each instance cl of PersonClass do
    for each instance x of cl do
        (x age) print
```

To avoid exponential checking, which arises when the iteration class is a variable (here, `cl` in the inner loop), we must find a specific compile-time constant which best represents the iteration class at each level of nesting, and permits a good measure of type checking in one scan of the loop body To accomplish this for nested instance iteration, we introduce the data model constraint that each metaclass must have a unique "most general instance," i e , all instances of the metaclass must be arranged in an (isA-defined) acyclic graph with a unique top element

In view of the above, type checking proceeds as if the inner loop header (`for do`) in the example is replaced by a local declaration `x Person` where `Person` is the most general instance of `PersonClass` Type checking proceeds with the same rigour as the non-nested case For example, the (static) type of `x age` is `Person age`

2 *Generalisation* This form of iteration allows us to traverse the isA hierarchy in either direction To print the `age` attribute type for specialisations of `Person` we have

```
for each specialisation c of Person do
    (c age) print
```

Type checking proceeds as if the loop header is replaced by an assignment of the iteration class to the loop variable

```
c <- Person
(c age) print
```

Note that this differs from our other techniques for type checking, in that we give the loop variable a value, not a type This weak check is, in fact, reasonable since every specialisation has at least the attributes of the iteration class, and makes iteration over specialisations analogous to iteration over instances

Likewise, we can iterate over generalisations of a class

```
for each generalisation c of SeniorCitizen do
    if (70 instanceOf (c age)) then
```

For type checking, we use the same method as for specialisation iteration,

```
c <- SeniorCitizen
if (70 instanceOf (c age)) then
```

but the results are different Now, when an attribute (e g , `age`) is defined on the specified iteration class (e g , `SeniorCitizen`), attribute selection will not be valid whenever the attribute is undefined for a generalisation However, we have an important negative result if an attribute is undefined on the specified iteration class, we know immediately that the expression is always erroneous In addition, this case can lead to strong conclusions concerning attribute *values* assume the if-condition is true when `c = SeniorCitizen`, now if any generalisation of c does indeed have an `age` attribute, the if-condition *must* be true

3 *Aggregation* The intention here is to allow iteration over the attributes of a class

```
for each attribute p of Person do
    (Person p) print
```

This can be used to write general transactions for user interfaces for examination of the schema and the facts in the database

```
for each specialisation c of Any do
    for each attribute p of c do
        (c p) print
```

Type checking for attribute iteration is equivalent to the declaration

```
p AnyAttribute
```

before the loop body, where AnyAttribute has as instances all defined attribute names for a particular schema

In some cases, the compiler can determine that an attribute selection is a legal expression (i e , the attribute is defined for the subject), but cannot statically determine the resulting *value* of an attribute selection This is the only case in the compiler where type information cannot be propagated when examining nested expressions Consequently we depend on run-time checking for this case, this is the price to be paid for the generality of this language construct More strongly-typed languages do not permit this kind of operation, reducing the need for run-time checking

4 *Inverse Attribute Iteration* While the above iteration forms are centred around classes and their attributes, inverse attribute iteration focuses on the attribute values of a specified *data entity*, say `ent`, by selecting all subject-attribute pairs (`subj`, `attr`) such that `subj attr = ent` A motivating use of this construct involves user-enforced compliance with the deletion constraint (which prevents an analogy of the "dangling pointer" problem in programming languages)

```
locals
  ent Person
actions

 for each inverse attribute selection subj attr of ent do
       subj attr <- nothing
 removeEntity ent
```

This is somewhat novel in that there are two iteration variables Surprisingly, reasonable type checking is possible, at least when the two variables are used together in the form `subj attr`, taking the type of `subj attr` to be the same as the type of `ent` [15] This is a weak check because the loop body can contain assignments of the form

```
subj attr <-
```

(which allows inverse references to be removed), thus the type of `subj attr` can change When `subj` and `attr` are used separately, we only know that `attr` is an attribute

4 Compiler Description

The Taxis compiler design [Nixon, 1983] [Chung, 1984] uses a Pascal subset compiler [Rosselet, 1980] as a model The use of S/SL (Syntax/Semantic Language) [Holt, 1982] in specifying the compiler permits a separation of concerns the abstract data structures and operations are

[15] Finding the type of `ent` is fairly simple, due to a language restriction that the "inverse entity" (e g , `ent`) must be a local variable whose type is a data class

specified (in S/SL) independently of the actual implementation (which in our case is written in Pascal) S/SL is a concise and portable language which permits the specification of functions, procedures and basic control-flow constructs (iteration, alternation, procedure call, etc) S/SL is useful not only for parsing, but also for semantic analysis It aids compiler development and maintenance, by providing a simple method of modifying both syntax and the order of processing It makes the task of designing fairly large compilers more manageable by providing some support for dividing a compiler into modules However, S/SL does not support scope rules, import and export lists for modules, full abstract data types, or formal specifications We found we often had to place constraints in the implementation-language which might be more suitably specifiable at the S/SL level Examples include stating the inter-dependencies between two modules, or that a procedure did not modify a data structure

Another difficulty we encountered, which is quite independent of our use of S/SL, was that it was generally hard to cleanly specify, in orthogonal modules, concepts which are pervasive throughout the language being compiled For example, specialisation (isA) has an impact on most features of Taxis, and had to be considered not only in the isA module, but also in modules for other components, such as those for class and attribute definitions

Now we turn to the design of the compiler, which consists of a parser, and two semantic analysis passes

The parser ensures that the source programme is syntactically correct Error recovery in the parser exploits the organisation of Taxis declarations as *classes* with *categories* containing *attributes* Recovery from syntactic errors can occur within each of these three levels As a result, error recovery occurs by the end of the class declaration which contains the syntactic error (and often sooner), except in the case of missing "end" tokens The parser produces a parse stream in a form that is suitable for processing by subsequent passes For example, all distinct identifiers used in the programme are uniquely labelled Also, identifier names are made available to subsequent passes (for error reporting) and to the run-time system (e g , to support a user interface for schema queries)

The first semantic analysis pass statically ensures that the data model constraints (such as the Schneider constraint) are indeed satisfied The pass builds data structures for class and attribute definitions, and for isA and instanceOf relationships, it also performs attribute inheritance and statically checks that attribute specialisation is done consistently Note that all class definitions, be they built-in or user-defined, are treated in a uniform manner The resulting data structures are then examined in the subsequent pass of the compiler (to ensure correct usage of defined items) and by the run-time system (whenever definitions are needed)

The second semantic pass receives expressions from the first pass, and performs type checking using the data structures created in the previous pass All expressions (whether they involve built-in operations, data access routines or programmer-declared transactions) are standardised into a single format, to allow uniform checking and a single run-time paradigm of accessing all information via attribute value selection with exception handling In addition, the specification of exception handlers in expressions is processed in a manner which permits run-time access to handler specifications with very little overhead (cf [Liskov, 1979])

Once the sequence of attribute value selections has been determined, it is transliterated into code (in our case, Pascal code) which allows procedure calls to be handled as outlined in section 3 2

Throughout the implementation, we aim for *portability* of the compiler and of the code it produces, and *retargetability* to standard relational database systems To this end, the compiler uses and emits a restricted subset of standard Pascal In addition, the relational database interface is written in a layered manner, so that system dependencies are captured at the lowest level, furthermore, only a limited number of relational operators are used

Generated code needs to be associated with considerable amount of run-time support code which provides standard routines (arithmetic, input-output, etc) and an interface to relational database manipulation routines which operate on the relations described in previous sections The built-in routines are themselves best described as Taxis transactions (e g , creating an entity has several `prerequisites`, several `actions`, and can raise exceptions) In view of the large number of such routines (over 100) and associated exceptions, it was considered most feasible to automatically create entries in the data structures for the built-in operations by boot-strapping, i e , writing most run-time routines in Taxis, and then compiling them (with some minimal modification to the resulting Pascal routines)

Here are some statistics on the size of the compiler and run-time support code

COMPILER	*Lines of code*		
	S/SL	*Pascal*	*Total*
Parser	2 000	2 000	4 000
First Semantic Pass	7 000	15 000	22 000
Second Semantic Pass	10 500	5 500	16 000
Total	19 500	22 500	42 000

RUN-TIME SUPPORT	*Lines of code*		
	Taxis	*Pascal & C*	*Total*
Built-in operations	3 000	1 000	4 000
Database interface		5 000	5 000
Total	3 000	6 000	9 000

5 Conclusions

Development of the Taxis compiler addresses issues that are of concern to a variety of researchers, including the implementors of semantic data models, knowledge-base

managers, and developers of compilers for object-oriented languages

From a purely data point of view, the core problems of representing entities, relationships and class hierarchies in a persistent store have also been dealt with in such pioneering efforts as the DAPLEX database system [Chan, 1982] (discussed in Section 2), as well as object managers developed for Artificial Intelligence applications (e g , [Balzer, 1984]), and for SMALLTALK–80 (e g , [Maier, 1986]) The significant novel issues considered in this paper have been the run-time use of meta-information, the more general iteration facilities and the multiple inheritance available in the Taxis data model We have, however, not taken into consideration issues of recovery and concurrency, which are of course very significant, and which have been studied in such seminal papers as [Atkinson, 1983]

From the point of view of procedural data manipulation, the work on compiling object oriented language such as SMALLTALK–80 [Goldberg, 1983] and the various descendants of Simula, has considered such problems as the specialisation of procedures and type checking in the presence of class hierarchies However, previous work has assumed specialisation of procedures along only *one*, distinguished parameter — the receiver of the "message" — while our work addresses specialisation according to several parameters, as well as multiple inheritance In addition, exception handling had not been treated as part of object-oriented languages until now

Perhaps the single most interesting aspect of the research reported in this paper is the integration of these features in a single linguistic framework and the development of an implementation that deals with all of them

The general aim of our approach in developing the Taxis compiler has been to maintain expressive power as much as possible for Taxis while offering reasonable run-time performance The compiler itself was written in a way that handles the different constructs of Taxis in a uniform way This was facilitated by the uniformity of Taxis itself

In addition to the features discussed above, Taxis offers *scripts* for modelling long term processes such as employing someone (a process that begins when a person is hired and ends when his employment is terminated) A script [J Barron, 1982] is built around a Petri net skeleton of states connected by transition arcs, which are augmented by activation conditions and actions to be carried out if a transition fires Scripts have been integrated completely into the Taxis framework, so that script classes are organised into an isA hierarchy according to their generality/specificity, have their states and transitions defined in terms of attributes, and their instances can be accessed through the same facilities used to access instances of data classes This allows, among other things, queries concerning the currently executing set of scripts However, the current implementation of the compiler does not support this feature, though a design of the data structures and algorithms required has been studied in [Chung, 1984] See also [Stonebraker, 1986] for an implementation of active databases, with somewhat different design goals

The compiler described here is only a prototype It does not allow separate compilation of classes, a feature that would be very useful in developing large systems Moreover, the compiler has not been integrated with a Taxis programme development environment that was built independently [O'Brien, 1982] [O'Brien, 1983] [Park, 1984] [Mylopoulos, 1986] and was implemented in LISP Finally, and perhaps most importantly, the compiler has not been tested yet to study its performance characteristics and to determine areas for improvement

In our view, semantic data models are a first attempt to import Knowledge Representation concepts to Databases We expect new waves of imported concepts as they are developed by Artificial Intelligence research and as their need is acknowledged in Database research For this cross-fertilisation to be successful, performance issues for knowledge representation languages must be addressed using the kinds of analytic tools and implementation techniques that were developed for database management systems

Acknowledgements

We acknowledge the contributions of Geoff Loker, Margaret Orzeszak and Jimmy Lee in the implementation of the compiler, and all members, past and present, of the Taxis project for helpful discussions We also wish to thank Grant Weddell, Ric Holt, Ken Sevcik and Joachim Schmidt for helpful advice This project has been supported by a three-year Strategic Grant from the Natural Sciences and Engineering Research Council of Canada

Bibliography

[Abrial, 1974] J R Abrial, Data Semantics In J W Klimbie and K L Koffeman (Eds), *Data Base Management* Amsterdam North-Holland, 1974, pp 1–59

[Albano, 1985] Antonio Albano, Luca Cardelli and Renzo Orsini, Galileo A Strongly Typed, Interactive Conceptual Language *ACM TODS*, Vol 10, No 2, Aug 1985

[Atkinson, 1983] M P Atkinson, P J Bailey, K J Chisholm, W P Cockshott and R Morrisson, An approach to persistent programming *Computer Journal*, Vol 26, No 4, November 1983

[Balzer, 1984] Robert Balzer, Neil Goldman and Bob Neches, Specification-Based Computing Environments for Information Management *International Conference on Data Engineering*, Los Angeles, CA, April 1984 IEEE Computer Society Press, pp 454–458

[D Barron, 1981] David William Barron (Ed), *Pascal — The Language and its Implementation* Chichester John Wiley & Sons, 1981

[J Barron, 1982] John Barron, Dialogue and Process Design for Interactive Information Systems Using Taxis *Proceedings, SIGOA Conference on Office Information Systems*, Philadelphia, PA, 21–23 June 1982, *SIGOA Newsletter*, Vol 3, Nos 1 and 2, pp 12–20

[Borgida, 1985] Alexander Borgida, Features of Languages for the Development of Information Systems at the Conceptual Level *IEEE Software*, Vol 2, No 1, January 1985, pp 63–73

[Chan, 1982] Arvola Chan, Sy Danberg, Stephen Fox, Wen-Te K Lin, Anil Nori and Daniel Ries, Storage and Access Structures to Support a Semantic Data Model *Proceedings, Eighth International Conference on Very Large Data Bases*, Mexico City, Sept 8–10, 1982, pp 122–130

[Chen, 1976] Peter Pin-Shan Chen, The Entity-Relationship Model — Toward a Unified View of Data *ACM TODS*, Vol 1, No 1, March 1976, pp 9–36

[Chung, 1984] Kyungwha Lawrence Chung, *An Extended Taxis Compiler* M Sc Thesis, Dept of Computer Science, University of Toronto, January, 1984 Also CSRG Technical Note 37, 1984

[Codd, 1979] E F Codd, Extending the Database Relational Model to Capture More Meaning *ACM TODS*, Vol 4, No 4, Dec 1979, pp 397–434

[Goldberg, 1983] Adele Goldberg and David Robson, *SMALLTALK-80 The Language and its Implementation* Reading, MA Addison Wesley, 1983

[Hall, 1976] P Hall, J Owett and S Todd, Relations and entities In G M Nijssen (Ed), *Modelling in database management systems* Amsterdam North-Holland, 1976, pp 201-220

[Hammer, 1976] M M Hammer and D J McLeod, A Framework for Data Base Semantic Integrity *Proceedings, Second International Conference on Software Engineering* San Francisco, CA, 13–15 Oct 1976

[Holt, 1982] R C Holt, J R Cordy and D B Wortman, An Introduction to S/SL Syntax/Semantic Language *ACM TOPLAS*, Vol 4, No 2, April 1982, pp 149–178

[Knuth, 1973] Donald E Knuth, *The Art of Computer Programming*, Vol 3 Sorting and Searching Reading, MA Addison-Wesley, 1973

[Liskov, 1979] Barbara H Liskov and Alan Snyder, Exception Handling in CLU *IEEE Transactions on Software Engineering*, Vol SE-5, No 6, Nov 1979, pp 546–558

[Maier, 1986] David Maier, Jacob Stein, Allen Otis and Alan Purdy, Development of an Object-Oriented DBMS In Norman Meyrowitz (Ed), *OOPSLA '86 Conference Proceedings*, Portland, OR, September 29 - October 2, 1986, *SIGPLAN Notices*, Vol 21, No 11, November 1986, pp 472-482

[Mylopoulos, 1980] John Mylopoulos, Philip A Bernstein and Harry K T Wong, A Language Facility for Designing Database-Intensive Applications *ACM TODS*, Vol 5, No 2, June 1980, pp 185–207

[Mylopoulos, 1986] John Mylopoulos, Alex Borgida, Sol Greenspan, Carlo Meghini and Brian Nixon, Knowledge Representation in the Software Development Process A Case Study In H Winter (Ed), *Artificial Intelligence and Man-Machine Systems*, Lecture Notes in Control and Information Sciences, No 80 Berlin Springer-Verlag, 1986, pp 23-44

[Nixon, 1983] Brian Andrew Nixon, *A Taxis Compiler* M Sc Thesis, Dept of Computer Science, University of Toronto, April 1983 Also CSRG Technical Note 33, May 1983

[Nixon, 1987] Brian A Nixon, K Lawrence Chung, David Lauzon, Alex Borgida, John Mylopoulos and Martin Stanley, *Implementing a Taxis Compiler* Technical Report CSRI-194, Computer Systems Research Institute, University of Toronto, January 1987

[O'Brien, 1982] Patrick O'Brien, *Taxied An Integrated Interactive Design Environment for Taxis*, M Sc Thesis, Department of Computer Science, University of Toronto, 1982

[O'Brien, 1983] Patrick D O'Brien, An Integrated Interactive Design Environment for Taxis *Proceedings, SOFTFAIR A Conference on Software Development Tools, Techniques, and Alternatives*, Arlington, VA, July 25–28, 1983 Silver Spring, MD IEEE Computer Society Press, 1983, pp 298–306

[Park, 1985] Sun G Park, *TAXIED-e Automation of Scripts and User Interface in an Integrated Interactive Design Environment for Taxis* M Sc Thesis, Department of Computer Science, University of Toronto, 1985 Also Technical Note CSRI-39, 1985

[Rosselet, 1980] Alan Rosselet, *PT A Pascal Subset* Technical Report CSRG-119, Computer Systems Research Group, University of Toronto, September 1980

[Schmidt, 1977] Joachim W Schmidt, Some High Level Language Constructs for Data of Type Relation *ACM TODS*, Vol 2, No 3, September 1977, pp 247–261

[Schneider, 1978] Peter F Schneider, *Organization of Knowledge in a Procedural Semantic Network Formalism* Technical Report 115, Dept of Computer Science, University of Toronto, February 1978

[Shipman, 1981] D W Shipman, The functional data model and the data language DAPLEX *ACM TODS*, Vol 6, No 1, March 1981, pp 140–173

[Smith, 1977] John Miles Smith and Diane C P Smith, Database Abstractions Aggregation and Generalization *ACM TODS*, Vol 2, No 2, June 1977, pp 105–133

[Smith, 1983] J M Smith, S Fox and T Landers, *ADAPLEX Rationale and Reference Manual*, Second Edition Computer Corporation of America, Cambridge, MA, 1983

[Stonebraker, 1986] Michael Stonebraker and Lawrence A Rowe, The Design of POSTGRES In Carlo Zaniolo (Ed), *Proceedings of ACM SIGMOD '86 International Conference on Management of Data*, Washington, DC, May 28-30, 1986, *SIGMOD Record*, Vol 15, No 2, June 1986, pp 340-355

[Thomsen, 1984] Christine Stougard Thomsen, Multiple Inheritance in Object Oriented Languages Manuscript, Computer Science Department, Aarhus University, Denmark, June 1984

[Tsichritzis, 1982] D C Tsichritzis and F H Lochovsky, *Data Models* Englewood Cliffs, NJ Prentice-Hall, 1982

[Tsur, 1984] S Tsur and C Zaniolo, An Implementation of GEM — Supporting a Semantic Data Model on a Relational Backend *Proceedings, 1984 ACM SIGMOD Conference on Management of Data*, June 1984, pp 286–295

[Wasserman, 1977] Anthony Wasserman, *Procedure-Oriented Exception Handling* Technical Report 27, Medical Information Science, University of California, San Francisco, Feb 1977

[Weddell, 1987] Grant E Weddell, *Physical Design and Query Optimization for a Semantic Data Model (assuming memory residence)* Ph D Thesis, Dept of Computer Science, University of Toronto, 1987

[Wong, 1981] Harry K T Wong, *Design and Verification of Interactive Information Systems Using TAXIS* Technical Report CSRG-129, Computer Systems Research Group, University of Toronto, April 1981 Also Ph D Thesis, Department of Computer Science, 1983

[Zaniolo, 1983] C Zaniolo, The Database Language GEM *Proceedings, 1983 ACM SIGMOD Conference on Management of Data*, May 1983, pp 207–218