Experimental Classification Facilities for Smalltalk

Phillip M. Yelland,
Intelligent Systems Research Section,
British Telecom Laboratories,
Martlesham Heath,
Ipswich IP5 7RE,
United Kingdom.
Tel: +44-473-648-273.
EMail: phil@imst.bt.co.uk.[1]

Abstract

There have been a number of attempts to blend object-oriented programming languages with techniques commonly employed for knowledge representation in artificial intelligence. In the main, such exercises have entailed the incorporation of rule-based programming ideas into object-oriented languages, or the imposition of object-oriented constructs on logical programming notations.

In this report, we describe a system with a slightly different approach to the problem, which augments an object-oriented language with *term classification* capabilities like those found in KL-One and its successors. We hope to establish that this approach results in a more natural and efficacious integration of conventional object-oriented programming and knowledge representation than has been attained up to now.

1. Introduction

The provenance of the system described in this report was our requirement for a general-purpose information repository or *knowledge base* for use by applications managing telecommunications networks. One example of the sort of application we hoped to support would be a monitor assessing the effects of faults arising in a network upon the network's users. The requirements imposed by this type of application are quite exacting. To begin with, the knowledge base has to store relatively large amounts of fairly simple information—telemetry data in the main, produced by a network's diagnostic hardware. At the same time, storage is required for more complicated forms of abstract information, such as contracts outlining services required by network users. As well as providing storage, the knowledge base is expected to assist the applications by efficiently relating these two forms of information. To do this effectively for the monitoring application, for example, the knowledge base would be called on to produce an abstract overview of the state of the network every five minutes from the (approximately) 30,000 items of telemetry data which would arrive in that period.

Our first steps in producing a knowledge base meeting these requirements involved extending an object-oriented programming system (Smalltalk[1]) with *term classification* facilities like those found in the knowledge representation language KL-One and its succes-

1. This research was conducted under B.T. Corporate funding.

1. More precisely, ObjectWorks®\Smalltalk™ (Release 4) from ParcPlace Systems, Inc.

OOPSLA'92, pp. 235-246

sors. In the next section, we examine our reasons for choosing this approach. Subsequent sections describe the facilities offered by the system and outline its implementation. We conclude by indicating possible directions for further development.

2. Possible Approaches

We elected to make our information store "object-oriented" from the outset. There were a number of reasons for this decision: Firstly, experience in the telecommunications industry has shown that the object-oriented approach is particularly well suited to modelling communications networks—so well suited, in fact, that many of the industry's standards are based on it. Furthermore, users of the knowledge base would be able to use object-oriented analyses of the network domain produced by other researchers in our organization. We also hoped eventually to incorporate an object-oriented database into our design so as to allow ready access to large amounts of structured data and to support features such as concurrency and transaction management.

We acknowledged, however, that many of our prospective users wished to store rather complicated forms of information, and would feel disinclined to encode it using the programming notation offered by most object-oriented programming languages. Many authors ([GN87], for example) have averred that representing complex information using low-level procedural code results in a loss of clarity which can make a system difficult to implement, extend and maintain. What we wanted was a mechanism for *declarative knowledge representation* (in the sense of [GN87]) in object-oriented systems.

In selecting such a means of knowledge representation, one of our primary concerns was to minimize the distinction which the users of the knowledge base would have to make between information encoded as native objects and information encoded using the knowledge representation mechanism. We wanted our users to be able to translate information easily between the two forms of representation, so that they could experiment in achieving the best balance of clarity and efficiency.

2.1 Knowledge Representation and Object-Oriented Programming

When we came to examine previous attempts to integrate declarative knowledge representation and object-oriented programming,[1] we decided that they could be broadly categorized either as rule-based programming facilities for object-oriented systems, or as combinations of logic- and object-oriented programming (a comprehensive survey of both types of system may be found in [S86]).

Unfortunately, we felt that both of these approaches were at odds with our desire for a "seamless" combination of object-oriented programming and declarative knowledge representation. For example, the control flow in a rule-based programming system is quite different from that in an object-oriented one; whereas control is normally passed explicitly in an object-oriented system by the dispatch of messages, in a rule-based environment, control is seized when certain patterns emerge in a shared "working memory". And whereas objects persistently encapsulate the state of a system, the state associated with rules is normally bound only transiently and shared globally through the working memory. The mechanics of most logic programming languages are even more alien to mainstream object-oriented systems—contrast, for example, Prolog's unification-based parameter passing and backtracking control flow with the more conventional mechanisms found in C++ or Smalltalk. Such considerations prompted us to look for other ways of representing knowledge.

2.2 KL-One and Term Classification

The techniques we actually decided to use in the information store originated with the KL-One knowledge-representation system. These techniques—described collectively as *subsumption* or *term classification*—have been studied extensively since the introduction of KL-One. Summaries of work in this area may be found in [M91]. We were particularly impressed by the natural way in which it appeared that

1. We have deliberately omitted discussion of *frame-based* systems because for our purposes, their declarative knowledge-representation capabilities do not differ significantly from those of object-oriented programming systems.

term classification facilities could be amalgamated with an object-oriented programming system.

In fact, the degree of commonality between term classification and object-oriented languages has been noted by researchers in the field—most notably in [P-S90]. In addition, there have been projects in the past which have sought to combine term classification with object-oriented programming, at least to some degree. KL-One itself, for example, made provision for attaching procedures and data to items of information (see [BS85]). KloneTalk—an implementation of KL-One in an early version of Smalltalk—is described in [F80]. The "boolean classes" of [MZ86] build a restricted form of term classification into object-oriented programming languages. There are also systems like Login [A-KN86], which unify term classification and other types of programming languages. To our knowledge, however, there have been no attempts at a comprehensive integration of term classification and object-oriented programming on the scale of that described here.

Our first step in constructing the knowledge base was to try and verify our initial suppositions by combining the Smalltalk object-oriented programming environment with term classification mechanisms. The the remainder of this report describes the resulting system, and we begin our account by detailing some of the classification facilities it currently offers.

3. Term Classification Facilities

From the user's point of view, the prime contributions of the term classification facilities are special kinds of classes called *concepts*, which describe their instances. Unlike conventional Smalltalk classes, whose instances are determined explicitly by the programmer when those instances are created, the instances of concepts can be determined automatically by comparing existing objects with the descriptions given by concepts. This has two main consequences:

- Since an object may match the descriptions of more than one concept simultaneously, it may have more than one class.[1] Additionally, because the instance descriptions of concepts may specify aspects of an object's state, a change in the state of an object may lead to revision of its set of classes.
- Some of the sub- and superclasses of concepts can be determined automatically. This is because the system may conclude that a concept c_1 is a subclass of concept c_2 (that c_2 *subsumes* c_1 in the parlance of term classification) if c_1's instance description describes a subset of those objects described by the instance description of c_2. (Note that in general, this may entail some form of multiple inheritance.)

Much of the power of term classification mechanisms in an object-oriented programming system arises from the way in which they combine with the native facilities of the host—in particular, with inheritance, method binding and enumeration of instances.[2] The ability to deduce subsumption relationships between such classes, allied with conventional object-oriented inheritance, allows the system to apply knowledge about more general concepts to more specific cases, as described in [BS85]. Also, when the classes of an object change, the effect of sending messages to it may alter too, as the method bindings specified by its classes are modified. This permits a form of "structured rule-based programming" (like that described in [SN86]), where a concept's description represents a rule pattern ("matched" when objects are discovered to be instances of the concept) and its associated methods act as rule bodies ("fired" by ordinary message-sending). In addition, term classification can be used to answer queries which are posed by defining classes whose instance descriptions embody them. Enumerating the instances of such a class produces all the objects which satisfy the query it represents (an example is given in [P-SBL84]).

The next two sections outline the central characteristics of the term classification facilities (from now on normally referred to collectively as "the classifier")—viz. the definition of concepts and the manipulation of

1. Multiple class membership (objects with more than one class) is a feature of some object-oriented programming systems without term subsumption facilities [S91].

2. Accounts such as [A91] show how a term classification system can prove very useful in its own right; in this report we have chosen to emphasize the synergies which result from the combination of term classification and object-oriented programming.

their instances.

3.1 Describing Objects

Objects appear to the classifier as individuals which possess a number of *roles*. In general, these roles are sets containing other objects. (Note that the roles of an object are distinct from its instance variables, which are normally of no interest to the classifier.) Concepts describe their instances by specifying a number of *restrictions* which their instances' roles must satisfy. In the following, we will show how roles and role restrictions are constructed, how restrictions can be combined, and how they are used in forming concepts.

Defining Roles

Two types of role may be associated with an object. *Primitive roles* are sets whose contents are determined explicitly by adding or removing elements. The contents of *defined roles*, on the other hand, are derived implicitly from the contents of other roles of the object. This is achieved in general by forming the intersection of a set of roles, and then selecting those elements of the intersection which are members of given collections or instances of given classes. Roles are usually given global names, and instances of more than one concept may have roles with the same name.

As an example of role-definition, imagine that in modeling some part of a communication network, we decide to declare a primitive role Circuits, intended to record the connections available to a given element of the network.[1] We could do this by evaluating the following expression:

```
PrimitiveRole name: #Circuits
  category: 'Elements-Roles' [2]
```

Next, assuming that the concept LiveConnection describes objects representing viable connections in a network, and that the role TrunkConnections holds all of the trunk connections of a network element, we can declare a defined role which contains all those circuits of a network element which are also live trunk connections:

```
DefinedRole name: #LiveTrunkCircuits
  subroleOf: Circuits, TrunkConnections
  restrictions: LiveElement
```

Role Restrictions

The experimental classifier recognizes role restrictions of two sorts:

- *Cardinality restrictions* limit the size of an object's roles.
- *Value restrictions* constrain the types of object which may occur in the roles of an object.

Cardinality restrictions restrict the size of a role by imposing a lower- and/or an upper-bound on it. For example, an object with at least thirty circuits would satisfy the restriction:

```
Circuits atleast: 30
```

One with at most forty circuits would satisfy:

```
Circuits atmost: 40
```

Value restrictions compel the objects in a role to be instances of a given Smalltalk class (or one of its subclasses), instances of a concept (or one of its subclasses), or members of a Smalltalk Collection. For example, assuming that an object has a role State, we might wish to have its state described by the tokens #operational and #nonOperational:

```
State all: (Set
              with: #operational
              with: #nonOperational)
```

For the user's convenience, there is provision for declaring a given Smalltalk collection as an AttributeSet, so that it can be referred to by name. For example, we might declare:

```
AttributeSet name: #OperationalState
  elements: (Set
               with: #operational
               with: #nonOperational)
```

—so that the role restriction above could be expressed:

```
State all: OperationalState
```

Alternatively, we might want a rather broader description of state which could be any Smalltalk String:

1. The actual definitions of network entities used in this report are purely illustrative, and not intended to represent elements of a realistic network model.

2. The specification of a category is used to determine how the role is displayed by system browsers; we will omit categories in future declarations.

State all: String

We could also specify that all circuits of an object are members of the concept LiveElement (or one of its subclasses):

Circuits all: LiveElement

Compound Role Restrictions

Role restrictions are combined by conjoining them with the ',' (comma) operator. Thus an object with between 30 and 40 circuits, all of which were live, would satisfy the conjunction:

(Circuits atleast: 30),
(Circuits atmost: 40),
(Circuits all: LiveElement)

The system provides a simple "macro" facility to allow restrictions to be combined conveniently. For example, one could define:

r between: *l* and: *u*	≡	(*r* atleast: *l*), (*r* atmost: *u*)
r exactly: *n*	≡	*r* between: *n* and: *n*
r = *e*	≡	(*r* all: (Set with: *e*)), (*r* exactly: 1)
r memberOf: *c*	≡	(*r* exactly: 1), (*r* all: *c*)

Concepts

A concept is defined by giving a set of classes which must be among its superclasses and a set of restrictions which its instances must satisfy. By consolidating its restrictions with the descriptions of its explicitly-specified superclasses, the classifier arrives at a comprehensive description of the concept's instances which can then be used to find other superclasses and subclasses. As with roles, the classifier distinguishes *primitive* and *defined* concepts.

Primitive concepts give a description of their instances which is accurate but not necessarily exhaustive. That is to say that while every instance of a primitive concept must satisfy its description, not every object which satisfies its description is necessarily one of its instances. One implication of this is that while the system can automatically determine the superclasses of a primitive concept, it must be told explicitly if a concept is to be a subclass of a primitive concept. Another implication is that the system must also be expressly informed if a primitive concept is to be one of the classes of an object or a superclass of one of those classes.[1] In our system, primitive concepts may define instance-variables for their instances, just like Smalltalk classes. As an example, a declaration of the primitive concept NetworkElement is given below; it is a subclass of Object, and each of its instances has a single Integer Identifier, a single State which is a member of OperationalState, and a private instance variable named "comment":

```
PrimitiveConcept name: #NetworkElement
  subclassOf: Object
  restrictions: (Identifier memberOf: Integer),
                (State memberOf: OperationalState)
  instanceVariableNames: 'comment'
```

In contrast to primitive concepts, the descriptions of *defined concepts* describe their instances precisely; every instance of a defined concept must satisfy its description, and every object which satisfies its description is an instance of it. This means that the system is able to deduce both the sub- and superclasses of a defined concept, and is able to conclude that a given object is one of its instances without being told so expressly. The experimental classifier does not permit a defined concept to declare any additional variables for its instances; all must be inherited from its superclasses.[2] The concept LiveElement referred to above might be defined simply as a subclass of NetworkElement, all of whose instances have a State which is #operational:

```
DefinedConcept name: #LiveElement
  subclassOf: NetworkElement
  restrictions: (State = #operational)
```

3.2 Using Objects with the Classifier

Next, we examine how the classification system may be used to create objects and monitor changes in their state.

Creating Objects

Instances of concepts are created—just like instances

1. Henceforth, we abbreviate "superclass of a class of an object" to "superclass of an object".

2. In conjunction with the restrictions on class changes described below, this stricture obviates the need to restructure an object each time its set of classes is altered.

of any other Smalltalk class—by sending a message (generally new) to the concept in question.[1] Complications arise from the dependence of an object's classes on its state. For example, it is possible that some oversight by the programmer might lead to objects newly created by the defined concept LiveElement declared above having a State which was not equal to #operational and thus did not satisfy the instance description of the concept. This might lead to class LiveElement somewhat paradoxically producing new objects which were not legally its instances. At the moment, the default implementation of the message new in concepts takes steps to detect such anomalies by sending the message initialize to newly created objects (the method for initialize is supplied by the creator of the concept), and then finding their classes on the basis of their resulting state. Should the concept not be amongst the classes of a new object after this initial classification, an exception is raised.

Changing the Classes of an Object

Once an instance of a concept has been created, there are constraints on the way in which its set of classes may change throughout its life. Recall from the discussion in the previous section that the description which a primitive concept gives for its instances is not complete. We noted that this implied that the classifier had to be told explicitly if a primitive concept was to be one of the classes (or superclasses) of an object. In the actual system, there is no way of explicitly changing the classes of an object once created—all class changes must occur as a side-effect of changes in the object's state. Thus a primitive concept can only be made a (super-)class of an object by creating the object as an instance of it (or one of its subclasses) in the first place.

The classification system also enforces the obverse of this restriction: once a primitive concept has been made a class or superclass of an object, it must remain amongst that object's classes or superclasses throughout its lifetime. If the object changes so that it no longer satisfies the instance description of the primitive concept, the system raises an exception. This allows the restrictions in the description of a primitive concept to be used as integrity constraints for its instances. If, for example, an instance of the primitive concept NetworkElement defined in the previous section were to try to record a State which was not an element of OperationalState, then it would cease satisfying the restrictions imposed by that concept's description. Since the system cannot remove NetworkElement from the object's classes, it raises an exception instead, indicating a violation of the constraint. The classifier can be used in this way as a form of run-time type system for objects, as described in [BS85].

Modifying Objects

As far as the classifier is concerned, all modifications of objects occur as a result of altering the contents of roles (changes to instance variables go unnoticed). Roles implement the protocol of Smalltalk's abstract class Collection, so their contents may be altered using standard messages, such as add: or remove:. A role is retrieved simply by sending its owner a message consisting of its name with the initial letter in lower case (thus: aNetworkElement state). Classification of an object (that is, adjusting its classes to reflect its current state) is invoked explicitly by sending it the message classify. This permits complex role modifications to take place without incurring the overhead of classification at each step, and also allows an object to temporarily violate integrity constraints between classifications.

The system automatically maintains the relationships between an object's primitive and defined roles determined by their definitions. For example, recall the definition of the defined role LiveTrunkCircuits:

```
DefinedRole name: #LiveTrunkCircuits
  subroleOf: Circuits, TrunkConnections
  restrictions: LiveElement
```

Executing the statement "anExchange liveTrunkCircuits add: aCircuit", the system will first verify that the element added to the role (aCircuit) is actually live (i.e. that its LiveElement is one of its classes or superclasses)—if not, an exception will be raised. The element is then added to role LiveTrunkCircuits, and since the latter is defined as the intersection of Circuits and TrunkConnections, the element is added to these roles

1. Recall that as their state changes, objects may later be found to be instances of classes other than their creator.

too. Conversely, if a LiveElement is removed from Circuits or TrunkConnections, the system ensures that it is also removed from LiveTrunkCircuits.

Automatic Classification

It is possible for an object to be classified automatically by the system, as a result of the classification of elements of its roles. To see how this might come about, consider the following definitions, which supplement the declarations of NetworkElement, Circuits and LiveCircuits given above:

```
DefinedConcept name: #DeadElement
  subclassOf: NetworkElement
  restrictions: (State = #nonOperational)

DefinedRole name: #DeadCircuits
  subclassOf: Circuits
  restrictions: DeadElement

PrimitiveConcept name: #Exchange
  subclassOf: NetworkElement
  restrictions: (Circuits all: NetworkElement)
  instanceVariableNames: ''

DefinedConcept name: #OperationalExchange
  subclassOf: Exchange
  restrictions: (Circuits all: LiveElement)

DefinedConcept
  name: #PartiallyOperationalExchange
  subclassOf: Exchange
  restrictions: (DeadCircuits atleast: 1)
```

These declarations introduce the concept of an Exchange (an object with Circuits which are NetworkElements), with subclasses OperationalExchange (all of whose Circuits are in an operational state) and PartiallyOperationalExchange (with at least one non-operational circuit).

Imagine the situation depicted in figure 1. Here we have an OperationalExchange object with one circuit—a LiveElement whose State is #operational. Now change the State of the LiveElement to #nonOperational and re-classify. This changes the class of the element to DeadElement. At this point, the system observes that the exchange object no longer qualifies as an OperationalExchange; on the contrary, it is now a PartiallyOperationalExchange. The system alters its class accordingly, with the result illustrated in figure 2.

Since the change in the class of the exchange is carried out at the instigation of the system, and not of the programmer, the object is notified of its re-classification. At present, this involves sending it the message classifiedDueTo: *after* re-classification, the single parameter being the element whose modification caused the re-classification. By supplying suitable methods for this message in concepts, a form of "data-driven" or "forward-chaining" inference [M88] can be implemented, with the consequences of modifications to objects being propagated through the system automatically.

4. Implementation

This section briefly sketches the implementation of our

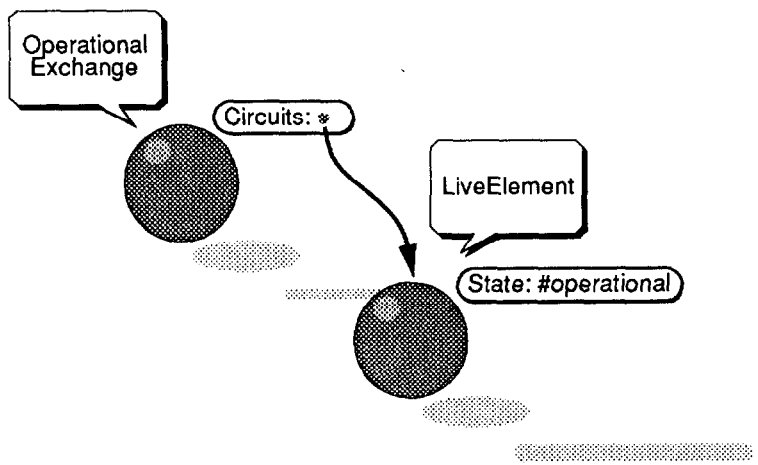

Figure 1: Initial Configuration

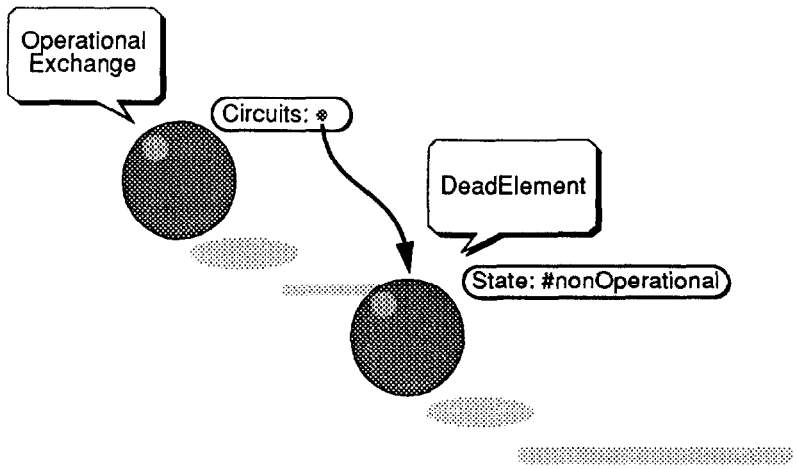

Figure 2: After Re-classification

experimental classifier by looking at the process of constructing and using concepts and their instances. This process can be divided roughly into five phases. In the first phase, entities such as roles, attribute sets and concepts are defined. During the second phase, these entities are classified to create hierarchies reflecting relationships between them—the most important of these is the hierarchy of concepts, which encodes subsumption relationships in the form of sub- and superclass references. Next, the concept hierarchy is extended by adding extra information to improve run-time performance. Inheritance information is then computed. Finally, in the "run-time phase", instances of concepts may be instantiated and modified. We will expand on each of these phases in turn below. (In most of the phases, concepts, roles and attribute sets are all subject to very similar processes; for convenience's sake we will often use the term "concept" to stand for all three types of entity.)

Definition

The mechanics of this phase are fairly straightforward. Since, as we declared at the outset, we wanted to integrate the classification system as closely as possible with Smalltalk, we took great pains to express the definitions of roles, concepts, etc. in standard Smalltalk syntax, avoiding any specialized extensions. Thus—as is the case with conventional Smalltalk classes—defining them is merely a matter of dispatching the appropriate messages. Even the restrictions used in the instance descriptions of concepts are produced by ordinary message sends.[1]

Classification

The next phase of the construction process involves classifying the definitions supplied by the user. Before this can take place, the definitions must be reduced to a canonical format or *normal form*. Such reduction is fairly standard in classification systems (an example is described in [PSKQ89]). Classification involves taking the normal form expression of a concept and comparing it with the normal forms of concepts already defined to determine all of its sub- and superclasses. Lack of space forbids a full description of the algorithms we use for classifying concepts—in any case, they are for the most part standard, and combine (amongst others) ideas from [M83], [PSKQ89], [N90] and [W91].

The classification algorithms are fairly efficient, typically examining only a small proportion the concepts in the system, and testing for subsumption with even fewer. This means that they can be used at run-time, for example, to conduct queries in the manner described in section 3. They are not, however, sufficiently

1. The ability to produce restrictions in this way allows the macros illustrated in section 3 to be implemented simply as methods in the appropriate classes.

fast that they can be used directly to manage the classification of objects (as has also been observed in [M88]). To do the latter efficiently, we first need to supplement the hierarchy of concepts with additional information, described next.

Augmenting the Concept Hierarchy

The processing in this phase attempts to eliminate two sources of run-time inefficiency:

- The need to compute the method-bindings of objects with changing classes.
- Searching for the new classes of objects during classification.

The strategy we use (which is fore-shadowed in [W91]) involves computing the *conjunctions* of programmer-defined concepts and then adding them to the hierarchy. The conjunction of a set of concepts is simply a defined concept which specializes all of the concepts in the set, without imposing any additional restrictions of its own. Essentially, the conjunctions added to the concept hierarchy represent the possible sets of classes which instances of concepts might possess. Thus at run-time, all of an object's classes may be represented by a single concept. By computing inheritance information for conjunctions, we eliminate the need to determine the method-bindings for objects at run-time. And adding the conjunctions to the hierarchy also means that once we know that an object is an instance of a given class, all of its other classes may be found simply by examining the subclasses of that class.[1]

Of course in theory, adding all possible conjunctions to the concept hierarchy may increase its size exponentially. In practice, however, the degree of expansion is considerably limited a number of factors, most of which ensue from the restrictions on the changes in objects' classes described in section 3.2:

- No concept needs to be conjoined with one of its sub- or superclasses.[2]
- Let the *primitive set* of a concept C be the comprising C's primitive superclasses and C itself, should it be primitive. It is not difficult to show that each conjunction formed must have a primitive set equal to that of at least one of its components.[3]
- All *incoherent* conjunctions which cannot logically describe any real object (such as the conjunction of LiveElement and DeadElement as they are defined above) can be rejected.

Computing Inheritance Information

Once the concept hierarchy has been expanded as described above, computing the information (instance variables and method bindings) inherited by concepts follows much the same lines as are followed in many other object-oriented programming environments with multiple inheritance, such as Trellis/OWL [SCBKW86], Extended Smalltalk [BI82] or CLOS [K89]. Some of these environments resolve possible inheritance conflicts using heuristics based upon the order in which the programmer declares the superclasses of classes. Unfortunately, the classifier computes the superclasses of a concept automatically, largely depriving the programmer of the opportunity to supply this sort of information. One response to this might be to adopt the approach employed in Extended Smalltalk and Trellis/OWL, which requires the user to resolve all possible inheritance conflicts. Prior experience with Extended Smalltalk gave us to believe that this would be unduly burdensome for our users. Therefore, the current version of the classifier uses Touretzky's inferential distance heuristic [T86] for conflict resolution.[4] Since this does not resolve all inheritance conflicts, any which remain are signalled to the user using the system browser, as in Extended Smalltalk.

We have made no attempt to address other issues,

1. For we know that if object o is an instance of class C (which we write "$o \in C$"), then for all other classes D, if $o \in D$, then by definition, $o \in C \wedge D$, where $C \wedge D$ represents the conjunction of classes C and D. However, by definition, we also know that $C \wedge D$ must be a subclass of C.

2. Since if for all objects o, $o \in C \Rightarrow o \in D$, then for all objects o, $o \in C \wedge D \Leftrightarrow o \in C$.

3. Otherwise, it would be impossible for an object whose classes are described by any of the components to adopt the classes represented by the conjunction itself. This circumscription prohibits, for example, the conjunction of two or more primitive concepts.

4. Roughly paraphrased, Touretzky's heuristic states that if a class C might inherit conflicting attributes from superclasses A and B, and A is a subclass of B, then the attribute specified by A should be chosen for C. We believe that a similar heuristic was used to resolve conflicts in a multiple-inheritance system for Smalltalk-80 produced at Tektronix.

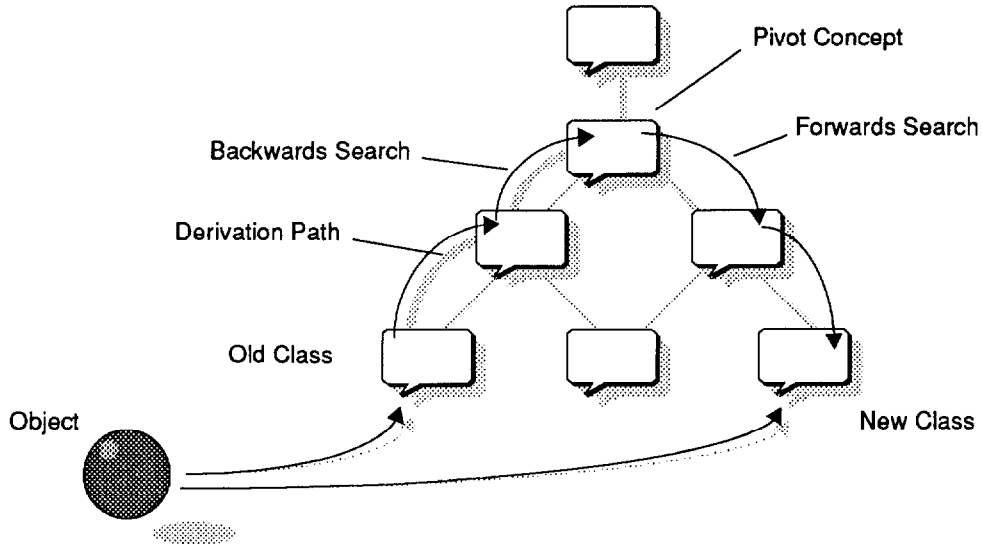

Figure 3: Object Classification

such as the aliasing effects which result from the "graph-based" implementation of multiple inheritance we chose to use (see [Sn91] for more details). Only experience with realistic applications will indicate whether our inheritance mechanisms are suitable, and we have deliberately made them "pluggable" so that they can be conveniently replaced if necessary.

Run-Time

The main activity of the classifier at run time is classifying the instances of concepts, adjusting the classes of such objects in the light of modifications of their roles (the addition, removal or classification of elements). As we pointed out above, adding conjunctions to the concept hierarchy means that all possible combinations of object classes are represented by single concepts. Therefore, though an object may appear to the user to have more than one class, the implementation represents them all with a single concept (so that for the rest of this section, we refer to the "class", rather than "classes", of an object). In fact by the definition of conjunctions, it is easy to show that run-time classification of an object amounts to searching the (expanded) concept hierarchy for the most specific concept whose description the object matches. The details of this search are described below (see figure 3 for an illustration).

The classifier attempts to speed the search for the new class of an object by relying on the "locality" of classification—the supposition that the object's new class is fairly closely related to its old one. It exploits this supposed behaviour by associating a *derivation path* with each object. This is a path through the hierarchy, proceeding from more general to more specific concepts, which records the concepts examined in locating the most recent class of an object. By arranging to begin the derivation path of an object at an appropriate point (normally the most specific primitive superclass of the concept that created it), one may be guaranteed that when the classification of an object is invoked, it should always be possible to find a concept that still describes it simply by tracing backwards along its derivation path. We call the first such concept found the *pivot concept*. If no pivot concept can be found on the path, then a violation of the object's primitive ancestors' integrity constraints is signalled. To speed up the search, we use an intelligent backtracking algorithm like that outlined in [M88]—by examining which particular roles of an object have been modified, the algorithm can often "skip" several concepts along the derivation path, eliminating a number of possibly expensive tests.

Once we have found the pivot concept, the argument advanced in the section above means that the new con-

cept can be isolated simply by examination of its subclasses. This may be carried out by a simple depth-first (backtrack-free) descent of the concept hierarchy. As the search progresses, the concepts traversed are appended to the object's derivation path. The concept found at the end of this search describes the current set of classes for the object. (Again, in order to respect the restriction on changes of primitive ancestors, the search ignores primitive concepts and their subclasses.)

5. Conclusions

We have described a modest extension to Smalltalk which we believe may be useful in expressing at least some kinds of complex information in a reasonably clear and readily-apprehended manner. Our approach to constructing this system has been to take ideas from the field of subsumption- and inheritance-based reasoning, integrating carefully them with the Smalltalk system. We felt that this approach would afford particular flexibility in balancing the relative efficiency of conventional object-oriented programming against the conceptual clarity offered by knowledge representation facilities.

Our investigations are very much on-going; users intend soon to begin applying the system to more ambitious problems in the telecommunications domain, and we expect the system to change (possibly significantly) to accommodate their demands. There are a number of areas in which further development looks especially appealing:

- The implementation of more comprehensive general term classification facilities. These might include those found in existing systems like Classic, or Loom,[1] or less "traditional" capabilities such as are suggested by [DP91].
- The implementation of "specialized reasoning modules", like those described in [LD90] or [MB87].
- Further examination (not touched directly in this report) of the completeness of the classifier used in the system—our current classifier, while sound, does not discover all the subsumption relationships which might be implied by a Tarksi-style model of concepts (see [N90] for a full discussion).
- Embellishments to the system's user interface; at present, the interface to the classifier consists largely of standard Smalltalk tools, with a few minor extensions for displaying concept hierarchies.

Initial experience (mainly, it must be said, in fairly small applications) tends to suggest that on the whole, the classification system substantiates many of our initial hopes. This, and the fact that researchers (in [S91], [C91], and [W86], for instance) have previously expressed the need for facilities in object-oriented environments which the classifier offers (at least to some degree), confirms our belief that classification systems like that described here might make a contribution to the development of object-oriented programming.

Acknowledgments

We would like to thank the management and staff of the Systems Research Division, Intelligent Systems Research Section and Network Management Department of British Telecom Laboratories for their encouragement and support for the project described in this report. We are also grateful to Christof Peltason and the Back project, Peter Patel-Schneider, Bolt Beranek and Newman and the USC/Information Sciences Institute for kindly providing valuable information in its formative stages. We acknowledge the permission of the Director of Network Technology, British Telecom Laboratories, for publication.

6. References

[A-KN86] H. Ait-Kaci, R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, Vol. 3, 1986.

[A91] G. Attardi. An analysis of taxonomic reasoning. *in* [LNS91].

[BBMR89] A. Borgida, R. Brachman, D. McGuinness, L. Resnick. CLASSIC: A structural data model for objects. *Proc. 1989 ACM SIGMOD International Conference on Management of Data.* ACM, 1989.

1. In particular, the ability to "partially describe" an object [BBMR89] (specifying the cardinality of a role, for example, without specifying its elements) should prove useful.

[BI82] A. Borning, D. Ingalls. Multiple inheritance in Smalltalk-80. *Proc. First National Conference on Artificial Intelligence.* AAAI, 1982.

[BS85] R. Brachman, J. Schmolze. An overview of the KL-One knowledge representation system. *Cognitive Science*, 1985.

[C91] E. Cordingley. Analysing texts for knowledge-based systems. *Knowledge-Based Systems and Legal Applications*, T. Binch-Capon (*ed.*), Academic Press, 1991.

[DP91] J. Doyle, R. Patil. Language restrictions, taxonomic classifications and the utility of representation services. *Artificial Intelligence*, 1991.

[F82] R. Fikes. Highlights from KloneTalk. *in* J. Schmolze and R. Brachman (*eds.*), *Proc. 1981 KL-One Workshop.* Rep. 4842, Bolt, Beranek and Newman, 1982.

[GN87] M. Genesereth, N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[K89] S. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Weseley, 1989.

[LD90] D. Litman, P. Devanbu. *Clasp: A Plan and Scenario Classification System.* AT&T Bell Laboratories Report, 1990.

[LNS91] M. Lenzerini, D. Nardi, M. Simi (*eds.*). *Inheritance Hierarchies in Knowledge Representation and Programming Languages.* Wiley, 1991.

[M83] M. Moser. An overview of NIKL, the new implementation of KL-One. *Research in Natural Language Understanding*, Technical Report 5421, Bolt Beranek and Newman, 1983.

[M88] R. MacGregor. A deductive pattern matcher. *Proc. 7th National Conference on Artificial Intelligence*, AAAI, 1988.

[M91] —. The evolving technology of classification-based knowledge representation systems. *in* [So91].

[MB87] R. MacGregor, R. Bates. *The Loom Knowledge Representation Language.* Technical Report ISI/RS-87-188, USC/Information Sciences Institute, 1987.

[MZ86] D. McAllester, R. Zabih. Boolean classes. *Proc. 1986 Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, 1986.

[N90] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems.* Lecture Notes in Artificial Intelligence 422, Springer Verlag, 1990.

[P-S90] P. Patel-Schneider. Practical, object-based knowledge-representation for knowledge-based systems. *Information Systems*, 15(10), 1990.

[P-SBL84] P. Patel-Schneider, R. Brachman, H. Levesque. *ARGON: Knowledge Representation Meets Information Retrieval.* Fairchild Technical Report 654, 1984.

[PSKQ89] C. Peltason, A. Schmiedel, C. Kindermann, J. Quantz. *The BACK System Revisited.* KIT-Report 75, Technische Universität Berlin, 1989.

[SCBKW86] C. Shaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpot. An introduction to Trellis/Owl. *Proc. 1986 Conference on Object-Oriented Programming Systems, Languages and Application*, ACM, 1986.

[S86] J. Saunders. A survey of object-oriented programming languages. *Journal of Object-Oriented Programming*, 1(6), 1989.

[S91] L. Stein. A unified methodology for object-oriented programming. *in* [LNS91].

[SN86] W. Swartout, R. Neches. The shifting terminological space: An impediment to evolvability. *Proc. 5th National Conference on Artificial Intelligence*, AAAI, 1986.

[Sn91] A. Snyder. Inheritance in object-oriented programming languages. *in* [LNS91].

[So91] J. Sowa (*ed.*), *Principles of Semantic Networks.* Morgan Kaufmann, 1991.

[T86] D. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, 1986.

[W86] W. Woods. Important issues in knowledge representation. *Proc. IEEE*, 74(10), 1986.

[W91] —. Understanding subsumption and taxonomy: a framework for progress. *in* [So91].