SERIES: BUI

NO.67
(1987)

BUILDING RESEARCH ASSOCIATION OF NEW ZEALAND

REPRINT

CI/SfB (Ahi) (Agm)
UDC 025.347: 681.3.06

Class language - a language for building expert systems

W.B. Mugridge, J. Hamer and J.G. Hosking

This report originated in work commissioned by BRANZ as part of its ongoing research into knowledge based systems. The views represented are not necessarily those of the Association.

Reprinted from ***Proceedings of the Second New Zealand Conference on Expert Systems,*** Auckland, 2-4 February 1987.

branz
BUILDING RESEARCH ASSOCIATION OF NEW ZEALAND

Class Language - a Language for Building Expert Systems

W.B. Mugridge, J. Hamer, J.G. Hosking
Department of Computer Science
University of Auckland

Abstract

A successful expert system needs to be structured according to a model of the application problem, so that it can be easily developed and maintained, and so that it can provide appropriate explanation to the user. Such structuring requires a language which incorporates a general model of representation. This paper introduces such a language, *Class Language*, and discusses the design of the language in the context of the *FireCode* expert system.

Introduction

Many features have been suggested to distinguish Expert Systems from conventional software (Waterman, 1985; Buchanan, 1986; Fenves, 1986). Such features include: separation of knowledge and control, ease of incremental growth of the system, and provision of explanation to the user (Fenves, 1986). These features make it imperative that the implementation of an expert system is structured around a model of the application problem. This implies that a suitable high-level language is available to represent information, both to aid the developer of the system and to facilitate the generation of explanation suitable for the expert system user.

The need for languages that provide better representational structures has been recognised for some time. Clancey (1985) examines the deficiencies of the uniform rule-based representation used in MYCIN and other such systems, and suggests that many implicit structures should be explicitly represented in these systems. Thompson and Clancey (1986) discuss the benefits of using the language HERACLES, which was designed to overcome some of the deficiencies. Chandrasekaran (1986) also addresses the need for isolating "generic tasks" in knowledge-based reasoning. This focus on structuring is similar to the *structured programming* drive of the 1970's, in which the control constructs, such as conditional loops and procedures, were being made explicit.

Class Language was developed to provide a representation language suitable for implementing the *FireCode* expert system (Buis et al, 1986; Hosking et al, 1987a). However, it has evolved as a general expert system language suitable for applications in which problem-solving based on classification is important. Classification problem solving is useful in a wide range of application areas, such as diagnosis, interpretation, and simulation (Clancey, 1985). The design of *Class Language* was strongly influenced by backward chaining expert systems like MYCIN (Buchanan and Shortliffe, 1984), object-oriented languages such as Smalltalk (Goldberg and Robson, 1983), and the database design theory of Smith and Smith (1977) which focusses on the complementary abstractions of *generalisation* and *aggregation*.

In this paper we introduce the representational and computational constructs provided by *Class Language*, illustrating their use in the *FireCode* system. We provide design rationale for introducing the constructs into the language, and show how the language has evolved and continues to evolve as we gain a better understanding of the development of different types of expert system.

Representation

Knowledge structures can be found independently of any computer system. The *FireCode* system is based on two parts of DZ4226, a draft New Zealand standard for fire safety. Several different sorts of knowledge can be found in the written code, both in its general structure and in the individual clauses (Hosking et al, 1987b).

Any programming language, be it a conventional program or an expert system, requires a means of representing the sorts of objects that arise in a problem and a means of reasoning with or manipulating those objects. The need to encode the forms of knowledge found in DZ4226 in appropriate structures was a partial motivation for the various representational and computational constructs provided in *Class Language*.

Simple Objects

The simplest objects that can be represented in *Class Language* correspond to scalar variables in a traditional programming language, and like them can take atomic values. Simple objects can be numbers, Booleans, symbols, or enumerated values (where the possible values are listed, as in Pascal). For example, a simple *number* object can be used to represent the height of a building:

```
height
  (ask 'What is the height of the building?')
  : number.
```

As illustrated here, *Class Language* permits additional information to be associated with an object, such as a suitable query for the user when seeking a value for a simple object.

Aggregates, Classes and Instances

There is often a need to group objects together into aggregation structures, collecting together related information and the ways of dealing with that information. For instance, a simplistic view of a building could consist of a height, an ordered collection of storeys, and a collection of stairways. A storey could then be defined as a collection of exitways, a height, an area and a number of spaces. A space could introduce further attributes. As shown in this example, aggregates themselves can consist of other aggregates.

In using DZ4226 there is a need to describe a number of different buildings or storeys. This means that we need to separate the description of buildings and storeys in general (the *class* of buildings and the *class* of storeys) from particular buildings and storeys (*instances* of buildings, such as the University Conference Centre and of storeys, such as the ground floor of the University Conference Centre). This distinction between *class* and *instance* comes from Smalltalk (Goldberg and Robson, 1983), and corresponds closely to the concepts of *type* and *variable* in Pascal, and *schema* and *record* in database systems.

The *class* construct of *Class Language* is used for defining both aggregation and generic structures. As an illustration of this, the following is a much simplified version of the class *building* from *FireCode*, along with two instances of that class:

```
class building.
    buildingStorey
      : array (numberOfStoreys) of storey.
    height
      (ask 'What is the height?')
      : number.
    numberOfStoreys
      (ask 'How many storeys are there >')
      : number.
...
endclass.

instance
    conferenceCentre,
    sportsCentre : building.
```

Here any *building* is defined to consist of a collection of *storeys*, the number being defined by *numberOfStoreys*, and a *height*. The instances *conferenceCentre* and *sportsCentre* are then patterned on the general structure defined by the class *building*. The *height* of the *conferenceCentre* may be referred to using the notation *conferenceCentre^height*.

Generalisation

Some classes are specialised versions of others, so it is useful to make this relationship explicit. All spaces in *FireCode* have components, such as the exitway width and the number of occupants. A specialised class of space, however, has extra information and constraints specific to that class. For example, seating is only important in assembly spaces (see Hosking et al, 1987a, Table 1).

Class Language, like Smalltalk, allows classes to be defined as specialisations of another class. For example, the class *spaceAM* defined below is for assembly spaces with 100 - 500 people. It is a specialisation of the class *space*, shown in Fig. 1.

```
class spaceAM.

generalisation space.

private
        seating : seatClass.
        checkSeating
            (ask 'Do you wish to check the seating? ')
            : boolean.

procedure checkSpecialisation.
        if checkSeating then
            call checkSeat of seating.
            endif.
        endprocedure.

rule ruleG6_3B_2
        if seating^seatIsBleacher then
            pathWidth := table^_3A^occupantLoad / 312.5.

endclass.
```

Consider two instances declared as follows:

```
        instance
             seminarRoom : space.
             anotherRoom : spaceAM.
```

The instance *seminarRoom* has components such as *minimumWidth*, as defined in the class *space*. Compare this with the instance *anotherRoom*, which has the components and methods defined in the *spaceAM* class, such as *seating*, as well as those defined in the *space* class. Inheritance of properties down the class taxonomy like this allows common information to be stated once in the most general class and shared with the specialisation classes. Each of the specialisation classes can define further components, highlighting their differences.

As well as avoiding redundancy in the definition of classes, the generalisation hierarchy is also used for the inheritance of computational structures and for the dynamic classification of objects, as described below.

Computation

Smalltalk (Goldberg and Robson, 1983) views an object as a collection of related data, plus the procedures (methods) for accessing and manipulating that data. Both the components and methods are defined in the class, with the restriction that only the procedures of an object can access the components of an object of that class. This notion of *information-hiding* means that changes to the implementation of an object are localised and thus can be hidden from other parts of the system.

```
class space    (numberOfExitways,
        buildingHeight : number.
        finalExitOnStorey,
        hasPartStoreys : boolean).
public
       spaceUseClass : classOfUsage.
       table6_3A : classTable6_3A.
       table6_4A : classTable6_4A.
       minimumWidth
          (ask 'What is the width of the smallest passage (in mm)> '),
       totalFloorArea
          (ask 'What is the total floor area (in square meters)> ')
          : number.
private
       pathWidth : number.
       name (ask 'Enter a name for this space)
          : symbol.
specialisation
       singleExitSpace,
       spaceAL, spaceAM, spaceAO, spaceAS, spaceMXH, spaceSC.
       ...
       classifyAM
          if spaceUseClass^useClass = am
          then classify as spaceAM.
       ...
procedure checkSpace.
       determine name.
       display('Checking ', name).
       summary('Summary for ', name).
       call checkGrading of spaceUseClass.
       call checkSpecialisation.
       ...
       call checkFloorArea of table6_4A.
       if table6_3A^occupantLoad > 150 and spaceDeadEnd then
          display('The space should not contain more than 150',
                  'people, as it is a deadend.').
          summary('The space should not contain more than 150',
                  'people, as it is a deadend.').
          summary('The space has ',table6_3A^occupantLoad,
                  ' occupants.').
          summary('See Clause 6.4.6.1').
          endif.
       ...
       endprocedure.
...
rule ruleG6_3B_1
       if spaceUseClass^useClass = ao then
          pathWidth := table6_3A^occupantLoad / 500.

rule ruleG6_3B_3
       if spaceIsDeadEnd then
          pathWidth := table6_3A^occupantLoad / 75.

rule ruleG6_3B_1
       if true then
          pathWidth := table6_3A^occupantLoad /187.5.
...
endclass.
```

Fig. 1: Simplified version of class *space* from *FireCode*.

A similar view of objects is taken in *Class Language*, where computations are embedded in class definitions. Three computational constructs are available: backward-chaining rules, procedures, and classification rules. As with properties of classes, rules and procedures may be inherited down a class hierarchy.

Rules

The backward-chaining rules of *Class Language* are based on those found in many expert systems, such as MYCIN (Buchanan and Shortliffe, 1984). These rules spell out the values that may be given to scalar objects, and the conditions under which those values are assigned. A rule based approach is especially suitable for complex conditional branching as is found in *FireCode*.

In *Class Language*, rules are interpreted in a backward chaining fashion, so that when a value is required for an object or property the associated rules are tested. For a rule to succeed, its preconditions must be true. Determining the value of preconditions may require further backward-chaining. The backward chaining bottoms out when a rule succeeds by making use of values that are already known or supplied by the user. An example rule taken from the *space* class in Fig. 1 is:

```
rule ruleG6_3B_3
     if spaceIsDeadend
     then pathWidth := table6_3A^occupantLoad / 75.
```

When determining a value for a property of an object, the rules in the class of the object are successively tried; if none are successful then appropriate rules are tried in successively more general classes.

Procedures

Procedures are an integral part of traditional programming languages and are also necessary in any serious expert system language (Hayes-Roth, 1985). For example, in *FireCode*, procedures provide a means of representing the steps that a building designer performs in checking a building. Procedures are appropriate for handling information which lays out the order in which things are done, such as in Clause 6.3.1.1. in *DZ4226* (as shown in Fig. 3. of Hosking et al, 1987a). An example procedure is given in the class *space* shown in Fig. 1.

Procedures were initially not provided in *Class Language*. Instead, procedural information was encoded using backward-chaining rules, where only the side-effects of the rules were significant. Making the procedural reasoning explicit, by introducing procedures into the language, led to a large reduction in the number of rules in *FireCode* and an overall simplification of the system.

Procedures in *Class Language* are limited, compared to an ordinary programming language, in that there is no assignment statement. Backward chaining is invoked automatically whenever a property value is needed, such as for the display of information to the user. Procedures are principally used at the outer level of the expert system, for handling such matters as introductory information, menus of available options, and final material to be displayed to the user.

Classification

The process of classification is often used by humans when discriminating between groups of objects. Classification plays an important role in regulations, where categories are designed to make important distinctions. In *DZ4226*, building spaces are classified by their *use class*, based on such attributes as the use of the space and number and mobility of occupants.

In *Class Language*, each class has information describing how to classify an instance of that class to one of its specialisation classes. This information is in the form of *classification rules*, which are similar in form to backward-chaining rules. The process of dynamic classification takes an object of a general class from that class to its most specific class; this may involve several classifications down the generalisation hierarchy. For example, the following rule from the class *space* shown in Fig. 1, may be used to classify an instance of the class *space* to the class *spaceAM*:

```
classifyAM
          if spaceUseClass^useClass = am
          then classify as spaceAM.
```

An object is classified when the first reference is made to it. For example, when a reference is made to *seminarRoom^pathWidth* in some procedure, classification of the object *seminarRoom* is attempted using the classification rules given in its class, *space*. Testing the precondition of a classification rule may neccessitate backward chaining. Once the instance has been classified, it automatically takes on the components, rules, and procedures of its new specialisation class, as well as retaining those of its original class, i.e. to all intents and purposes it is then treated as if it is an instance of the specialisation class.

Hence, as classification takes place, further information is accumulated about the object, allowing more specific constraints to be gathered. For example, if *seminarRoom* is classified as *spaceAM* it will collect the special rule for *pathWidth* that only applies to assembly spaces. Thus, classes both modularise the knowledge base as a static entity when it is being constructed, and provide focus during the use of the system, through dynamic classification of objects.

Most other object or class-based expert system languages and representational systems make use of the generalisation hierarchy (or lattice) for inheritance only (Fikes and Kehler, 1985; Stefik et al, 1983) and do not make use of dynamic classification. Classification is provided in CSRL (Bylander and Mittal, 1986; Chandrasekaran, 1986), but it occurs in a simple type hierarchy and CSRL does not provide the powerful aggregation features that are complementary in Class Language.

Implementation and Environment

As well as the need for a suitable high-level language for constructing expert systems, it is clear that a productive environment for both the user and knowledge engineer must be provided. Discussion of the environment provided by *Class Language*, including an extensive dialogue, is given in Hosking et al (1987a) in the context of the *FireCode* system.

Class Language is implemented in C-Prolog running under Unix. A VMS/Prolog-1 version has also been developed and is being used by BRANZ. The current implementation is fairly slow, but a compiled version of Class Language is under development which, with the use of a Prolog compiler, should provide a speedup of 10-50 times.

Applications of Class Language

Class Language is suited to applications which are based on a classification problem solving method (Clancey, 1985), where the solutions/goals can be pre-enumerated. This is in contrast to more generative systems based on constraints, where there is a need to search for a solution.

Class Language is also suitable for routine design problems, due to the way that generalisation and aggregation work together. I.e. those in which it is reasonably clear which *refinement* to make to an evolving design. In *FireCode*, classification is used to build up a model of the building to be checked. This mechanism is also appropriate for creating simple designs, in which the design is refined by classification as design decisions are made. *Class Language* used in this way corresponds to the DSPL language (Chandrasekaran, 1986), which provides for routine design problems in which the complications of a constraint-based approach are unnecessary.

A number of expert systems have been developed, or are under development, using *Class Language* as a basis. In addition to *FireCode*, a number of classification based expert systems are being developed by BRANZ including *Damp*, a system to diagnose problems caused by damp in buildings, and *Adhesive* and *Sealant*, for selecting appropriate construction materials (Whitney, 1987). The authors are commencing work on further regulation based systems, including a design-based system for wall bracing. We expect that useful feedback will be provided on the design of *Class Language* from development of these systems.

Future work

Many design issues remain in the further development of *Class Language*. Some extensions to the language are under development, while others need further investigation. A few of the features being investigated are:

Clustering

It is often useful to classify an object in ways which are not mutually exclusive. For example, an important category of space is a deadend, with only one exit. Spaces can be categorised as to whether or not they are a deadend (almost) independently of the categorisation according to use class illustrated above. Smith and Smith (1977) discuss non-mutually-exclusive *clusters* in a static sense in the context of database design. The addition of dynamic classification should provide a powerful modelling mechanism.

Non-hierarchical structures

Class Language structures are currently hierarchical. This is appropriate when the application area is essentially hierachical, but suffers with more complex relationships. Investigation of other structuring methods is being pursued.

Knowledge Engineering Methodology

Class Language incorporates a general model of representation, based on generalisation and aggregation, which suggests a particular approach to the construction of an expert system. For example, an early step in the design of an expert system is to consider the major classes involved in the application. This is followed by designing the generalisation and aggregation structuring of those classes. The generalisation structure, or taxonomy, focuses attention on the classification aspects of the expert system. The development of this approach into a knowledge-engineering methodology is an important aspect of any future work.

Conclusion

Class Language is a language for constructing expert systems which provides a means of integrating the benefits of both aggregation and generalisation in a common framework. The class generalisation structure allows both inheritance and classification of objects within the generalisation taxonomy. The three computational constructs provided (backward-chaining, procedures, and classification) interact to provide a powerful modelling facility. The language is ideally suited for developing classification-based and routine-design-based expert systems.

Acknowledgements

The authors gratefully acknowledge the Building Research Association of New Zealand for financial assistance.

References

Buchanan, B G, 1986: "Expert systems. Working systems and the research literature". *Expert Systems* 3, 1, pp 32 -51.

Buchanan B G and Shortliffe E H (eds), 1984: *Rule-Based Expert Systems The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, Massachusetts.

Buis M, Hamer J, Hosking J G and Mugridge W B, 1986: "An expert advisory system for a fire safety code" *The second Australian conference on applications of expert systems*, (New South Wales Institute of Technology, Sydney) pp92-109.

Bylander T and Mittal S, 1986: "CSRL: a language for classificatory problem solving and uncertainty handling", *AI Magazine*, August, 1986, pp66-77.

Chandrasekaran, B, 1986: "Generic tasks in knowledge-based reasoning: high level building blocks for expert system design", *IEEE Expert*, Fall 1986, pp23-30.

Clancey W J, 1985: "Heuristic classification" *Artificial Intelligence* 27 pp289-350.

Fenves, S.J., 1986: "What is an expert system?" in C.N.Kostem and M.L.Maher (Eds), *Expert*

Systems In Civil Engineering, American Society of Civil Engineers, 1986.

Fikes R and Kehler T, 1985: "The role of frame-based representation in reasoning" *CACM*, **28** pp904-920.

Goldberg A and Robson D, 1983:*Smalltalk-80: The Language and its Implementation* Addison-Wesley, Reading, Massachusetts.

Hayes-Roth F, 1985: " Rule-based systems" *CACM* **28** pp 921-932.

Hosking J G, Mugridge W B and Buis M, 1987a: "Expert systems for regulations and codes" *Proc NZES'87*.

Hosking J G, Mugridge W B and Buis M, 1987b: "FireCode: a case study in the application of expert systems techniques to a design code" *Environment, Planning and Design B*, to be published.

Smith J M and Smith D C P, 1977: "Database abstractions: aggregation and generalization", *ACM Transactions on Database Systems*, **2**, No. 2, 1977, pp 105-133.

Stefik M, Bobrow D G, Mittal S and Conway L, 1983: Knowledge programming in loops: report on an experimental course" *AI Magazine* (Fall) pp 3-13.

Thompson and Clancey, 1986: "A qualitative modelling shell for process diagnosis" *IEEE Software* March 1986.

Waterman D A, 1986: *A Guide to Expert Systems*, Addison-Wesley, Reading, Massachusetts.

Whitney R, 1987: "Knowledge-based systems for building technology: strategic aspects" *Proc NZES'87*.

ISSN 0111–7459