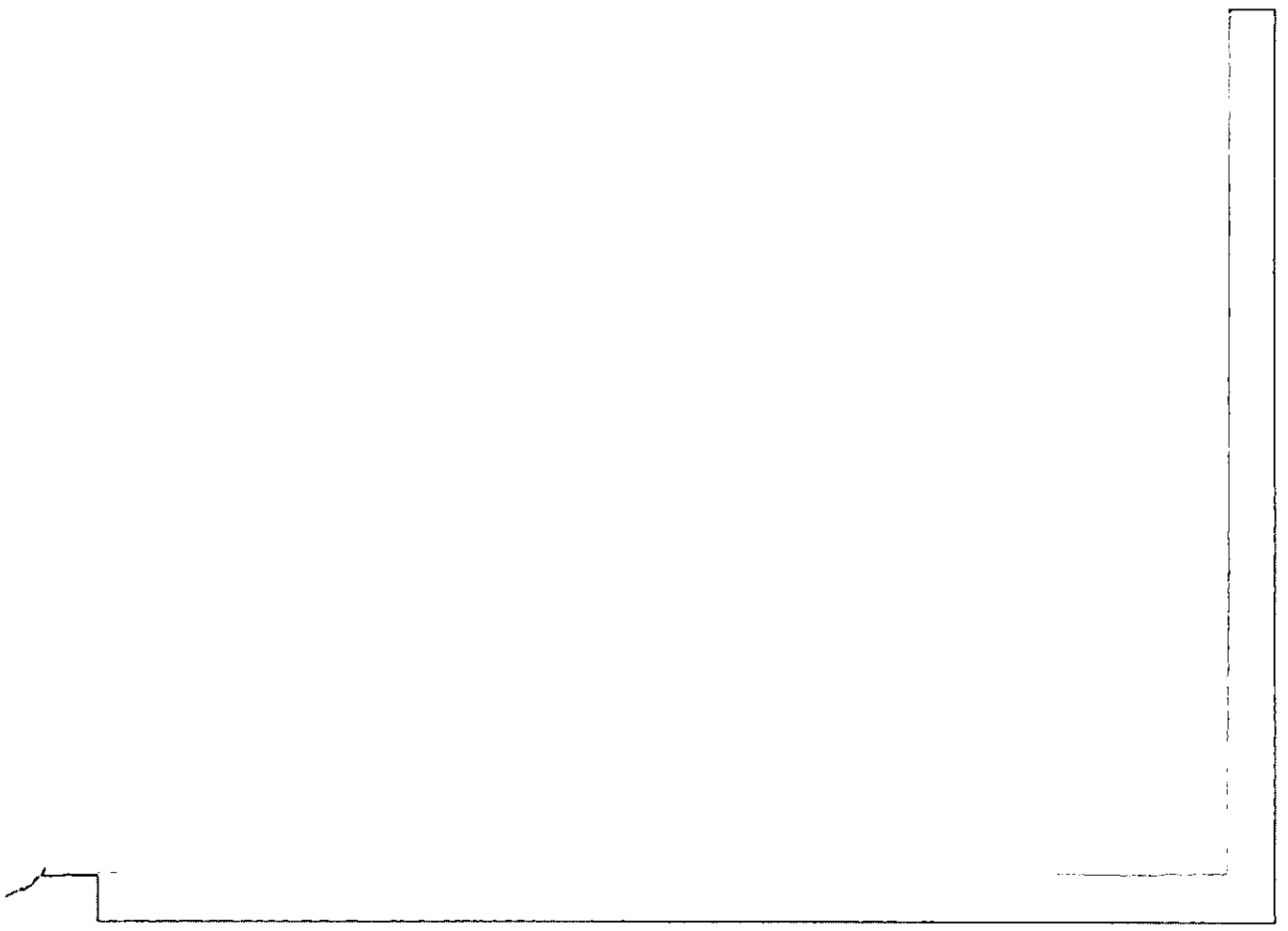
SERIES BU1



BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

CI/SfB

(A5gn)

UDC

REPRINT

NO. 106 (1991)

From FireCode to ThermalDesign: KBS for the Building Industry

J G Hosking, J Hamer, W B Mugridge
& A H Dechapunya

Paper presented at the 4th NZES Conference 1990

ISSN - 0111-7459

From FireCode to ThermalDesign: KBS for the Building Industry

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

A.H. Dechapunya
Building Research Association of New Zealand
Private Bag, Porirua
New Zealand

Abstract

Members of the Department of Computer Science at the University of Auckland and the Building Research Association of New Zealand have been collaborating for several years in applying knowledge based systems techniques to the building industry. The principal focus has been the application of codes of practice. In this paper we briefly review the collaborative work, describe its current directions, and summarise the conclusions reached.

1. Introduction

Since 1985, members of the Department of Computer Science at the University of Auckland (UofA) and the Building Research Association of New Zealand (BRANZ) have collaborated in a programme of research into knowledge-based systems (KBS) for the building industry.

BRANZ is a producer, repository, and supplier of information and expertise in matters relating to the building industry. Its mission is "to promote better building through the application of acquired knowledge, technology and expertise". BRANZ thus has an interest in new ways of disseminating its information and expertise (Whitney and Dechapunya, 1985), leading to a natural interest in KBS technology. UofA personnel view the KBS area as providing research opportunites, but recognise that complex "real world" problems are useful to focus such research. Hence the collaboration is a natural one, with BRANZ providing "real world" problems and expertise, UofA providing application solutions and KBS expertise used in developing them, and both BRANZ and UofA benefiting from products of the collaboration.

The principal, but not exclusive, focus of the collaboration has been systems and tools for representing and applying codes of practice. We begin with a brief review of systems developed, follow with a discussion of our approach to code of practice interpretation, other results, and current research directions.

2. Brief Review of Applications

The *FireCode* project (Hosking et al, 1987; Mugridge et al, 1988) aimed to develop a KBS to help check building plans against requirements of Parts 2 and 6 of *DZ4226: Code of Practice for Design for Fire Safety* (Hay, 1984), a draft New Zealand Standard.

Part 6 of DZ4226 contains provisions to ensure a building contains adequate means of escape for all occupants, such as minimum numbers of exitways and minimum widths and maximum lengths of exit paths for different building layouts and usages. A major parameter used in calculating the limits is the *use class* of each area in the building. Use

classes, defined in Part 2 of DZ4226, take into account factors such as occupant density, occupant mobility ("are they wheelchair bound?"), and inflammability of stored materials.

FireCode allows a designer to check an entire building or individual components such as one storey. Information is gathered using question and answer interaction, controlled by the system. Users can request an explanation as to why the current question is being asked (the current line of reasoning) or what the question is about (information about the item being requested). In addition, users can modify previous answers permitting what-if style exploration. More complete descriptions of *FireCode*, including sample dialogues, are in Hosking et al (1987) and Mugridge et al (1988).

Three projects followed *FireCode: Seismic* (Hosking et al, 1988) and *WallBrace* (Mugridge and Hosking, 1988) were developed by UofA, while *SubFloorBrace* (Dechapunya and Whitney, 1988) was developed by BRANZ. *Seismic* helps designers to meet the seismic loading provisions of *DZ4203: 1986 General Structural Design and Design Loadings* (SANZ, 1986). *WallBrace* and *SubFloorBrace* assist designers to meet the wall and subfloor bracing provisions respectively of *NZS3604: 1984 Code of Practice for Light Timber Frame Buildings* (SANZ, 1984). Of these three, the bracing systems are the most developed and merit further description.

Light timber frame construction is the most common method of building residential dwellings in New Zealand. Being in a seismically active region, and with parts of the country having high wind exposure, adequate bracing on timber framed structures is vitally important.

Each project consisted of two parts: a development-oriented conformance checker and a research-oriented design aid. The developmental aim was to produce applications that could eventually be made available online for checking design compliance. The research component addressed the provision of advice and assistance in designing bracing to meet the standards.

Both checkers are complete to field prototype level. From user supplied plan information they calculate design loadings, based on seismic and wind forces, check that sufficient bracing has been added to both external walls and internal bracing lines (for *WallBrace*), and subfloor components (for *SubFloorBrace*), to satisfy NZS3604. Inexperienced users are assisted with bracing line layout design. *WallBrace* design extensions, completed to a research prototype level, take a building layout and allocate bracing automatically to each external wall or bracing line to satisfy the code provisions.

Each of the systems described above has been implemented in, and motivated the design of, Class Language, a tool supporting construction of code-of-practice based applications. In the next section we discuss the design principles used in development of the language.

3. Representation of Codes of Practice

A typical code of practice consists of collections of provisions, usually grouped in structures such as clauses, sections, parts, etc., together with ancillary information such as tables, term definitions, and commentary. An example provision, from Hay (1984), is shown in Fig. 1.

> 6.3.2.3. Bedspaces: The **occupant load** of any portion of a **storey** intended as a sleeping area shall be the number of bedspaces approved.

Figure 1: A Provision from Hay (1984).

A major result of our work is a model for representing and applying codes of practice, with Class Language as its practical realisation. In developing this model we had two apparently conflicting aims. Firstly, we wanted to use obviously appropriate AI/KBS representational and reasoning mechanisms. For example, a rule-based representation of the provision in Fig. 1 is natural. Our second aim was to use techniques and tools that result in well *engineered* systems. The systems are large, subject to maintenance and large scale reorganisation (with code of practice revisions), and used in situations (such as the design of a load bearing wall) where correctness is critical. Therefore, we did not want to abandon sound software construction principles when adopting AI approaches. In the following discussion we describe features of the representation, illustrating by example how these two apparently disparate aims are reconciled. In the Appendix is an example Class Language program and a description of its execution.

• *Rules for provisions.*

A rule based representation was adopted for provisions. This seems a natural way of representing their conditional nature. Fig. 2 illustrates how the provision of Fig. 1 could be represented.

```
sleeping_area: boolean.
occupant_load: integer.
number_of_bedspaces: integer.

if sleeping_area then
  occupant_load := number_of_bedspaces.
```

Figure 2: Representing Clause 6.3.2.3 in a simple rule format

A functional, as opposed to multi-directional constraint, interpretation is placed on rules. Constraint propagation was considered too computationally expensive, and in addition a functional view of provisions is supported in the literature (Fenves et al, 1987). Rules are "executed" in a backward-chaining manner as values for their "result" data items are needed.

Some thought was put in to transferring the rule representation to a more conventional framework. For example, as shown in Fig. 2, each data item is defined with an associated type, constraining our rule representation to a strongly typed framework. Uncertainty schemes were rejected as being too ad-hoc, and have not been missed. The use of rules as the basis of automatic explanation generation was adopted, as having potential benefit, in line with conventional wisdom in the KBS community at the time (Hayes-Roth et al, 1983).

• *Procedures for sequence and output*

While rules are well suited to capturing provision semantics, some aspects of code of practice interpretation are inherently sequential. The code clause of Fig. 3 provides an example.

```
6.3.1.1 General method: Except where reduced for fixed seating,
the widths of paths of travel in each storey shall be determined
by
      (a) calculating the occupant load of each storey as
          specified in 6.3.2;
then  (b) determining the minimum widths as specified in 6.3.3;
then  (c) applying the minimum widths to determine the total
          capacity of exit-ways serving that storey as specified
          in 6.5.1;
and   (d) adding the minimum widths of any exitways combined
          under the conditions specified in 6.5.5;
but   (e) observing the special requirements specified in 6.3.3;
          6.4.1; 6.4.7; 6.5.1 to 6.5.4.
```

Figure. 3: Clause 6.3.1.1. from Hay (1984).

We explicitly represent sequence using a procedural representation. This is useful not only for examples like the one above, but also for displaying output in a sensible order. The procedural representation is intentionally limited though, having no assignment statement. All value computation is done via rules; reference to data items within procedures cause rules for that data item to be executed.

• *"When" procedures for opportunistic control*

There is some need for opportunistic intervention in the procedural/backward-chaining model described above. For example, detecting input constraint violations such as an area with two exits being described as a dead end in *FireCode*, requires opportunistic intervention as the constraint violation occurs at some distance from the data input. "When" procedures, a form of daemon, provide the necessary mechanism. These consist of a conditional expression and a procedure, which executes when the conditional expression becomes true.

• *Truth maintenance for consistency*

To make code of practice systems useful exploratory usage must be permitted, so that variations on designs can be considered and errors corrected as they are found. At the same time, it is vital to maintain consistency between input data and results. To support these desirable *engineering* features, AI/KBS ideas of logical dependency and truth maintenance were adopted. Thus at one level, Class Language is a single assignment language, where there can be only one value generated for any data item, ensuring protection from inconsistency via re-assignment. At an underlying level, however, there is a *change* facility, permitting users to modify input data. A dependency graph is constructed during execution and is used to flush values dependent on the modified data and reinstate new values (where necessary).

• *Classes for aggregation and generalisation*

Aggregation and generalisation abstractions have importance in building code compliance. The data used, building plan information, is inherently aggregational: buildings are composed of components and attributes such as storeys, roofs, subfloors, heights, widths, and locations. Each component can in turn have subcomponents and attributes. Categorisation is at the heart of most codes of practice: DZ4226, for example, is based around a hierarchy of categorisations by usage of areas in a building. Category hierarchies naturally map to generalisation relationships.

Several representations are available to support aggregation and generalisation relationships such as (from AI) semantic networks and frame based systems. Our choice, however, was a strongly typed object-oriented framework, with classes representing abstract aggregates and inheritance supporting generalisation relationships. Following conventional object-oriented wisdom (Meyer, 1988), feature declarations, rules and procedures are included within classes, providing both a context for rule application, and modularisation of encoded provisions This is in contrast to, for example, the separation of rules from frames in KEE (Intellicorp, 1984). Information hiding was also adopted, providing further encapsulation and better management of interaction between system components, in contrast to the open architectures of most AI representation systems. The Appendix example includes four class definitions.

This encapsulation approach has paid dividends in the development of our systems. For example, although *WallBrace* was developed as a succession of prototypes, the modular nature of the implementation meant that much code was able to be reused between prototypes. Major reorganisations could be effected by changing relatively small numbers of classes (Mugridge and Hosking, 1988).

•• *Classification for provision selection*

A major task in using codes of practice is the identification of relevant provisions. As discussed in the previous section, categories are an important ingredient as they statically partition the provisions. Just as important, however, is the dynamic process of *classification*, or selection of an appropriate category, as this provides a way of pruning irrelevant provisions and focussing attention on relevant parts of the code.

Classification moves from the general to the specific, in contrast to the more usual generalisation mechanisms, such as inheritance, which move from the specific to the general. The "establish and refine" mechanism of Bylander and Mittal (1986) and Chandrasekaren (1986), where objects can be classified to an immediate sub-category, provides one approach to supporting classification. Another is the relation classification mechanism of KL-ONE (Brachman and Schmolze, 1985).

However, none of these approaches appealed. They seemed on the one hand ad-hoc, and on the other, not flexible enough - in particular, being incapable of supporting the multiple independent classifiers essential for codes of practice (Fenves et al, 1987). Our approach was to use Smith and Smith (1977) ideas of static generalisation support in relational structures as the basis of a dynamic mechanism. Included within classes are classification features, the allowable values of which are names of other classes. When a classification feature of an object is given a value, the class membership of that object is modified to include the class named by the value (and, by inheritance, its superclasses). Classification is *lazy*, *incremental* and *multiple*. *Lazy* because classification properties are evaluated only when a value is needed for an object's feature and classification can bring into scope a rule or procedure body for that feature. *Incremental* because classifying once can bring into scope additional classification properties which can cause further classification. *Multiple* because a class can have more than one classification property, each of which can cause an independent classification. Further details are provided in the Appendix and in Hamer (1990).

The overall effect of the classification mechanism is that objects are considerably more mutable than in conventional typed object oriented languages, but in a controlled way. A sound theoretical basis for classification has been developed as part of our work (Hamer, 1990).

•• *Seamless integration for elegance*

While the previous discussion has presented each feature of the representation independently, an important characteristic of Class Language is that each feature

integrates with the others in a seamless fashion. Thus programmers are encouraged to use all of the representational features, without having to provide ad-hoc interfaces between them.

Fig. 4 below shows the interaction of the control mechanisms, illustrating the tight integration. Control passes between each mechanism without explicit programmer interaction. Execution of a Class Language program commences with execution of a procedure. Whenever this refers to an expression, the expression/rule evaluation mechanism determines the expression value, making use of rules. Procedures may call other procedures. Classification is invoked whenever an expression is to be evaluated or a procedure called, and classification can make available additional rules or procedure bodies. Classification in turn uses the rule/expression evaluator to find the value of classification features. When procedures monitor expression evaluation and opportunistically execute if their condition part becomes true. Execution of their procedure is via the procedure execution mechanism.

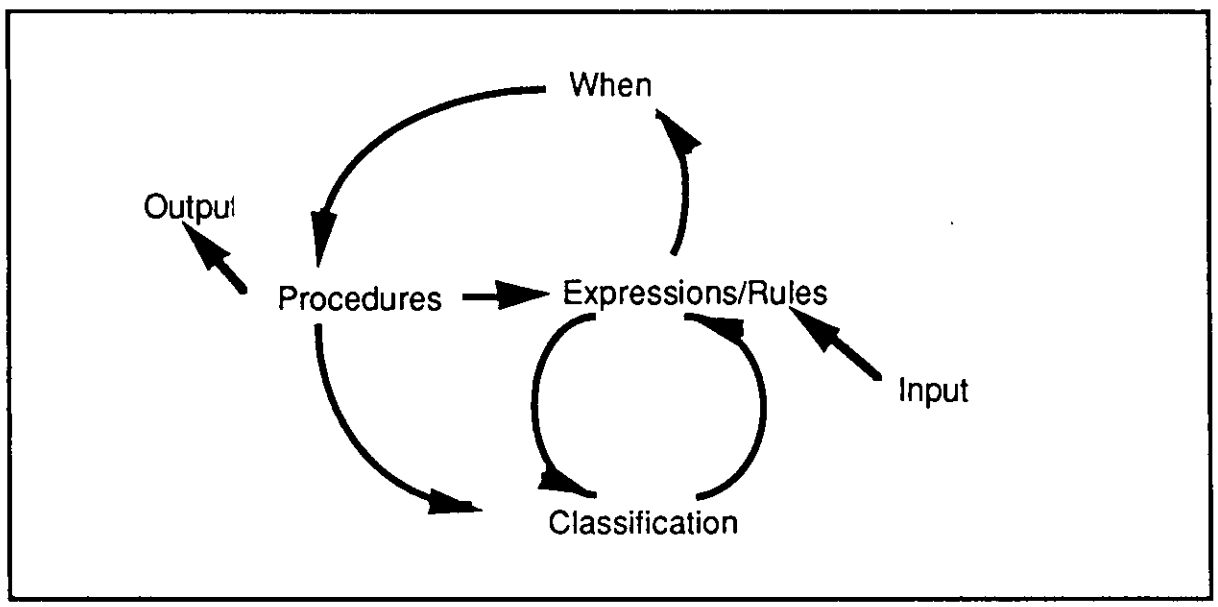


Figure 4. Interaction of control mechanisms

Each mechanism contributes a small but significant kernel to the representation, but each relies on the others for their implementation. The resulting tightness is something we have deliberately striven for. Taking this approach has meant that we have been cautious about adopting new features into the language, and hence has made development appear slow at times, but has, we feel, resulted in an elegant and well engineered tool.

• *Summary*

Apart from the classification mechanism, there is nothing particularly novel about the individual elements of our representation. What is novel, however, is the way in which we have integrated the elements together, in particular the adaptation of the AI elements into the typed object oriented framework.

As a tool for codes of practice, the representation has proven itself in the applications we have developed. However, the language is not limited to this type of application and has been successfully used in the development of materials selection systems, a building moisture problem diagnosis system, and a number of adminstrative systems.

An interesting use of the representational model is as a tool in the development of codes of practice themselves. Here, two approaches suggest themselves:

• Concurrent development of a code interpreter

Each system we have developed has uncovered ambiguities, inconsistencies, and other errors in the codes of practice. Other workers in this area have reported the same effect

(e.g. Sharpe, et al, 1986). Concurrently developing an executable version of the code provides a way of checking that the code can be interpreted as intended, and the simple act of formalisation will pick up a number of errors. A byproduct of this is a package to assist users in adopting the new code of practice, however such an exercise is valuable whether or not the computer system is developed into a releasable product or not.

- Structuring a new code in an object-oriented fashion

Current codes tend to be structured according to the basic functions which are being checked for compliance. An alternative approach is to structure codes in an object-oriented fashion, with major sections based on the (for building codes) building components. Blackmore (1989) advocates this as a possible approach to reorganisation of the Australian Uniform Building Code. Provisions are directly associated with components they refer to, making for more appropriate indexing. The approach also complements work in building product models (see 4.2 below).

4. Applying the model

The above model has focussed on code of practice representation and application. However, there is more to a useful system than the internal representational model. In this section we take a more holistic view of our work, focussing on some of its deficiencies, and decribe some of our approaches to rectifying those deficiencies.

4.1 Application User Level

A major aim of our collaboration is to develop usable systems readily accepted by the end user community. In this regard, our success is qualified. All of our systems have (for historical reasons) been developed for an interactive, line-at-a-time text-only environment. Within the constraints of this environment, we feel that our systems perform well, giving required advice in a reasonably helpful manner. The change mechanism has proved particularly useful, allowing a plan to be experimented with quite easily.

However, text-only interaction provides a severe limitation on user acceptance. Entry of plan information, for example, is fraught with difficulty. It is difficult, if not impossible, to simply, yet unambiguously, ask the user textually for geometric plan information. At the least, the resulting dialogues are quite long and tedious. Building plans are inherently visual, and hence a graphical approach to plan entry is preferred.

Similarly, the line-at-a-time, system always in control approach, is at odds with current thinking in user interface design, where direct manipulation interfaces are the norm, and more user control is provided, with the user viewing the program more as a set of services than a single monolithic package.

The tediousness of data entry highlights a problem of our current systems. Much plan data entered for one application is common to other applications. Increased productivity and greater incentive to make use of multiple applications would result if that data could be entered once and shared between applications.

Automatic explanation facilities, based on rule trace, have not proved particularly succesful, even when augmented with canned explanation text associated with each rule. Explanations often seem to be out of context and do not give a good indication of the strategy behind the questions.

To act as a focus for correcting some of these deficiencies, and to further our experience the potential of KBS technology for building design-based problems as part of an integrated information resource for the building industry, we are developing *ThermalDesign*, a system to assist in the provision of adequate thermal insulation for residential dwellings. The knowledge base of this system is based on the draft revision of the thermal insulation standard (SANZ, 1987), a BRANZ paper C1 (BRANZ, 1978), a

recent revision of the ALF manual (Basset et al, 1990), and BRANZ expertise in the subject. Visual interaction, particularly for entry of plan information, is an essential feature of the system.

4.2 Application Development Level

• *Direct Manipulation Interface*

While our representational model is succesful, the user level problems of the previous section highlight a major deficiency: the integration of Class Language with other tools is not well developed. An interface to the underlying implementation language (currently Prolog) exists, but is awkward to use, and by no means as seamless as other aspects of the language. Hence programmers are actively discouraged from using other tools in conjunction with Class Language.

To support direct manipulation interfaces, an interface from Class Language to X-Windows is being developed (Mugridge, 1989, 1990). This includes a form description and management system for easy construction and manipulation of simple forms containing buttons, radio boxes, text entry fields, etc. An interesting issue is the distribution of responsibility between Class Language and the form management system, and the provision of a seamless mechanism for interaction. On the Class Language side, external classes are defined, specifying the names and types of data items and procedures involved in the external interface, and their roles (e.g. input, prompt, output). The layout of the form is defined separately as part of the form management facility. Fig. 5 provides an example of a form and Fig. 6 the corresponding external class definition.

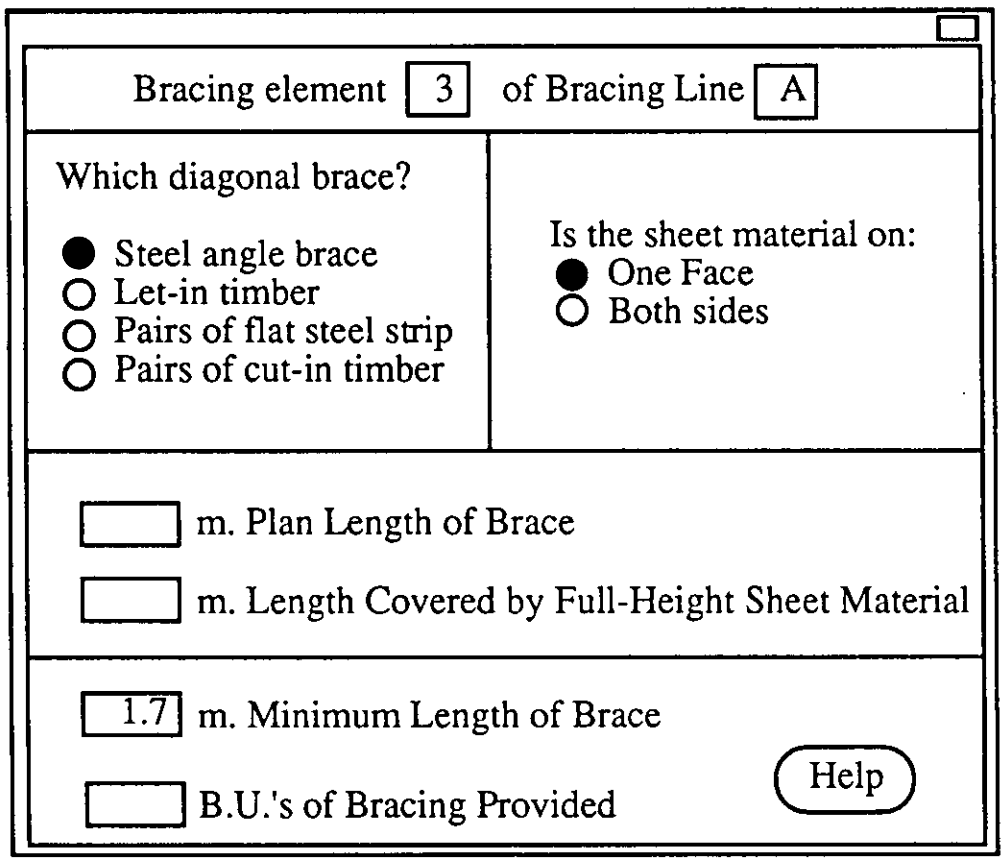


Figure 5. Example form, for wall bracing element data

```
external class DiagonalBracesInterface.
  determinant braceNo : integer.
  determinant braceLine : text.
  output minimumLength : metres.
  output bracingProvided : bu.
  output lengthError : text.
  procedure help.
  input diagonalBrace: [steelAngle, letinTimber,
                        pairsFlatSteel, pairsCutinTimber].
  input sheetMaterial : [oneFace, bothFaces].
  input planLength : metres.
  input sheetLength : metres.
  external procedure show.
  external procedure hide.
end DiagonalBracesInterface.

class Bracing.
  generalisation DiagonalBracesInterface.
  parameter minimumLength, braceNo, braceLine.
  bracingProvided
     := planLength * sheetLength * buPerMetre.
  buPerMetre : integer
     := bu(diagonalBrace, sheetMaterial).
  lengthError
     := 'The brace must be at least the minimum length.'
                  if diagonal^planLength < minLength
     | ''.
  procedure help.
     ...
     end help.
  ...
end Bracing.
```

Figure 6 External class for form of Fig. 5 together with use of that class

By adopting a form based approach with the user free to enter information in any order within the form, and to request additional services via buttons (such as the help button in Fig 5), the user is provided with much more control over program execution. Hypercard-like (Apple, 1988) form navigation facilities provide simple browsing through entered data, forming, we feel, a better basis for explanation facilities than rule tracing. Editing of previously entered data invokes the (somewhat expanded) change facility to provide input-output consistency.

• *Plan Entry and Building Product Model*

The problems of entering and reusing plan information have prompted development of a specialised package supporting graphical plan entry and display, a mockup of which is shown in Fig. 7.Though initially developed for *ThermalDesign*, it is intended to be more general purpose, allowing (with extension) entry and editing of plan data for a variety of applications, such as the bracing packages. In the latter role, there is a strong need for concurrent development and implementation of a general model for building plan information, a so called *building product model*. Such models are an active research topic (Amor, et al, 1989; Bjork, 1989).

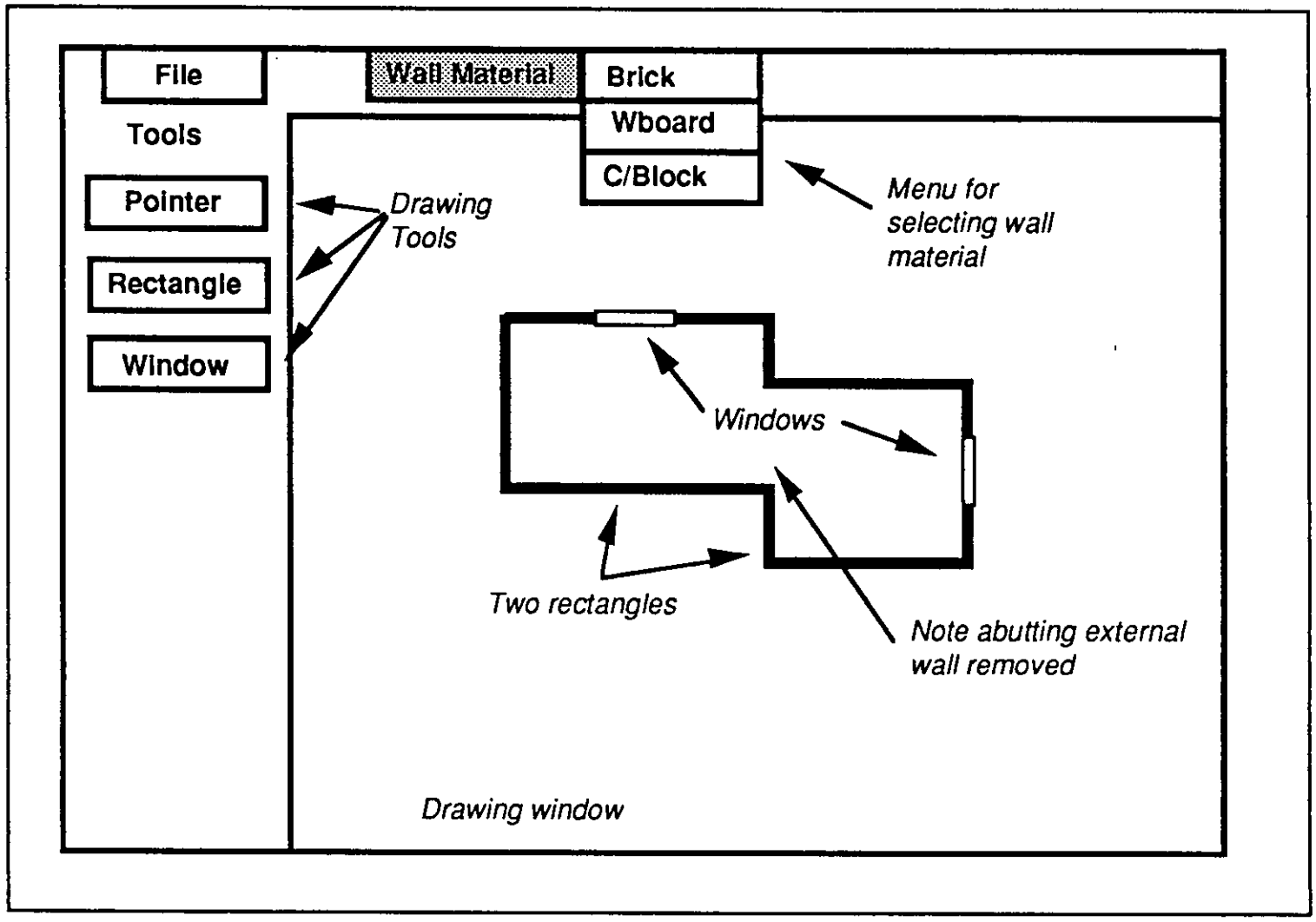


Figure 7. Mockup of prototype plan entry package

• *Database Interface*

In addition to better user interfaces, interfaces to other packages are needed, in particular to database systems for storage and retrieval of persistent data. For example, the Class Language based *Adhesive* and *Sealant* systems (Dechapunya and Whitney, 1988) developed by BRANZ provide advice on compound names of suitable adhesives and sealants. To make these systems more useful, an interface to a database of product information providing brand names, pricing information, availability, etc is needed. Similarly, persistent storage for the common building product model described in the previous section is required.

Accordingly, a database interface for Class Language is being developed. The work of Smith and Smith (1977), which acted as motivation for *classification*, is proving useful in providing the transition from an object-oriented world back to a relational database world.

• *Visual programming environment*

The Class Language programming environment is somewhat lacking. While some purpose built facilities have been provided, such as a browser and debugger, there is much room for improvement.

A visual programming environment is being developed for Class Language, allowing programmers to graphically represent the generalisation and aggregation relationships (in particular) of a Class Language program, and have them automatically converted to and from the equivalent textual form. Fig. 8 shows a screen dump from a prototype version of the environment. As a later extension, a graphical form editor is planned to complement the form management facilities described earlier .

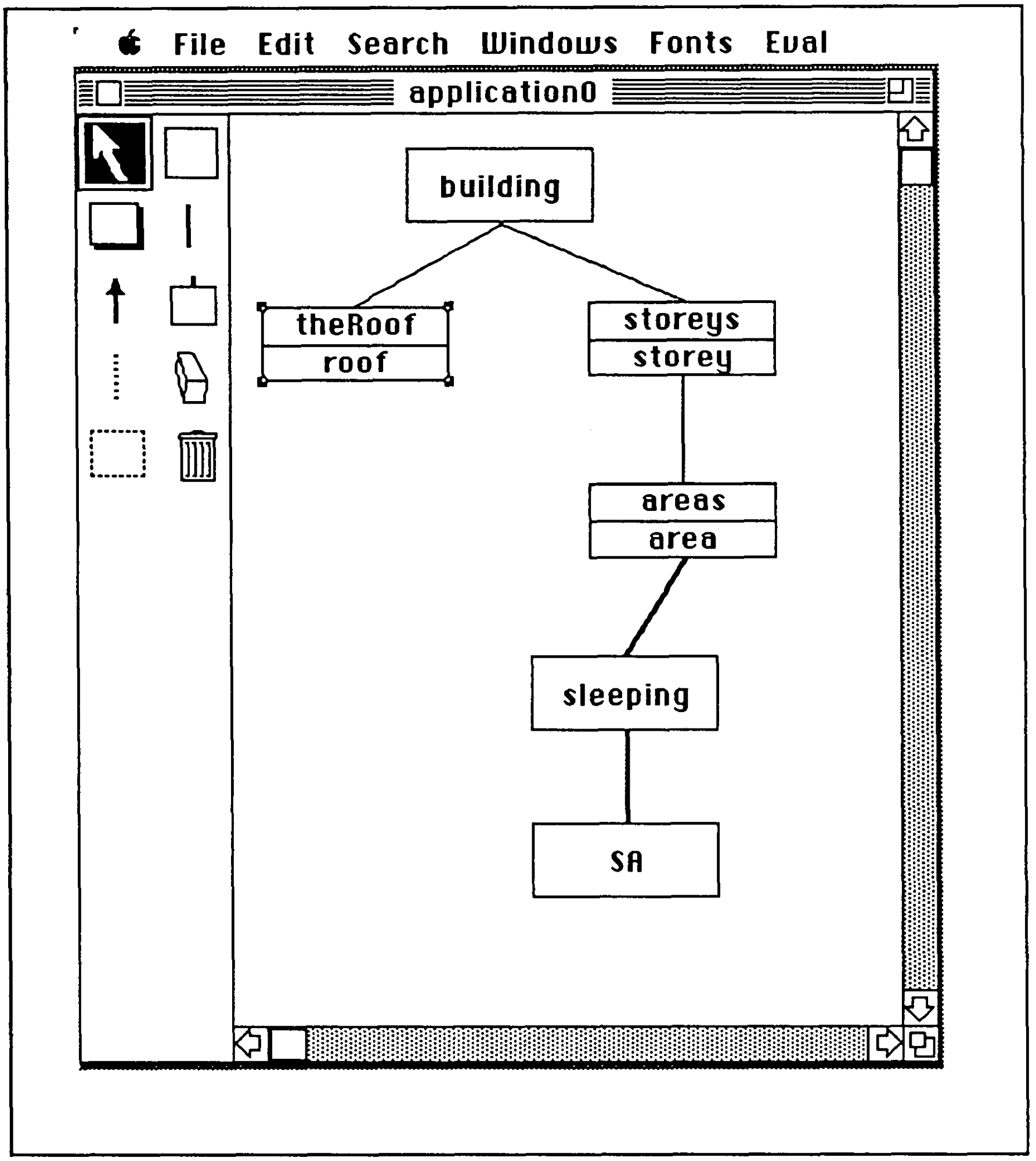


Figure 8. Screen dump from prototype visual programming environment

• *Representational extensions*

Some representational extensions are also planned. The rule mechanism of Class Language can alternatively be viewed as a weak form of functional programming (Bird and Wadler, 1988). To improve the power of the functional component, rules with parameters (ie functions), and higher order function support are being added.

A further extension is needed for design-oriented systems. Here we have an interest in blurring the distinction between class and instance in a controlled manner, in line with work in prototype languages (see e.g. Ungar and Smith, 1987). The exact mechanism is still under investigation.

4.3 Tool Implementation Level

The existing implementation of Class Language consists of a compiler, written in Prolog, which compiles Class Language source to Prolog. In developing the implementation our

major aim was to prototype the language in as quick and flexible a way as possible in order to prove the language principles. Execution speed and portability were therefore not an issue. Prolog proved a convenient and productive environment for this purpose.

Having proven the language principles and stabilised its design, we now feel it appropriate to improve portability and execution speed. Hence, a virtual machine for Class Language has been designed, and a C-based runtime system is under implementation.

5. Conclusions

The aim of our collaborative work has been to investigate the usefulness of KBS techniques to the building industry, and in particular to the task of code of practice interpretation and application. The results of our work support the use of KBS techniques, such as rules, daemons, and truth maintenance, as components of a code of practice representation. More importantly, perhaps, is that we have been able to adopt these techniques within an environment encouraging sound software development.

Deficiencies of our work are primarily in interfacing our representational model with other tools, and in particular, user interface constructors. Work is in progress to correct these deficiencies.

The collaboration itself has been very succesful, making good use of limited resources. The "real world" problems and the availability of expertise in solving them has provided direction to our work, without significantly compromising the research element, as opposed to development. While some problems have been encountered through geographic separation of the two organisations, these have not proved significant, being readily addressed by improvements in communications technology.

Acknowledgments

The authors would like to acknowledge the contribution made by colleagues at BRANZ, UofA, and elsewhere, in particular: Rob Whitney; Grant Coupland, Dennis Bastings, Martin Buis (FireCode); John Cranston, Andrew King, Steven Lomas (Seismic); Joe Ten Broeke (WallBrace, SubFloorBrace); Christopher Lee (SubFloorBrace); Roger Fowkes, Harry Trethowen (DAMP); Wayne Sharman (SEALANT); Philip Watkinson, Chris Fromont (ADHESIVE); Bill Irvine (ThermalDesign); and John Grundy (Visual Programming). The UofA personnel also gratefully acknowledge the financial support for the collaboration provided by BRANZ, the University of Auckland Research Committee, and the UGC.

References

Amor, R., Groves, L., and Donn, M. (1989). Knowledge representation for integrating design tools, *Proc. Australasian Conf. on Expert Systems in Engineering, Architecture, amd Construction*. University of Sydney.

Apple (1988) *Hypercard Users Guide*, Apple Computer Inc, Cupertino, Ca.

Basset, M.R., Bishop, R.C., van der Werff, I.S. (1990). *ALF MANUAL: Annual Loss Factor: Design Manual - An Aid to Thermal Design of Buildings,* BRANZ, Judgeford.

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*, Prentice-Hall, UK.

Bjork, B. (1989). Basic structure of a proposed building product model, *Computer-Aided Design*, **21**(2), 71-78.

BRANZ, (1978). *C1: A Construction Guide to Home Insulation,* BRANZ, Judgeford.

Blackmore, J. (1989). *Priv. Comm.*

Brachman, R.J., and Schmolze, J.G. (1985) An overview of the KL-ONE knowledge representation system, *Cognitive Science*, **9**, 11.

Bylander, T. and Mittal, S. (1986). CRSL: A language for classificatory problem solving and uncertainty handling, *AI Magazine*, August 1986.

Chandrasekaren, B. (1986). Knowledge based reasoning - high level building blocks for expert system design, *IEEE Expert* Fall.

Dechapunya, A. H. and Whitney, R.S. (1988) Knowledge-based systems for building technology in New Zealand, *Proc Symposium on knowledge-based systems in civil engineering*, Monash, August.

Fenves, S.J., Wright, R.N., Stahl, F.I., Reed, K.A. (1987). *Introduction to SASE:Standards Analysis, Synthesis, and Expression*, U.S. Department of Commerce, May 1987.

Hamer, J. (1990). *Expert Systems for Codes of Practice*, PhD Thesis, University of Auckland Department of Computer Science.

Hay, R.D. (Ed), (1984). *DZ4226: Code of Practice for Design for Fire Safety*, Standards Association of New Zealand, Wellington, New Zealand.

Hayes-Roth, F., Waterman, D.A., and Lenat, D.B. (1983). *Building Expert Systems*, Addison-Wesley, Massachusetts.

Hosking J.G., Lomas S., Mugridge W. B. (1988). The development of an expert system for seismic loading, *Proc Knowledge-Based systems in Civil Engineering Symposium*, Monash University, Melbourne, Australia.

Hosking, J.G., Mugridge, W.B. and Buis, M., (1987). FireCode: a case study in the application of expert system techniques to a design code, *Environment Planning and Design B* **14**, 267-280.

Intellicorp (1984). *The Knowledge Engineering Environment*, Technical Article, Intellicorp.

Meyer, B. (1988). *Object-Oriented Software Construction*, Prentice Hall.

Mugridge, W.B. (1989). *An External Interface for Class Language*, BRANZ Research Contract 85-024 Technical Report No. 8, 30 pp.

Mugridge, W.B. (1990). *Enhancements to an object-oriented language*, PhD Thesis, University of Auckland Department of Computer Science.

Mugridge, W.B. and Hosking, J.G., (1988). The development of an expert system for wall bracing design, *Proc The 3rd New Zealand Expert Systems Conference*, Wellington, New Zealand, 10-27.

Mugridge, W.B., Hosking, J.G., and Buis M. (1988). FIRECODE: An expert system to aid building design, in Newton, P.W., Taylor, M.A.P., and Sharpe, R. (Eds) *Desktop Planning* Hargreen, Melbourne.

SANZ, (1984). NZS3604:1984 (1984). *Code of Practice for Light Timber Frame Buildings*, Standards Association of New Zealand, Wellington, New Zealand.

SANZ, (1986). DZ4203:1986 (1986). *General Structural Design and Design Loadings for Buildings*, Standards Association of New Zealand, Wellington, New Zealand.

SANZ (1987) DZ4218: 1987 *Minimal Requirments for the Thermal Insulation of Residential Buildings*, Standards Association of New Zealand, Wellington, New Zealand.

Sharpe, R., Marksjö, B.S., Thomson, J.V., Wilson, D. (1986). Expert systems applications in building and construction, *Proc. 1st Aust Art Intelligence Congress* (Melbourne).

Smith, J.M and Smith, D.C.P. (1977): Database abstractions: aggregation and generalisation, *ACM Trans. on Database Systems* **2** (2).

Ungar D., and Smith, R.B. (1987). Self: the power of simplicity, *Proc OOPSLA'87.*

Whitney, R.S. and Dechapunya, A.H. (1985). The building of a system, *New Zealand Interface*, September, 77-82.

Appendix - Class Language

Class Language blends functional and object-oriented programming concepts together in one language. The language is strongly typed. The components of a class include argument-less functions and procedures. Multiple inheritance of classes is supported. The key differences from other OOLs are: the support for *dynamic classification*; and lazy expression evaluation.

The example in Fig. 9, and execution dialogue in Fig. 10 illustrate some of these features. The purpose of the program is to compute the occupant load of a storey. The occupant load of a storey is the sum of the occupant load of each area within the storey. For sleeping areas, the occupant load is the number of bedspaces; for assembly areas it is the number of occupants, but not less than 5. The program uses four classes: one for the storey; one

for the general class of areas; and one each for the sleeping and assembly areas. The `Storey` class includes a collection of component `Areas`. The `Area` class includes a classification feature which selects the appropriate area type. Exported features are identified by the keyword **public**.

The program starts execution with the procedure 'main'. This procedure contains a single output statement, which refers to the object `'the_storey'` and a feature of that object, `'occupant_load'`. As all object references are function invocations, code for evaluating `'the_storey'` and then `'the_storey^occupant_load'` is executed. The former creates a new `Storey` object. Evaluating `'occupant_load'` in the context of this new object causes evaluation of the associated expression, resulting in a prompt for the number of areas in the storey, and then creation of that number of `Area` objects. The `occupant_load` of each of these areas is required to complete the summation. The presence of a classification property in the `Area` class that can classify to subclasses containing expressions for `occupant_load` causes evaluation of the classification property for each area. Having performed this classification, the appropriate expression in the subclass is evaluated, resulting in further information being sought from the user. Once this evaluation, including the summation, is completed, the `occupant_load` is displayed and the program terminates.

```
instance the_storey : Storey := new.
procedure main.
    display('Occupant load = ', the_storey^occupant_load)
end main.

class Storey.
     public occupant_load.
     areas: list Area
           := collect(i in 1..area_count, new Area(seq_no :=i)).
     occupant_load: integer
           := sum(collect(a in areas, a^occupant_load)).
     area_count: integer
           := ask('How many areas are there:? ').
end Storey.

class Area.
     public occupant_load.
     parameter seq_no: integer.
     classification
           areaType: [Assembly, Sleeping]
                 := ask('Is area #', seq_no ' a'&
                 '[ 1 ] Assembly area'&
                 '[ 2 ] Sleeping area'&
                 'Enter 1 or 2 ?: ').
     occupant_load: integer.
end Area.

class Assembly.
     generalisation Area.
     occupant_count :integer
         := ask('How many occupants are in area #',seq_no,':?').
     occupant_load
         := occupant_count if occupant_count > 5
         | 5.
end Assembly.

class Sleeping.
     generalisation Area.
     bedspace_count: integer
           := ask('How many beds are in area #',seq_no,':? ').
     occupant_load := bedspace_count.
end Sleeping.
```

Figure 9: An Example Class Language Program

```
How many areas are there:? 2
Is area #1 a
[ 1 ] Assembly area
[ 2 ] Sleeping area
Enter 1 or 2 ?: 1
How many occupants are in area #1:? 20
Is area #2 a
[ 1 ] Assembly area
[ 2 ] Sleeping area
Enter 1 or 2 ?: 2
How many beds are in area #2:? 10
Occupant load = 30
```

Figure 10. Dialogue resulting from executing Fig. 9

From FireCode to ThermalD
esign : KBS for the build

BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

BRANZ MISSION

To promote better building through
the application of acquired knowledge,
technology and expertise.

HEAD OFFICE AND RESEARCH CENTRE

Moonshine Road, Judgeford
Postal Address - Private Bag, Porirua
Telephone - (04) 357-600, FAX - (04) 356-070

REGIONAL ADVISORY OFFICES

AUCKLAND
Telephone - (09) 5247-018
FAX - (09) 5247-069
290 Great South Road
PO Box 17-214
Greenlane

WELLINGTON
Telephone - (04) 357-600
FAX - (04) 356-070
Moonshine Road, Judgeford

CHRISTCHURCH
Telephone - (03) 663-435
FAX - (03) 668-552
GRE Building
79-83 Hereford Street
PO Box 496