EXPERT SYSTEM DEVELOPMENT WITH KAPPA-PC

Maurice Danaher

School of Computer and Information Science, Edith Cowan University, Perth, WA 6050, AUSTRALIA, m.danaher@cowan.edu.au

ABSTRACT

This paper describes KappaPC – an expert system development environment. KappaPC is being used in the development of a knowledge-based system for preliminary structural design. As very little information has been published on KappaPC a discussion of this environment should prove very helpful to potential developers. An overview of the system is presented and its capabilities and features are described. The advantages and benefits of the system are discussed as well as weaknesses and limitations.

1. INTRODUCTION

KappaPC (Intellicorp 1996) is an application development environment for PCs. It is being used to develop a prototype knowledge-based system to assist with preliminary structural design. Preliminary structural design is the first stage in the design of buildings when overall concepts are considered. These include shape, form, layout, materials, cost and others. During this early design stage many decisions are based on experience, rules of thumb and judgement.

Little information has been published on KappaPC, apart from a handful of very brief papers. These reports primarily discuss the application for which it was used and give little details about its features, capabilities, usefulness or ease of use. Hasan et al. (1994), Kiernan et al. (1996) and Tsang and Bloor (1994) have reported on knowledge-based systems that they built or were building using KappaPC. For this study the system has been examined in detail.

KappaPC is designed to provide the following:

- Expert system tools, including an inference system.
- Graphical object-oriented application development in a standard C implementation;
- Integration with existing MS-Windows applications including support for Windows Dynamic Data Exchange (DDE), and Dynamic Link Libraries (DLLs);
- Production of ANSI C program code executables, which allow for the efficient distribution of the finished programs;
- Interfaces to SQL databases, spreadsheet programs and CAD packages.

For this study version 2.4 of KappaPC was used on a Pentium 3, 550MHz computer with the Windows 98 operating system.

KappaPC may be described as a complete development environment, which provides a wide range of edit tools and debuggers for designing and running applications. In the KappaPC system, the active components of the application domain are represented by data structures called objects. These objects can be either classes or instances within classes and they may represent concrete things like building subsystems, such as the floors or the walls or components like beams and columns. The objects can also represent intangible concepts like cost or evaluation criteria. A developer can link objects together into an object hierarchy to represent the equivalent relationships among the objects in a model abstracted from a particular domain.

The object-oriented programming tools within KappaPC can be used to provide these objects with methods, which contain algorithmic code like that found in the functions in conventional programs. Once the objects and methods have been identified for a knowledge base, then the system can be developed. System development commences with the production of a specification to describe how the objects are to behave and how the system will reason about the objects. Systems built on KappaPC usually require a set of pre-written rules, where each rule specifies a set of conditions and a set of conclusions to be made if the conditions are true. The conclusions may represent logical deductions about the objects in the knowledge base and how they might change over time.

In KappaPC each rule is a relatively independent module and a reasoning system can be built gradually, rule by rule. KappaPC also allows the developer to use object-oriented programming to combine and unify many standard AI methodologies such as, frame-based representation, production rules, demons or monitors and graphics into a comprehensive hybrid system.

The package starts up with a series of windows as shown in fig. 1. These windows control the operation of the Kappa system and allow the user to bring into view a number of other windows. The Object Browser window allows the user to view graphically and edit the class structure of a program. The EditTools windows provide the facility to edit data objects, which consist of classes and instances, rules, goals and functions.

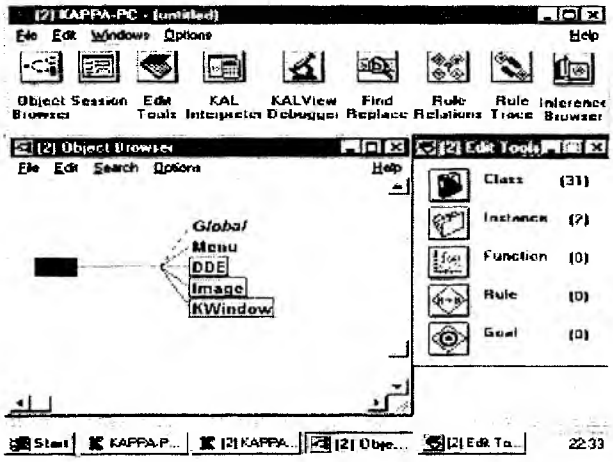

Figure 1. KappaPC application development input screens

User interaction with KappaPC proceeds graphically, the system being accessed via a mouse, or by typing into one of the five custom editors or via the Interpreter Window, which allows commands, statements and functions to be input and executed interactively. Graphical input can also be effected via the Object Browser or via one of KappaPC's Session Windows.

2. OBJECTS

Objects are represented in KappaPC as classes and instances of classes. These can be organised into hierarchies or taxonomies using subclass and instance relations. Fig. 2 reproduces the Object Browser, which displays part of the knowledge base developed during the study. These objects are all classes and the links between them are shown.

The solid lines indicate subclass links, which partition the Rib-Moulds class. These links represent is_a_subclass (is_a_member_of) relationships. KappaPC also provides for the is_a_kind_of or instance_of relationship. However, these are the only relationships provided for explicitly in KappaPC and other kinds of relationships must be implemented indirectly. For example a developer can use the slots in objects to create links to other objects in order to represent association type relationships. These is_a_member_of relationships are used throughout the design tool system created during the study to construct representations of the design alternatives at different levels in the design hierarchy.

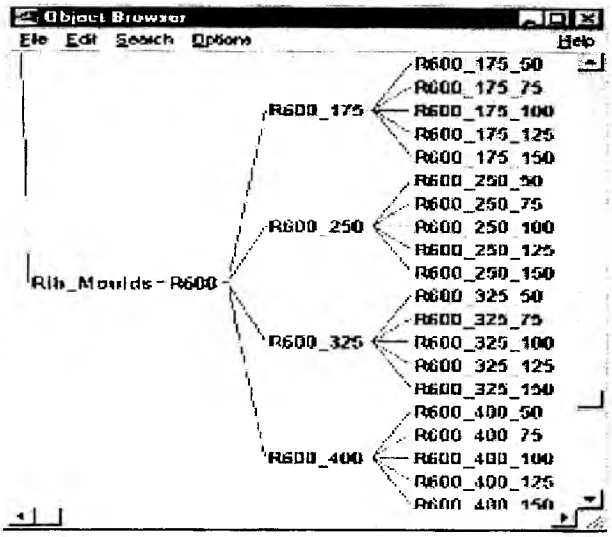

Figure 2. The Rib-Moulds Object Hierarchy

The links between objects also provide the paths via which objects inherit attributes from other objects higher in the hierarchy. Each class can have any number of slots and KappaPC provides two kinds of slots, member and own. The member slots of a class are inherited by its subclasses while the own slots are not. Furthermore, when a subclass inherits a member slot the slot also acts as a member slot for the subclass, if this subclass is a subclass of the parent. Otherwise it inherits it as an own slot and cannot pass it on to its subclass.

3. SLOTS

KappaPC provides a data type, referred to as a slot, which resides in the Kappa object, which may be either a class or instance. The user can update the slots to tailor an object so that it may represent the important properties of a real object. Each slot can be used to describe a characteristic or attribute of the object. To specify the attribute, the user assigns a value to the slot.

Slots are inherited down the object hierarchy, and as the hierarchy grows, the classes lower down gradually accumulate inherited slots. As noted above, objects can have their own slots and they can inherit slots from ancestor classes, i.e. classes above them in the class hierarchy.

When an object inherits a slot from an ancestor, the object does not have to maintain the inherited slot value; the user can make the slot local to the

subclass and then insert a different value from the one inherited by the slot. KappaPC also allows slot values to be changed programmatically. This feature is very useful for programming knowledge-based systems. Slot inheritance provides a shortcut to updating attribute values throughout the hierarchy. If a slot value is changed at a point in the hierarchy then the change will be reflected in values of the slots lower down the hierarchy, which have been inherited down through the hierarchy.

Local slots describe features that are private to the object that contains them. If the object is a class, its local slots describe that class itself (as opposed to its members). If the object is an instance, its local slots provide information about that particular instance. The user can input and change slot values using the slot editor, which is shown in fig. 3.

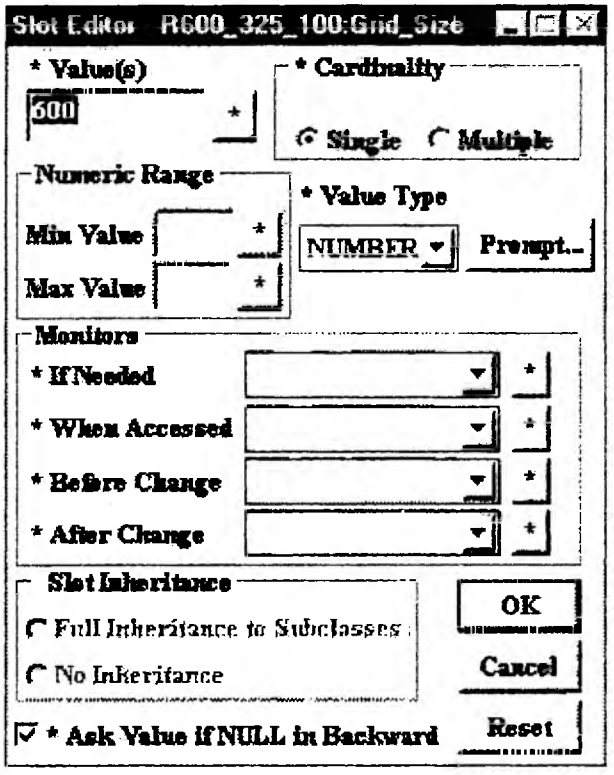

Figure 3. The Slot Editor

Once a slot is made local and the value of the slot is changed, all classes and instances that subsequently inherit the slot get the new value. This feature was used in the system designed during the study. As the system's search tree of design objects grows, new partial designs are added and at certain levels in the tree detailing calculations are done to estimate and fix the initial sizes of component parts. These calculations result in changes to various slot values in the design objects. These changes are effected programmatically using a variety of assignment functions and the new values are then reflected in the subsequent levels in the hierarchy. This shadowing effect of inheritance is a useful feature of object-oriented programming; all objects below an object with a local slot are affected by the change.

KappaPC provides a set of standard slot options to describe and manipulate object slot values. These slot options describe slots in much the same way that slots describe the attributes of objects. Furthermore, a given slot can have many different options, while at the same time having no value assigned. If a slot does not have a value, at a point in time, then KappaPC assigns it the value NULL. Also if a slot value is reset (and it did not have a value before it was assigned one), the new value of the slot will be NULL.

The types of slot options provided by KappaPC are:

- Cardinality (single or multiple), this specifies the number of slot values allowed, if multiple is chosen the slot can have multiple values, which are input in the form of a list;
- Allowable Values, this describes the set of allowable slot values, ie. a Boolean slot would have two values; TRUE and FALSE;
- Value Type, this option controls the type of the slot values, ie. text, number, Boolean or object, which can be the name of a class or an instance;
- Slot Inheritance. This option controls the inheritance behaviour of the slots; the values of which can be passed down the hierarchy or stopped at this object using the Slot Inheritance option; and
- Change monitors or demons, these options include the If Needed, WhenAccessed, Before Change and After Change monitors. These are methods that are activated when object and slot pairs are accessed. They are used extensively in systems that rely on rule-based reasoning. Monitors may be defined as private functions or functions that change the value of slots elsewhere in the object hierarchy. The If Needed option contains the name of a method in this object. The method is automatically executed when the value of the slot is requested and there is no value in the slot i.e. when a value is needed. Likewise, if the WhenAccessed option is attached to the slot, then the method is executed when the slot is accessed, even if the value of the slot is known.

4. METHODS

Apart from information that describes the object's characteristics each object also contains information that specifies its behaviour. Each action that an object can carry out is represented by a method, which is a procedure, usually written as a KAL program function. Furthermore, KappaPC facilitates the characteristic object-oriented process of method activation by programmatically sending and receiving messages. When an object receives a message that corresponds to one of its methods that method is activated and the object carries out whatever procedure is specified by the method. Kappa objects inherit methods in the same way that they inherit slots and this feature has been used during the study to organise the behaviour of the new system.

KappaPC methods provide for the object-oriented characteristic of polymorphism. Thus different Kappa objects can have their own individual methods with the same name as the methods in other objects. This then allows the different objects to respond in their own characteristic way, to the same message put out by the application. This facility is used in the prototype system to incorporate an element of polymorphism. Thus the application can issue a single instruction to commence the detailing process of all the partial design objects in the vertical subsystem. This is done when the design has proceeded down the design hierarchy to a certain level. The instruction to commence detailing is then passed round the design hierarchy at that level, using a series of messages and each object reacts according to its type. The user can create object methods via the method editor, which is shown in fig. 4. Methods can also be created programmatically using the MakeMethod function.

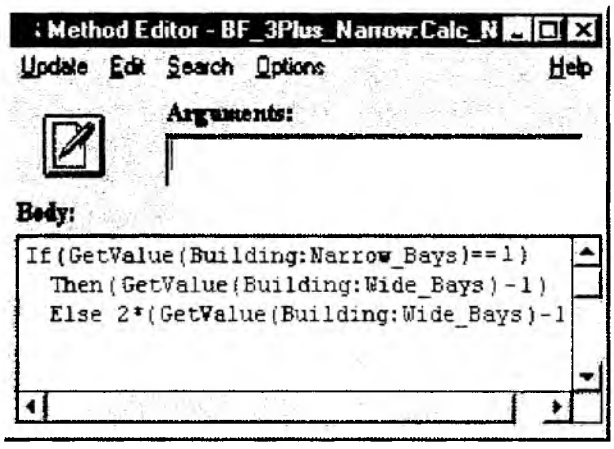

Figure 4. The KappaPC Method Editor

A method can be coded to include any KAL function or sequence of functions. Each method has three default arguments: self, theParent and theOwner. The value of the self variable is the object that receives the message and it allows methods to access the values of other slots in the same object. They can also initiate other methods in the same object by sending the message to self. Methods can perform several kinds of actions:

- Change the state of the application, generally by changing slot values in an object;
- Send messages, either to the same object or to other objects; and
- Activate other facilities of the KappaPC system, such as rule-based reasoning or data access.

If a method causes changes in an application, the changes are typically to slot values in the object that receives the message. If changes need to be made outside of the object that receives the message, then appropriate messages can be sent to the necessary objects.

Method inheritance acts in a similar way to the inheritance of slot values. It can be used efficiently to create and refine the behaviour of objects. Like slots, methods can be inherited, made local and edited at the class or instance level. If the object contains a method, any of its subclasses that do not contain a method of the same name will inherit the method unchanged. If a message is sent to an object to invoke a named method then that method will be invoked in the object, which receives the message, not the other objects in the hierarchy, which may have methods with the same name.

5. OBJECT-ORIENTED PROGRAMMING

The KappaPC objects, which have been described above, allow the user to describe real world objects and support the main characteristics of object-oriented programming, which are: inheritance, encapsulation and polymorphism.

Inheritance is used in this project to achieve conceptual clarity - similar types of objects are grouped into subclasses, which share a common parent. For example, design options, which include Rib-moulds, Waffle-moulds and Steel-decks are grouped into their own distinct class groupings. Each of these groupings has a common parent class, which has the generic attributes for the whole hierarchy.

Inheritance simplifies object creation. Thus if a new class is to be created, which is similar to an existing one, then it can be created as a subclass of the existing class. The new class automatically inherits its parent slots and the user need add only the new slots, which are required to differentiate it from its parents. This facility is used in the study system during the creation of the design objects, which make up the search tree of design alternatives. The generic class Building is placed at the root of this tree and the inheritance mechanism is used to create new subclasses at each design level.

KappaPC can support single inheritance only, so additional functional coding was performed to provide for the multiple inheritance required during the generation of the new levels in the tree.

6. KAPPA APPLICATION LANGUAGE - KAL

KAL is a high-level application development language, which allows users to program the functions required to support procedural programming. During the development of the system used in the study procedural programming was used extensively to program the design synthesis and evaluation activities.

KAL can be used to manipulate application objects, mathematical functions, strings, lists, files, control blocks, windows, popup menus, input forms, application graphics, interfaces, and system access. It also allows the user to write functions, methods and rules, create message passing schemes and activate the inference engine, to complete calls to external functions, employ graphics and animation and to facilitate data access.

KAL source code can be compiled to ANSI C. Furthermore, a suitable C compiler can further compile this C code into a dynamic link library (DLL), which runs an average three times faster than the original interpreted KAL code.

As well as object-oriented programming KAL allows the user limited access to non-object local variables, which are used with Let and loop constructs and which are settable, i.e. they can be used in assignment statements.

A debugger provides the user with a means to debug KAL source code. The user can view functions, methods and the execution stack and can set break points for functions and methods. The user can also set watches on the value of object slots or any other coding entity, by selection. The debugger has two modes; 'step-over' and 'trace-into'. In addition KappaPC provides a 'Find/Replace Utility' to allow for local and global find and replace capabilities. Fig. 5 shows a typical debugger display.

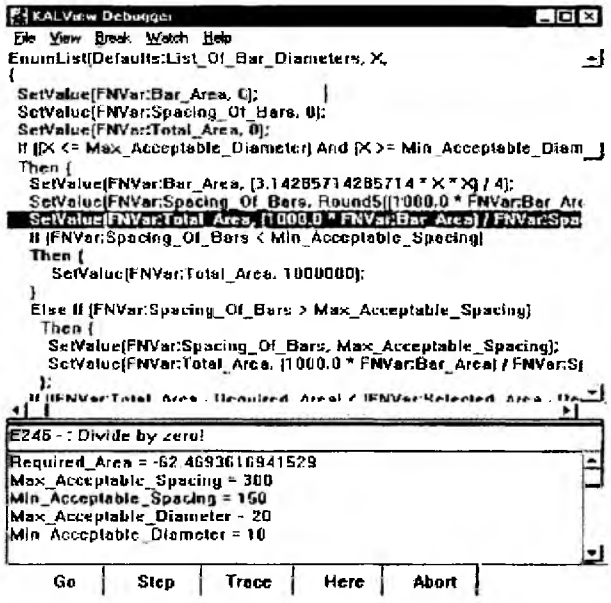

Figure 5. Debugger Display During Function Trace.

7. REASONING MECHANISM

KappaPC provides facilities for rule-based reasoning, which allows the user to develop rule-based systems. These systems represent knowledge in terms of a set of rules, which determine what the system should do or what conclusions the user should draw in different situations.

In KappaPC the rules are represented as "if" (conditions) and "then" (actions) statements, they are associated with a subset of facts, represented as a set of object and slot pairs drawn from the domain knowledge in the system. The KappaPC reasoning mechanism consists of a combination of the rules and object slots, which are organized into an inference network and a system interpreter, which controls the application of the rules.

The interpreter has two main modes of reasoning: agenda-controlled forward chaining and goal-driven backward chaining. The study system employs forward chaining through out. In forward chaining the facts in the system are held in working memory, which is continually updated as rules are invoked. The rules represent possible actions to take when predetermined events change these facts in working memory. These actions usually involve adding or deleting items from working memory.

The interpreter controls the application of the rules, given the contents of working memory, and

thus controls the actions taken by the system. The interpreter works through the rules in cyclic manner as follows:

- Check to find rules, which have the conditions satisfied;
- Select a rule, based on a predetermined strategy; and
- Perform the action in the action part of the rule, thereby modifying current working memory.

KappaPC has several features to enhance its rule-based reasoning, these include four rule-firing schemes: depth-first, breadth-first, best-first, and selective, pattern matching on objects. It also allows priorities to be set for conflict resolution, and provides a flexible explanation facility to explain the conclusions arrived at by the inference mechanism. KappaPC also provides features, which allow a developer to debug the inferencing scheme being used. These include, rule trace and break capabilities, slot trace and break capabilities and the ability to step through the inferencing process. These tools are accessed through three specialised editor windows in the development environment: the Rule Relations window, the Rule Trace window and the Inference Browser window

The Rule Relations window dynamically displays rule networks and interdependent rules. It displays "if" and "then" dependencies for related rules and allows browsing through the compiled rule network and provides interactive editing of rules and their relationships.

The Rule Trace window allows the user to specify application components to be examined during the inferencing process. It provides capabilities for active trace, where the user can step through inferencing one step at a time and can momentarily stop inferencing at pre-defined states, change parameters, and then resume the process. The rule trace window displays the active rule list, agenda contents, and trace outputs. The system provides a choice of automatic or active trace, as well as an interactive stepper mechanism.

The Inference Browser window facilitates graphical debugging of the rule systems and allows interactive editing of rules. It shows the active path, and the status of slots (known or unknown, which are to be queried from the user, or which are to be deduced from rules), rules (active or inactive, to be expanded, rules pending, or fired to true or false), and goals (true, false, or unknown). It also provides a step mechanism. The developer sees how the system arrived at its conclusions by examining its lines of reasoning once the reasoning process is complete. The Inference Browser can also be used to trace the source of errors in the application's knowledge base. It was used in this project to test the rules required to satisfy various goals. It was found to be an extremely useful tool for analysing the inferencing process and debugging the system.

8. CONCLUSION

The object-oriented nature of KappaPC provides many benefits for developing knowledge based expert systems. Object-oriented programming allows you to associate behaviours with objects by storing methods in the objects. It provides a uniform interface to disparate kinds of objects with distinct behaviours, facilitating development and maintenance of an application. It provides a good way to represent active things and is useful for implementing most procedural parts of a KappaPC application. It can also be used when the procedures you want do not require the inferencing capability provided by rules. However, for certain cases it can be difficult to decide which is best and there are very few sources of reference to guide the programmer as to which is suitable in a given case.

KappaPC methods provide for the object-oriented characteristic of polymorphism. Thus different Kappa objects can have their own individual methods with the same name as the methods in other objects. This then allows the different objects to respond in their own characteristic way, to the same message put out by the application.

A limitation of KappaPC is that it supports only single inheritance, that is, a class can inherit only from one parent. For this study multiple inheritance was required, that is, inheritance from a number of parents, so additional programming was required to accommodate this.

KappaPC is quite an extensive environment and it has a large range of specialist debugging and tracing tools both for the KAL language and for the inferencing mechanism. To use it effectively a user must gain a good knowledge of the system and must, for example, understand how to integrate the object hierarchies used to represent domain objects with the production rules needed for inferencing.

REFERENCES

[1]. Hasan, K., Ramsay, B., Ranade, S., & Ozveren, C.S., (1994) An Object-oriented Expert System for Power System Alarm Processing and Fault Identification. Proceeding *7th IEEE Electrotechnical Conference*, USA

[2]. Intellicorp, (1997), Online Help KappaPC 2.4, Intellicorp Inc, Mountain View: California, USA

[3]. Kiernan, M.J., & Brown, K.E. (1996) The Application of Knowledge Based Techniques To Subsea Acoustic Data Interpretation. *IEEE Expert*, USA

[4]. Tsang, C.H.K., & Bloor, C., (1994), A Medical Expert System Using Object-oriented Framework. Proceeding *7th Annual IEEE Symposium on Computer Based Medical Systems*, USA.