STATE OF THE ART
KNOWLEDGE-BASED SYSTEM TOOLKITS

Paul W.H. Chung and John K.C. Kingston

AIAI-TR-54

March 1988

This paper was an invited presentation to the 5th annual conference on Expert Systems in Medicine, Royal Free Hospital, London, 28 March 1988.

Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

Abstract

Toolkits for building knowledge-based systems have been available for several years. The most sophisticated commercially available toolkits are *multi-paradigm* systems. Multi-paradigm systems are favoured because:

- Large programming problems can usually be divided into a number of sub-problems.
- A single paradigm may not be suitable for solving all of the sub-problems
- Multi-paradigm systems allow the programmer to choose the right tool for each sub-task.

Multi-paradigm systems typically incorporate rule-based programming, object-oriented programming and access-oriented programming. More sophisticated multi-paradigm systems may include manipulation of 'contexts' (hypothetical states of the world) and extensive facilities to aid program development. Such systems usually use a schema-based representation of data, and relationships between data. They may be designed by providing interfaces for a selection of programming languages which are popular in Artificial Intelligence (such as POPLOG[1] or Knowledge Craft[2]), by augmenting an existing system with a new language construct (such as the version of the HOPE[3] functional language which incorporates unification), or by designing a new system that supports different paradigms (such as Inference ART[4] or KEE[5]).

This talk looks at the features of three commercially available systems: ART, KEE and Knowledge Craft. This is not intended to be an evaluation of these three systems, since multi-paradigm systems cannot be compared very well by lists of options, and also the relative merits of each system may vary for different application areas.

1. Introduction

The first generation of knowledge-based systems were built using *production rules*. Well-known examples include MYCIN (Shortliffe 1976), a system which deals with the diagnosis of bacterial infections, and R1 (McDermott 1980), the current version of which is used for configuring DEC VAX computers. However, it quickly became apparent that production rules were inappropriate for some types of knowledge-based system. One solution to this problem was to use *schemata*, also known as *frames* or *units*, to represent data, and to perform inference using *object-oriented* and *access-oriented* programming. However, schema-based systems also have their limitations (Kingston 1987). This situation has led to the creation of more sophisticated toolkits for building knowledge-based systems, which are *multi-paradigm* systems; that is, they incorporate and integrate rule-based programming and schema-based programming.

This paper will give an overview of three such toolkits - the Automated Reasoning Tool (ART), produced by Inference Corporation; the Knowledge Engineering Environment (KEE), developed by IntelliCorp, Inc.; and Knowledge Craft, implemented by Carnegie Group Inc. These three products currently dominate the commercial market for multi-paradigm toolkits. All three are currently available only in Lisp implementations. The overview aims to highlight the facilities that each toolkit provides, and to point out similarities and differences between them. It is not meant to be an evaluation. Evaluations of such complex systems are difficult, because comparing them by looking at a list of

---

[1] **POPLOG** is marketed by SD-Scicon plc.

[2] **Knowledge Craft** is a trademark of Carnegie Group, Inc., of Pittsburgh, Pennsylvania. It is marketed in the UK by Carnegie (U.K.), Ltd., GSI House, Stanhope Road, Camberley, Surrey GU15 3PS, England.

[3] **HOPE** is a software package developed for research purposes at the University of Edinburgh (Burstall, MacQueen and Sannella 1980), and development has continued at Imperial College, London. All enquiries concerning HOPE should be addressed to H. Glaser, Dept. of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ.

[4] **Inference ART** and **ART** are trademarks of Inference Corporation of Los Angeles, who produce the package. It is marketed in Europe by Ferranti Computer Systems Limited, Ty Coch Way, Cwmbran, Gwent, United Kingdom NP44 7XX. For the purpose of brevity, this document refers to the product simply as 'ART', although this should not be confused with any other product with a similar name.

[5] **KEE** (Knowledge Engineering Environment) is marketed by IntelliCorp, Inc. of California. It is marketed in the U.K. by IntelliCorp Ltd, Runnymede Malthouse, Runnymede Road, Egham, Surrey TW20 9B0. **KEE**, **Knowledge Engineering Environment**, and **IntelliCorp** are registered trademarks of IntelliCorp, Inc.

options available in each system may be misleading; also, the usefulness of each toolkit may vary according to the application area of the knowledge-based system that is being built.

All three toolkits include a schema-based representation of knowledge, forward and backward chaining rules which can make use of this knowledge, and object-oriented and access-oriented programming. They also include facilities for context manipulation, and sophisticated development environments. This paper will discuss these features, and then go on to describe their application in each toolkit.

1.1. Schema representation

A schema is a data structure which groups together information about one particular object or concept. It resembles a 'record' in Pascal, or a 'property list' in Lisp. Each schema contains a number of *slots*, which represent the attributes which the object represented by the schema may have. Each of these slots may have one or more *values*. For example,

{**car-1**
REG: JB007
COLOUR: silver
MAKE: Rolls-Royce}

would be a typical schema.

All three toolkits allow *inheritance* to occur between schemata. Inheritance normally occurs between two schemata, one which represents a class of objects, and the other of which represents a subclass or member of that class. It allows a system to assume that all characteristics (slots and values) pertaining to a class are true of any member of that class. This means that information common to all members of a class only needs to be represented once in the system. This reduces the time required for knowledge base development and updating, and, in some implementations, it enables system resources to be used economically. It also simplifies the creation of objects or concepts which closely resemble other objects. Links between schemata (such as class-subclass links) are specified using slots, which are known as *relations*.

1.2. Rule-based programming

Rule-based programming typically involves rules of the form:

IF condition-1 is true
AND condition-2 is true

THEN conclude conclusion-1 is true
AND conclude conclusion-2 is true
AND execute action-1

An example of a rule might be:

IF there is a *share* which is being traded
AND there is a *dealer* who has executed *more than 20* deals in the *share* today
AND the price of the *share* has changed by *less than 3* pence today

THEN conclude that the *dealer* is possibly breaching the Financial Services Act
AND investigate how much *commission* the *dealer* gets for each deal in the *share*

Rule-based systems maintain a database of facts about the world (the *working memory*), so that they can perform reasoning; if a fact about the world matches a condition of a rule, that condition is judged to be fulfilled.

Inference using rules can proceed in a *forward chaining* or a *backward chaining* manner. When forward chaining is being used, rules fire (have their conclusions asserted and actions executed) when they have enough information - that is, when all their conditions are matched by incoming data, or by the conclusions of other rules. If more than one rule has all its conditions matched at a particular time, various *conflict resolution* strategies are used to select one rule to fire. These strategies may include *refraction* (not re-using a rule which has already fired, but is still matched by the same data), *recency* (prefer a rule matched by data which has most recently entered the working memory), *specificity* (prefer the most specific rule - that is, the one with the most conditions), and user-defined priorities. Whatever the strategy is, rule-based systems will only fire one rule before re-evaluating the conditions of all rules against the contents of the working memory. Forward chaining rules may be compiled into a network based on the Rete algorithm (Forgy 1982), which is a fast pattern matching algorithm. Using such a network speeds up the process of matching the conditions of rules against facts in the working memory.

Backward chaining systems work by focussing on a particular part of the problem (that is, a solution to the problem); rules operate only if one of their conclusions matches the solution under consideration. If a rule is used, its conditions become new foci for the system. The search terminates when a rule's conditions are matched by data existing in the system, or data input by the user in response to a question.

Rule-based systems sometimes incorporate facilities for *truth maintenance*. The essence of truth maintenance is that reasoning can be performed based on assumptions, and can be undone if any of the assumptions are found to be wrong. Some systems may explicitly alter the working memory to remove all erroneous conclusions; others may leave those conclusions intact, but mark them as inconsistent.

1.3. Object-oriented programming

Object-oriented programming is a method of drawing inferences based on a schema-based representation of knowledge. An *object* is a schema, some of whose slots contain pointers to procedures (known as *methods*) written in a programming language. This allows methods which are most relevant to a particular schema to be linked to slots within that schema, which is useful for creating a tidy and modular knowledge base. Inference occurs by sending a message to the slot to which a method is attached, which causes that method to be activated. Sending a message can be thought of as an indirect function call.

Like other values of slots, object-oriented procedures can be inherited between schemata. Some systems allow local modification to inherited procedures.

1.4. Access-oriented programming

Some knowledge-based system toolkits allow the user to define functions which are executed whenever certain data is fetched or stored. Such functions are known as *active values*, or *demons*. Such functions are useful if the value of a variable is dependent on another. Some toolkits also provide graphical front ends which use active values which are activated by moving the cursor, or clicking mouse buttons. These active values update the schema slot or slots which the graphical display represents, according to the position of the cursor, or the mouse button that was clicked. Active values can also used for constraint propagation, by making the value of one slot dependent on the value of another.

1.5. Contexts

Contexts allow a system to investigate the consequences of following several different lines of reasoning. When a program reaches a point where it must make a choice, and it has insufficient information available to make the best choice, a context can be generated to represent the consequences of each choice. These consequences can then be compared, and the best one chosen. Contexts can thus be thought of as possible states of the world, or as "hypothetical worlds".

Many knowledge-based programs require a search through a range of possible alternatives. Contexts are ideal for such tasks, because contexts effectively allow the creation of multiple knowledge bases, since contexts inherit data from parent contexts (previous states of the world, or choice points). However, generating contexts for all possible alternatives is often computationally expensive, particularly if there are many alternatives, or if the alternatives differ from each other by more than a few items. Contexts may also be used to represent the state of a time-varying system at different instances.

A system which is capable of maintaining and accessing several different, possibly conflicting, contexts, is described as a "context layered" system.

1.6. Development environments

All three toolkits are accompanied by a text editor, from which rules and schemata can be altered, and individually recompiled into the existing loaded knowledge base. This avoids having to reload the knowledge base every time a change is made. File handling facilities are available for saving schemata, and sometimes rules and Lisp functions, to a file. The toolkits also incorporate textual and/or graphical facilities for browsing the knowledge base and for tracing and debugging the rule base. Facilities are available to develop graphical user interfaces for knowledge-based systems developed using the toolkit. Finally, all the toolkits allow access to the Lisp development environment, which allows tracing of function calls, and other useful features.

Having described the facilities which are available in state of the art knowledge-based system toolkits, the implementation of such features in the toolkits can be discussed.

2. ART

This review is based on ART version 3.0. ART was originally developed on Symbolics Lisp machine hardware, and is now available on a range of Symbolics and Texas Instruments Lisp-based workstations, SUN 3 workstations, and a range of DEC VAX hardware. Inference Corporation have announced that a version of ART is being developed for IBM mainframes, although no release date has been given.

2.1. ART - Schema representation

ART can represent data using either schemata or arbitrary propositions (known as *facts*). When a schema is created, it is compiled into a series of facts with three elements - a slot name, the schema name, and the value of that slot. The value of a slot can be an arbitrary Lisp structure. Inheritance occurs between schemata, and multiple inheritance (inheritance of several values from different schemata) is allowed - ART creates a separate fact for every value of a slot, whether asserted directly or inherited.

For every slot, a schema exists which defines the behaviour of that slot. This is a powerful feature, because the user can define his own slots, and specify the characteristics of the slot-behaviour schema himself. This allows the user to define whether slots can contain one or many values, whether the slot can be inherited, and so on. Some slots are used as *relations* - links between schemata across which inheritance may occur. The user can define whether such relations are transitive or reflexive, whether they can have inverses, and, in certain circumstances, whether the creation of a relation should automatically cause other relations and schemata to be generated.

2.2. ART - Rule-based programming

ART's rules are based on production rules, but have been considerably extended. The left-hand side of an ART rule may include facts or schemata, truth maintenance operators, and arbitrary Lisp expressions. Forward chaining rules use an IF-THEN format; backward chaining rules are the same as forward-chaining rules, except that they have a goal pattern on their left-hand side. When a goal pattern is satisfied, a backward chaining rule competes for firing in the same way as forward-chaining rules do. ART performs conflict resolution based on user-defined rule priorities (known as *salience*); it only uses recency if saliences are equal. The right-hand side of an ART rule allows modifications to facts or schemata, and the execution of arbitrary Lisp functions.

ART also allows the user to improve the efficiency of a system by restricting the invocation of backward-chaining rules, and by optimising the construction of the join network (which is central to the function of networks based on the Rete algorithm).

ART's rule-based programming is therefore very powerful, despite one or two limitations - for instance, ART's truth maintenance is unable to reassert facts which have been retracted, which limits its usefulness. If its conflict resolution strategy is inadequate, it is possible for the user to implement his own; however, this may slow down execution speed significantly (Inder 1988).

2.3. ART - Object Oriented Programming

ART supports a simple syntax for creating methods and sending messages. The methods themselves (known as *actions*) are defined using Lisp functions. Actions can be defined as *primary* actions, *before* or *after* actions, or *whopper* actions. Primary actions attached to an object are normally executed when an object receives a message; if any before or after actions exist, they will be executed before or after execution of a (possibly inherited) primary action. Whoppers may invoke any of the other three actions if they exist, and the results produced by actions and/or passed to other actions may be modified *en route* by the whopper. Whoppers therefore allow unlimited local modification of actions.

2.4. ART - Access Oriented Programming

ART supports active values which are defined using schemata. They can be executed just before or just after a slot's value is read, asserted, changed, or deleted. The current implementation of active values creates a lot of facts if active values are inherited; it "constructs a second inheritance hierarchy isomorphic to the one motivating the inheritance" (Inder, 1988).

2.5. ART - Context Manipulation

ART supports a context mechanism which allows rules to create, delete, or merge contexts. In ART, a lot of context generation takes place automatically. ART tries to match the conditions of rules in any context and, if a rule has conditions which are matched in two or more different contexts, ART will automatically merge those contexts to create a new context in which that rule can fire. ART allows the user to specify contradiction rules; if the conditions of these rules are fulfilled in a particular context, that context is *poisoned* (marked as inconsistent, and deleted). Contradiction rules can be used in conjunction with ART's truth maintenance system, which allows the user to specify an *assumptive base* for a context. Contexts with a specified assumptive base are automatically placed at the optimal position in the context hierarchy by ART.

ART will automatically poison a context if a single-valued slot in a context can inherit multiple values from other contexts; and it will let the user *believe* a particular context, thus deleting all others. ART also increases the efficiency of the context mechanism by not actually copying any information into a new context - it simply marks which facts may be inherited by that context, and which may not. ART's context mechanism therefore supplies a very wide range of features.

ART also supplies the *viewpoint* mechanism, which allows the user to maintain multiple distinct but interacting context hierarchies. This allows the user to factor out different kinds of knowledge. For instance, a context hierarchy might be used if a stockbroker wanted to examine the consequences of investing in a particular stock, given any one of a number of future economic conditions. If the stockbroker had two clients, one of whom was interested in a very high rate of return, and the other was most interested in protecting his initial investment, then he might use viewpoints to allow him to make decisions using two different strategies, but based on the same context

2.6. ART - Development environment

ART's development environment is mouse and menu-based.[6] The ART studio provides a top-level menu which can be used to access a wide variety of information and optional actions. Among other options, the user can view the agenda of rules which are about to fire, examine particular rules, examine and maybe modify the knowledge base, ask for rules to be traced, or for rule firings to be justified, display a diagram of an inheritance hierarchy (schema or context hierarchies), which may be dynamically updatable, and obtain statistics about the program to aid in improving program efficiency. The ARTIST[7] icon editor allows icons to be created using the mouse and menu options; schemata are

---

[6] ART also provides a "scrolling studio" for use on text-only terminals which are remotely logged in to a host which runs ART.

[7] ART Interface Synthesis Tool. **ARTIST** is a trademark of Inference Corp.

created automatically to represent icons. As for file handling, the version of ART that runs on Symbolics hardware allows the user to create files (and, more importantly, alter and then incrementally recompile rules, facts and schemata) using Symbolics' own ZMACS (emacs) editor. On the SUN 3 workstation, similar facilities are provided by GNU Emacs,[8] which is supplied with ART.

3. KEE

This review is based on KEE 3.05. KEE was originally developed on Xerox Lisp hardware, and is currently available on most Lisp workstations (Symbolics, TI Explorer and Xerox 1186), and a range of 'engineering' workstations (SUN, Apollo, DEC VAX and IBM 6150). A delivery version of KEE is available on a Compaq 386 PC, and a full development version of KEE for a 386 PC will be released shortly. A delivery version of KEE is also available on IBM PC ATs connected to a VAX host. IBM and IntelliCorp have recently announced that a development version of KEE will become available on IBM mainframes.

3.1. KEE - Schema Representation

The majority of data in KEE is represented using schemata, which are known as 'units' in KEE, although it is also possible to represent unstructured arbitary facts. KEE supports two relations between units, allowing the user to define either a member or a subclass of a class. These two relations define the inheritance paths between units. Multiple inheritance is permitted. Each unit may contain two types of slot - slots which can inherit values from other units, and slots which cannot. KEE does not directly support user-defined relations, although this facility is available in SimKit, which is an extension to KEE, providing extra facilities for building knowledge-based simulation systems.

In KEE, each slot is provided with a series of constraints or facets which constrain the possible values of the slot. The facets of a slot allow the user to control the inheritance of multiple values by the slot, to specify the type or range of a slot value, to put cardinality restrictions on a slot value, to attach active values to monitor a slot value, and attach graphical images which display (and may permit updating) of the slot value. The value of a slot is stored in one of its facets.

3.2. KEE - Rule Based Programming

KEE incorporates two types of rules - *standard* rules, and *deduction* rules. Deduction rules are used for reasoning where the conclusions of a rule will always be dependent upon the premises of the rule. They are implemented using KEE's truth maintenance system.

Rules are partitioned into rule-sets, and reasoning is restricted to the rules within selected rule-sets. In KEE the same rules can be used for both forward and backward chaining; if a fact is asserted, and a particular rule-set is invoked, then forward chaining may occur; if a query regarding a fact (possibly containing one or more variables) is made, then backward chaining may occur. Intermixing of forward chaining and backward chaining is possible, as is the simultaneous consideration of several rule-sets.

The KEE user is provided with a number of switches to control the strategies used in rule selection. The backward chaining strategies include depth-first, breadth-first, and best-first search, as well as user defined strategies. For forward chaining the user can choose between strategies based on recency, specificity, or user-defined priorities; KEE also allows the user to define his own strategies.

3.3. KEE - Object Oriented Programming

The object-oriented facilities of KEE are well developed, since KEE's development was based on UNITS, an object-oriented system. Lisp functions or lambda expressions can be attached to the value of slots. Messages to invoke these methods are sent using a simple command that specifies the unit and the slot to which the message is being sent, as well as any arguments that the method function might require. Inherited methods can be overriden by local methods, or modified by attaching extra local code before, after, or "wrapped" about, the inherited code.

---

[8] GNU Emacs is a piece of software licenced free of charge by the Free Software Foundation for use or redistribution by anyone, on the condition that no attempt is made to charge for its use.

3.4. KEE - Access Oriented Programming

An active value can be defined by creating a unit. This unit can then be attached to the slot(s) to be monitored by adding the name of the unit to the appropriate facets of the slot(s). The slots of the active value unit define what actions should be taken whenever a monitored slot value is read, asserted, added, or removed.

3.5. KEE - Context manipulation

Contexts in KEE are known as *worlds*. KEE allows contexts to be created, deleted or merged. When a slot value is changed in a world, a new copy of that slot is created, which is considered to exist in the specified world. Slots and values which exist within a world may be inherited by other worlds. The user is allowed to specify that certain worlds may not be merged, or to define rules that detect inconsistent states within a world. KEE will also declare a world to be inconsistent if it violates any restrictions on slots, such as the cardinality, type, or range of slots. KEE allows inconsistent worlds to be 'resurrected' if the inconsistency disappears.

Chaining of rules can take place within the context of a particular world, and the conclusions of each rule may be asserted, creating a new world.

3.6. KEE - Development Environment

KEE provides an extensive development environment. As well as an EMACS editor, it provides a schema and knowledge base browser, a graphics interface development package, a language for querying the knowledge base, and various sorts of debugging and trace information.

The graphics interface development package is mouse and menu based. It is based around KEEPictures, an extensive range of pre-defined graphic items which can be defined using units (and animated by changing the values of slots). These graphic items include histograms, gauges, switches and thermometers. Another feature of the package is ActiveImages, which are graphics which allow direct access to the values of slots. An ActiveImage displays a slot value, alerts the user if the value reaches a pre-defined limit, and allows the user to change the value using the mouse.

The language for querying the knowledge base is known as TellAndAsk, which is an English-like version of Lisp. It allows the user to interrogate or modify the knowledge base.

KEE's debugging information includes an viewer for the agenda (conflict set) for forward chaining rules; dynamic graphic traces for forward chaining rules, backward chaining rules, and worlds; textual traces for rules and methods; a rule cross referencer; and the (machine-dependent) LISP debugger.

KEE automatically saves units, compiled methods, and the source code of methods, into a file.

4. Knowledge Craft

This review is based on Knowledge Craft version 3.1. Knowledge Craft version 3.1 was developed on Symbolics hardware, and is available on Symbolics and TI Explorer Lisp machines, SUN 3 workstations, and DEC VAX minicomputers and workstations. Knowledge Craft is built around Carnegie Representation Language (CRL), which is an extension of the Schema Representation Language designed at Carnegie-Mellon University in Pittsburgh.

4.1. Knowledge Craft - Schema Representation

Almost everything in Knowledge Craft is represented using schemata. Knowledge Craft allows the user to specify the characteristics of a particular slot by using a *slot control schema*. A slot control schema is a schema with the same name as a slot which appears in another schema. It allows the user to put demons (active values) on a slot, restrict the values that can fill a slot, or restrict the schemata that can contain that slot.

Slot control schemata also allow the user of Knowledge Craft to define relations (slots which link one schema to another), and to restrict which slots and values can be inherited across each relation. The user is allowed to define the transitivity of a relation; for instance, if a spark plug is a COMPONENT-OF an engine, and an engine is PART-OF a car, Knowledge Craft allows the user to specify that a spark plug should be considered to be a COMPONENT-OF a car. Transitivities and restrictions allow the user considerable subtlety in defining the semantics of a relation.

Knowledge Craft also uses schemata to represent *meta-knowledge* about the knowledge base. Meta-knowledge can be used to locally override or replace slot control schemata, or to record which schema a value has been inherited from.

Knowledge Craft permits multiple inheritance, as long as the appropriate global variable is set correctly.

4.2. Knowledge Craft - Rule Based Programming

Knowledge Craft uses CRL-OPS for forward chaining. CRL-OPS is an enhanced version of OPS5, which is a production rule system. The enhancements include allowing a limited range of Lisp functions on the left-hand side of a rule; allowing CRL, Lisp, CRL-Prolog, or OPS5 commands to be executed on the right-hand side of a rule with a minimum of awkward syntax; and allowing schemata to be translated into CRL-OPS' working memory. Conflict resolution in CRL-OPS is based on recency, and, if rules are of equal recency, specificity is used. CRL-OPS rules are compiled into a Rete network.

Backward chaining in Knowledge Craft is handled by CRL-Prolog. CRL-Prolog is approximately equivalent to Edinburgh Prolog (see Clocksin & Mellish 1987) in functionality, although its syntax is more reminiscent of Lisp than Prolog. It provides functions to permit access to Lisp and CRL functions. It also has a simple schema interface which allows CRL-Prolog to perform pattern matching on schemata and slots.

4.3. Knowledge Craft - Object Oriented Programming

Knowledge Craft supports object oriented programming. It allows a Lisp function or a lambda expression to be defined as a method and attached to a slot. Methods are executed using a special function which specifies the schema and the slot. Methods can be inherited. Knowledge Craft does not provide facilities for executing additional local methods as well as inherited methods.

4.4. Knowledge Craft - Access Oriented Programming

Active values are known as *demons* in Knowledge Craft. A schema is defined which represents the demon, and it is attached to the relevant slot using a slot control schema. The slots of the demon schema indicate what operations on the slot value will cause the demon's function to be executed, whether that function will be executed before or after the operation that activated the demon, and whether the demon will block that operation, alter its parameters, or leave it unaffected. Demons must be explicitly enabled before they can be used.

4.5. Knowledge Craft - Context Manipulation

Knowledge Craft allows contexts to be created, and deleted. It also allows two contexts to be combined if one is a descendant of the other. Knowledge Craft provides no facilities for explicitly declaring a context to be inconsistent; if two contexts which are combined have conflicting values, one value overwrites the other.

In Knowledge Craft, when a slot value is altered in a context, the whole schema is copied into that context. Schemata can be moved or copied to different branches of the context hierarchy.

Knowledge Craft includes a range of methods of constraining CRL-OPS rules to match in a particular context, or in contexts on one branch of the context hierarchy.

4.6. Knowledge Craft - Development Environment

Knowledge Craft uses the system editor on Lisp machines, and supplies an EMACS-like editor on the SUN workstation and the VAXstation. These editors allow incremental recompilation of rules, schemata, or Lisp functions which have been altered while a knowledge base is loaded. Knowledge Craft also supplies some facilities for debugging the knowledge base, including the Knowledge Craft workcentre, which can be used to issue CRL functions; the CRL-OPS and CRL-Prolog workcentres, which supply a variety of windows which give various sorts of trace and debugging information for CRL-OPS and CRL-Prolog; a graphical schema network editor; and two schema editors. These facilities are mouse and menu-based.[9]

There are also some facilities for development of schema-based graphics, facilities for redefining some of Knowledge Craft's error handling functions, facilities for handling events, queues and time for simulation applications, and a tool for saving schemata created using the user interface tools to a file.

5. Possible extensions to knowledge-based system toolkits

Although these toolkits provide an impressive range of facilities for building knowledge-based systems, further extensions are still possible. Perhaps the most obvious is integration with more conventional computer hardware; although single-user workstations are coming down in price, and some of the toolkits run on VAX minicomputers, many companies who are interested in expert systems are unwilling to buy anything that cannot be integrated with their existing hardware (often supplied by IBM). At the very least, toolkits should be able to access database management systems and conventional languages; apart from some facilities provided by KEE to access a range of databases or the C programming language, the user of a toolkit must write his own Lisp functions to access other packages. There are also worries about the usefulness of knowledge-based systems for real-time problems, because they typically run more slowly than conventional programs of a similar size. There are two main approaches to these problems. One is that the systems should be improved and extended to provide the above features. Product announcements from the toolkit vendors suggest that some features will indeed become available in later versions. The other approach is to treat the toolkits as engines for rapid prototyping of knowledge-based systems. This approach suggests that a prototype system is developed using a toolkit, and is then used as a specification for re-implementing a delivery system on different hardware. This approach is becoming more feasible as more sophisticated mainframe-based or PC-based knowledge-based system delivery environments become available[10]

Other possible extensions to toolkits would depend on who would be using the toolkit. Those who are familiar with numerical certainty factors might be surprised to find that the toolkits do not provide certainty factors. However, the most common implementation of certainty factors is both easy to implement using ordinary arithmetical operators, and theoretically unsound (see Ross 1986). Or the target user might be a domain expert with minimal computing expertise; in such cases, the toolkits would require very sophisticated user interfaces for building knowledge bases, incorporating semantic checking, cross-referencing, version control, natural representation of spatial and temporal relations, and the ability to control inferencing at a high level of abstraction (Mettrey 1987). Such facilities are very unlikely to be incorporated in toolkits in the near future, because there are considerable theoretical difficulties with any system requiring an understanding of semantics, and because the effort involved in building and maintaining such facilities probably exceeds the current demand for them.

Conclusion

Multi-paradigm toolkits provide a wide range of features, which aim to allow systems to be developed very rapidly. The user of a toolkit today needs a basic knowledge of Lisp, and it is helpful to have a grounding in the principles of Artificial Intelligence programming. Toolkits are very useful for designing prototypes of knowledge-based systems rapidly, but would benefit from better integration with more conventional hardware and software.

---

[9] Knowledge Craft provides a languages-only version, which is available on text-based terminals remotely logged in to a host. This version does not supply any graphical editors or workcentres, although debugging information can still be obtained using textual commands.

[10] Multi-paradigm toolkits for developing knowledge-based systems (such as **KEE** or **GoldWorks**) are becoming available on PCs. However, such toolkits currently run best on PCs which use the powerful 386 chip; the price of these PCs is comparable to that of some workstations. It will be interesting to see how popular and useful the PC versions of multi-paradigm toolkits are.

GoldWorks is marketed by Artificial Intelligence Ltd, Intelligence House, 58-78 Merton Road, Watford, Herts, WD1 7BY

Acknowledgements

Many thanks are due to Ian Filby, Robert Inder, and John Fraser of the AI Applications Institute for their comments on earlier drafts of this paper.

References

Burstall R.M., MacQueen D.B and D.T. Sannella (1980) HOPE: An Experimental Applicative Language. *CSR-62-80*, Department of Artificial Intelligence, University of Edinburgh.

Clocksin W.F. and C.S Mellish, (1987) Programming in PROLOG (Third edition). New York: Springer-Verlag.

Forgy C.L. (1982) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, vol 19 no 1, 17-37, Sept 1982.

Fraser, J.L. (1987) Overview of KEE. In *Airing, no 2* AI Applications Institute, University of Edinburgh.

Inder R., (1988) The State of the ART. *AIAI-TR-41*, AI Applications Institute Technical Report Series, University of Edinburgh.

Kingston J.K.C., (1987) Rule-based Expert Systems and Beyond: An Overview. In *Proceedings of the Expert Systems Workshop*, British Association of Accountants Conference 1987. University of Glasgow: Glasgow Business School.

Kingston J.K.C., (1988) An Overview of Knowledge Craft: Parts 1 and 2. In *airing no 3 and no 4*, AI Applications Institute, University of Edinburgh.

Lurent J., Ayel J., Thome F. and D. Ziebelin, (1986) Comparative Evaluation of Three Expert Systems Development Tools: KEE, Knowledge Craft, ART. *The Knowledge Engineering Review*, vol.1, no.4, pp.18-29.

McDermott, J. (1980) R1: An Expert System in the Computer Systems Domain. In: *Proceedings of AAAI-80*, American Association for Artificial Intelligence, pp 269-274.

Mettrey W., (1987) An Assessment of Tools for Building Large Knowledge-Based Systems. *AI Magazine*, Winter 1987: 81-89.

Richer, M.H. (1986) An Evaluation of Expert System Development Tools. *Expert Systems*, vol.3, no.3, pp 166-183.

Ross P., (1986) Expert Systems M.Sc. Course Notes. Department of Artificial Intelligence, University of Edinburgh.

Shortliffe E.H., (1976) Computer Based Medical Consultations: MYCIN. New York, Elsevier.

Wall R.S., Apon A.W., Beal J., Gately M.T. and L.G. Oren, (1985). An Evaluation of Commercial Expert System Building Tools. *Data & Knowledge Engineering*, vol.1, pp 279-304.