The Knowledge Engineering Review

Experiences with a knowledge engineering toolkit: an assessment in industrial robotics

V. ROBINSON, N.W. HARDY, D.P. BARNES, C.J. PRICE, M.H. LEE
Robot Research Group, University College of Wales, Aberystwyth, Dyfed, UK

Abstract

Some of the issues involved in purchasing and using one of the powerful Knowledge Engineering Toolkits are examined. A case study problem is described and several products are reviewed as potential environments for implementing a solution. Experiences of using one system, ART, are described and observations are made regarded the selection, evaluation and use of such packages.

1 Introduction

This paper describes early experiences with the knowledge engineering toolkit ART from Inference Corporation. It is based around work on a particular knowledge-based application where previous work had been undertaken using the procedural language RTL/2. The application does not require a "conventional" expert system in that it is not providing a man-machine dialogue to give assistance in a particular area of expertise. Rather, it is an enhanced programming and control system for industrial robot workcells.

Industrial robotics and the wider field of flexible manufacturing systems require software to coordinate the activities of multiple and varied effectors and sensors to realize prespecified tasks in the real world. Typically there will be considerable complexity in both the task specification and the workcell environment and the latter will contain significant uncertainties.

The effectors will range from simple valves, pistons and rotary drives to jointed robot arms with servoed position control. The software to control individual devices will range accordingly from simple to highly sophisticated, possibly incorporating real-time closed loop control and substantial numerical computation for the calculation of positional and dynamic parameters.

Sensors will range from simple binary switches, providing unambiguous one bit input, through devices such as range and proximity detectors, requiring traditional signal processing software, to information rich vision and tactile sensors which may require sophisticated hardware and advanced algorithms to yield useful data at an acceptable rate.

Software (or more typically firmware) is often closely associated with particular devices, creating a workcell which is a collection of semi-intelligent devices, loosely coupled and requiring coordination. The software to achieve the coordination may be termed application programs.

Current commercially available applications programming facilities tend to be robot centred; treating all other devices as peripherals. They provide a conventional procedural language with special types, operators and statements for robot control and simple sensor input [Bonner]. Various techniques can be used to supplement these facilities. For example numerically controlled machines may form part of the workcell and programmable logic controllers may be used to handle large numbers of signal lines.

The perceived inadequacy of procedural language environments may be partly attributed to the lack of modern programming techniques, constructs and data structures that appear in MODULA-2 or ADA for example and to the absence of associated tools such as integrated editors and debuggers. The technology exists to make good these deficiencies but more fundamental problems remain with this general approach.

These problems concern the nature and level of abstraction of the specification of the task to be performed in the workcell. Currently, specifications are presented in a purely procedural form at a very low level. All required activities must be fully described in the available language. The descriptions are actuator centred, specifying what actions each must perform. No indication is given of what effect these actions are intended to have or why they should be performed in the given order. Many attempts to improve robot programming have been directed towards object-level (or task-level) systems. In such systems the task specification describes the effects of actions on the state of the objects (Lozano-Perez).

The immediate benefit of object level programming is the reduction in programming effort required for a particular application. A description of what must be achieved is typically much shorter than a description of how to achieve it. The existence of knowledge about goals also allows for dynamic replanning of actions. If the execution of an action centred plan fails then there can be no possibility of continuing beyond an exact retry. If, in contrast, one method of achieving a goal fails another method may be generated and used.

The translation of an object level program to an effector level one for execution in the workcell is largely a planning activity and requires AI techniques. Grasping, gross movement, parts mating and assembly all require details of positions, control strategies, sensor usage and expectations. A model of the workcell forms the foundation for such systems and this may include both symbolic and geometric components.

Our work concerns the use of AI techniques to reduce programming effort and increase the effectiveness of robot workcells by attacking the problem of detection and response to errors in the environment. (The anticipation, detection and correction of such problems accounts for a large part of most effector level programs.) Originally we used the procedural language RTL/2 which supports structured programming constructs but the techniques widely used in AI, such as frames, inheritance and pattern matching are not provided. We implemented these facilities ourselves to discover where such techniques might be used. Later, convinced that these facilities are necessary to build an autonomous, intelligent robot control system, we bought ART and reimplemented our system.

By using a particular application with both ART and procedural implementations, various comparisons can be made. We believe that there are three important features of our problem area that provide a focus for our comparisons.

1 *Prototyping.* The work was carried out in a research environment where alternative approaches must be attempted and compared. In such an environment design criteria are not rigidly fixed and experimental versions of systems are often later discarded. It was therefore important to be able to achieve acceptable functionality rapidly, possibly at the expense of elegant or efficient methods.
2 *Real-time device interfaces.* The application involved interfacing with a wide variety of external devices, some requiring real time operation. This requires the provision of primitive real-time input and output routines and, ideally, interrupt handling.
3 *Support for team development.* It is essential that the work can be undertaken by a team. This is due to the size of the project and the variety of expertise needed.

In reporting on our experiences and making comparisons we have attempted to avoid comments on issues which arise from the marked differences between the hardware on which RTL/2 and ART were being run.

2 The case study—AFFIRM (Aberystwyth Framework For Industrial Robot Monitoring)

The application concerns the implementation of a robot supervisory system. It is designed to provide programmable control of an industrial robot work cell. It is a system with knowledge of how to control and integrate the actuators and sensors that are present in a typical robot assembly cell. The knowledge based approach encourages information about the work-cell domain, and its current context or focus of interest, to be made explicit. This knowledge is then available for interpretation

by monitoring or diagnostic processes throughout the execution of a particular assembly task. Such information is essential for the diagnosis of task errors and their ultimate recovery.

2.a AFFIRM's knowledge

AFFIRM's knowledge representation method is based upon the frames formalism [Minsky]. The various types of frame used to represent supervisory knowledge can be summarised as follows:

1 EVENT frames. These contain the knowledge of the plans, intentions, and ultimate goals that a user has for a particular robot work cell.
2 EQUIPMENT frames. These are concerned with the various devices and components that are available within the work cell in order to accomplish the various EVENTs.
3 EXPECTATION frames. These are used to contain the expectations that a user has for when a particular EVENT occurs using particular pieces of EQUIPMENT.

In keeping with the formalism, all these frame types can be organised into a hierarchical structure. Typically it is the EVENT frames that form the core of this structure. The slots within these frames are associated with EQUIPMENT and EXPECTATION frames and with other EVENT frames—hence the hierarchy. They can contain both declarative and procedural knowledge; for example an EVENT frame can not only describe how to achieve some sub-event but also actually cause that sub-event to be achieved. It is this structure that permits a full contextual description of what is happening within the work cell to be maintained.

When an EVENT frame has slots associated with other EVENT frames, then these slots are contained within a rule structure. This allows a pre-condition and a post-condition to be specified for each slot. For example, an EVENT frame might be concerned with how to activate various devices within the work cell. Consequently it might contain the knowledge of how to activate the work cell jigs and feeders. At any one point in the assembly, only one of these devices will be required. A slot's pre condition allows this selection procedure to take place, the condition specifying when a particular slot is relevant. The post condition can be used to alter the default flow of slot relevance.

The top of the hierarchy contains an EVENT frame for a high-level job such as "Assemble Gearbox". To achieve this, various sub-events will need to be performed. They embody such knowledge as pick and placing an object, inserting an object, feeding an object. Likewise, to accomplish these events further sub-events are required. These contain the knowledge of how to accomplish and monitor the higher level events. The event sub-division continues until at the bottom of the hierarchy is the knowledge of how to activate a specific actuator or sensor.

Every slot can be regarded as a predicate returning success or failure. A post condition can be specified and depending upon the value returned, the normal sequential first slot to last slot relevance can be altered. This mechanism allows additional sub-events to be performed when some assembly plan adaption is required; for example, when an error occurs.

EVENT frames inherit context specific knowledge from other EVENT frames. This can occur both down and up a frame hierarchy, i.e. forward or backwards. For example, should an EVENT frame be concerned with moving an assembly component then this frame would initially contain default values for some slots. When the frame became relevant then the specific values for these slots would be inherited from the preceding relevant frame. Should an event cause these specific objects to be compounded e.g. when a robot grasps a component then a new object is created "robot + component", then this new object is back inherited to the previous EVENT frame in order that it can then be forward inherited when the next frame slot becomes relevant.

2.b AFFIRM's control

Figure 1 shows the basic operations that are performed when AFFIRM is executed. The basic centralized control facility is a stack structure called the Agenda. The first operation is to place a copy of the highest level EVENT frame on the Agenda. This frame's control attempts to instantiate its

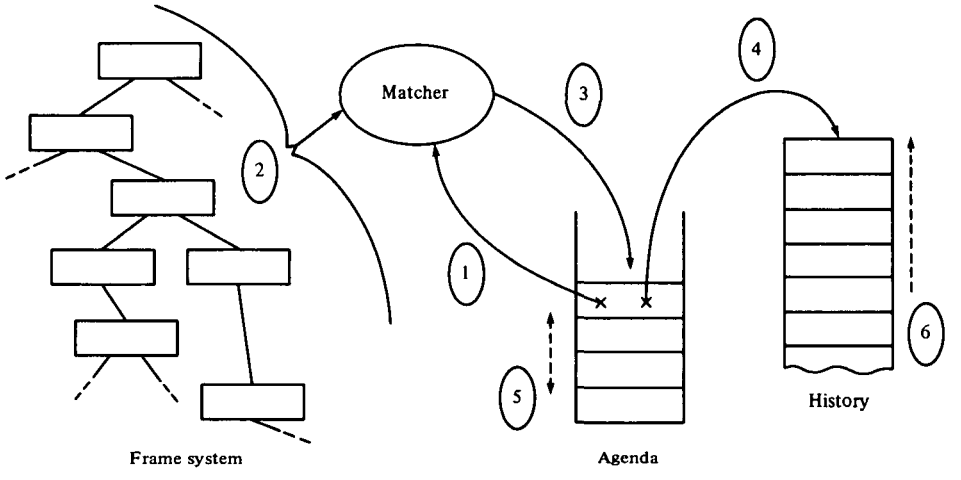

Figure 1

slots in a first to last sequence provided no pre or post conditions are present. Instantiation is performed by a frame requesting (1) that the system Matcher search the frame system (2) to find the EVENT frame with which a particular slot is associated. Once found, a copy of this frame is made and this too is placed on the Agenda (3). Any forward inheritance of specific objects then takes place making the frame an "instance" of the previous archetype. This frame will then attempt to instantiate its slots causing other EVENT frames to be added to the Agenda. This will continue until effectively the bottom of the frame hierarchy is reached.

At this point, the instantiation of the lowest frame causes an actuator or sensor to perform its function. Using the knowledge stored in an EXPECTATION frame, such activity can be validated. If it is successful, then any necessary object compounding and object location updating takes place, this information being stored in the appropriate EQUIPMENT frames. Any back-inheritance takes place and the frame is removed from the Agenda and placed in the system's History (4). Control is returned to the frame on top of the Agenda, in order to instantiate any further slots.

The process of slot instantiation and success causes frames to be added and removed from the Agenda (5) and the system History to steadily grow (6).

Should a slot fail due to some assembly error and if a post condition for this result is present, then instantiation can be passed to the frame containing appropriate error recovery slots. When no post condition is present, or if it is not appropriate, then the system will be halted. When this occurs, the Agenda and History contain a current and past record respectively, of all the work cell activities. Hence they can be examined for the purposes of error diagnosis.

Due to the pre and post condition facility in EVENT frames, the AFFIRM knowledge structure can be regarded as a hybrid between the frame and production rule formalisms.

Current industrial robot programming languages provide the user with the information that a "MOVE" to "LOCATION-X" has just been performed by the robot and that "SENSOR-Y" has a "VALUE-Z". For error diagnosis this information has minimal value. As AFFIRM has an entire hierarchy of information above this level, it is able to provide the type of information that is essential for diagnosing assembly errors.

3 Using a procedural language

During 1983 and 1984 AFFIRM was implemented in RTL/2. At that time a procedural language was preferred to an AI language because no such implementations suitable for the available hardware and operating system and adaptable to controlling external devices were known to exist. Also it was believed that suitable facilities for the AI work could be developed. RTL/2 was chosen in particular due to the availability of in-house expertise, a proven implementation and the possibility of interfacing to or incorporating existing software.

3.a The implementation

3.a.1 RTL/2

RTL/2 provides no higher level programming facilities beyond a modest set of data types (with the possibility of user defined structures), normal procedural language control structures and a subroutine library which includes a comprehensive but low level I/O package, trigonometric functions and a small subset of operating system calls.

Essentially RTL/2 is similar to PASCAL in its programming constructs but RTL/2 also has modules and allows them to be compiled separately.

A very small amount of the implementation required assembler segments within RTL/2. This amounted to a few tens of lines and was all associated with addressing external devices and calling operating system routines not directly supported by the RTL/2 library.

3.a.2 Data Structures

The first requirement was for data structures to represent frames. Each AFFIRM frame type was represented by in RTL/2 by a separate user defined type. For each of these it was necessary to write a suite of service procedures to support them in a consistent and efficient manner. These suites included the following.

1 Disc input and ouput routines. For long term storage (and later as backing store during running) it was necessary to hold knowledge on disc. It was essential that access be fast and accordingly binary dump formats for each frame were devised. Specialised I/O routines were needed for each type.
2 Display output. Both semi-graphical output to screens and textual output to printers were required for examining and displaying knowledge.
3 Creation and deletion. During knowledge entry and use it was necessary to modify frame systems by the addition and deletion of frames or frame components. It was therefore essential that such components be freely available and reusable.

Both declarative and procedural knowledge were represented within frames. RTL/2 maintains a strict distinction between the two.

3.a.3 Procedural knowledge

Procedures cannot be constructed or dissected at run-time in most conventional languages, and certainly not in RTL/2. This means that the inclusion of new procedural knowledge in AFFIRM requires the recompilation and relinking of a module.

A proportion of the procedural knowledge for control is contained within the frames themselves, as references to procedures available in the Run Time System. The remaining procedural knowledge for execution was in the form of the Run Time System executive. This latter represented the more generalized control knowledge. No programmed inspection of either type of control knowledge was possible.

The device-specific procedural knowledge for driving actuators and sensors in the world was also available in the Run Time System but referenced from the declarative knowledge by symbolic names. Explication of this was impossible but due to the rigid definitions of its use implicit knowledge of its function was held by the system. It was therefore a module of the run time system which required recompilation when new procedural knowledge was added.

3.a.4 Declarative knowledge

Declarative knowledge can be constructed and dissected at run-time but the input required special software. Two utilities were built. First we developed an editor. It provided guidance and simple

consistency checking of the input and perusal or modification of the knowledge base. Elaborate and distinct facilities were required for each frame type. Due to the complexity of the data a semi-graphical format was required for this.

A second knowledge input tool was constructed for the more constrained and directed input of specific types of knowledge associated with sensory and actuator data.

The data structures, their service routines and the knowledge input tools provide a system for the structuring and storage of knowledge in the form required by the application.

3.a.5 The run time system

The run time system combines the declarative knowledge with the procedural knowledge mentioned above, to supervise the running workcell. Three additional types of facility were also required.

1 The agenda was composed of a linked list of suitable frames and the developing world model was a simple tree of frames. This area illustrates a significant benefit of RTL/2 for this work. The modules already developed for manipulating frames in the editor could be used again without alteration or fear of unwanted interaction.
2 An interrupt handling system was included, based upon that of the operating system but extended into the frame environment. Interrupts from devices resulted directly in the (asynchronous) update of information within particular frames. Synchronously with a particular level of frame execution the occurrence of an interrupt was checked for and relevant frames polled and the interrupt serviced in terms of frame execution.
3 Software for the interpretation of the pre and post conditions on EVENT frame slots was provided.

3.b Future requirements

In planning to move forward from the RTL/2 implementation we saw a number of shortcomings in our programming facilities and also certain advantages. These latter we wished to retain in a new system, or at least would wish to have in any future production system for a non-experimental version.

3.b.1 Enhanced facilities sought

The shortcomings can best be expressed as a "wish list" of desirable features and this can be divided into general and specific features.

General features

1 A higher level of programming with more powerful and flexible constructs and greater support for abstraction.
2 A richer software component pool.
3 An incremental development environment, contrasting with the previous edit-compile-link-load-run system (overall execution speed is not critical).

Specific features

1 Easier procedural knowledge integration, and the possibility of dynamic creation and inspection of it.
2 More flexible data structures with system support, especially in the areas of I/O, displays, variants and dynamic management.

3 Integrated, intelligent, graphical knowledge editors (or suitable building blocks for creating them).
4 Management tools for the control of software production, the encouragement of some form of project standards and the close integration of documentation.

3.b.2 Perceived advantages of procedural languages

These could broadly be grouped under the heading of software engineering in that they concern the production of safer, more reliable and more understandable code.

1 Facilities for clear, full control of knowledge application. This is a vital requirement in a robot control system.
2 Modularity of code with its benefits for teamwork, reusability and ease of implementation modification.
3 Suitability for hardware interfacing.
4 The commonly understood programming paradigm with its advantages in communicating results.

4 Selecting an AI tool

4.a Examination of the possible choices

As has been mentioned above, some sort of artificial intelligence system development tool was desired for our new project. We surveyed the various alternatives available and classified them into four groups.

4.a.1 AI languages

This category includes Prolog, POP11 and many distinct dialects of LISP. AI languages are the most general tools available, giving the power of a conventional computer language plus some AI features such as backward chaining in Prolog.

The main disadvantage of this category is that extra work must be done to implement all the control and knowledge representation features required by most applications.

An additional drawback occurs because these AI languages are very unconstrained. System builders have to impose their own house rules and working structures if software engineering principles are to be employed. The very flexibility of these systems means that there are few restrictions and so the programmer carries a great burden of responsibility to check that all interactions between modules are as desired. For example, the lack of type checking and the range of side effects can lead to subtle bugs of considerable complexity.

4.a.2 AI language environments

This is a less obvious category. Some of the AI languages have been built into development environments where the environment is at least as important as the basic language provided. Poplog is one example of this. Poplog contains POP11, Prolog and CommonLisp, put together so that it is possible to write the different parts of an application in the most appropriate language. An intelligent editor and a highly interactive system are provided, giving the knowledge engineer the ability to try out ideas very rapidly.

Other examples of such systems are mostly tied to some particular fairly expensive Lisp-based computer—for example, ZetaLisp and Flavors on the Symbolics, InterLisp-D on the Xerox 1108.

With AI language environments, the knowledge engineer still has the need to design his own control and basic representation features. He also still has a generally applicable conventional programming language at his disposal.

4.a.3 AI toolkits

Several comprehensive, commercially-available knowledge engineering toolkits have appeared over the last few years (KEE, ART, KnowledgeCraft, Omega).

All of these environments have several features in common. They provide an interactive, graphical development environment supporting rule-based and object-based styles of programming. Generally they provide some form of automated truth maintenance where the user is able to pursue several contradictory lines of possible reasoning at the same time (called ViewPoints in Art, KeeWorlds in KEE). They usually allow access to the language in which the system has been implemented (now CommonLISP in most cases), but use of that language may be restricted if it contradicts the control strategy of the rule-based system.

These environments have been developed on machines which are expensive LISP-architecture processors. Some of them are being rewritten in C to run on general workstations such as the VAX Workstation and the Sun 3 series, but it remains to be seen whether the development environment available on such machines will match that provided on machines like the Symbolics. However, one advantage conventional machines can offer is better facilities for real-time control, e.g. interrupts.

4.a.4 Shells

There are now so many commercially available shells that it is impractical to list them all. They vary from simple shells such as M1 from Teknowledge, through χ_i from Expertech, to sophisticated shells like Envisage from Systems Designers (Keen 85).

Their shared characteristics are that they are easy to begin to use. Much of what the knowledge engineer will want to do has already been done for him. The shell controls how questions are asked and decides the strategy for using the knowledge which is encoded in rules. Shells provide rudimentary data types (numbers, truth values and perhaps strings in the simpler shells plus Pascal-like records and arrays in the more sophisticated ones).

However shells are not the right type of tool for many AI applications. Their fixed control strategy is inadequate for many complex tasks, and their data structures are too simple for a lot of AI work.

OPS5 is difficult to classify. Its data structures are more general than those found in other shells, but it does not have the ease of use which a shell provides. Its ONLY control structure, forward-chaining, can be so unwieldy for some tasks that the user would be much better off writing directly in LISP.

4.b How our tool was chosen

When the general details of the four categories were matched against the needs of the project, an AI toolkit seemed to meet those needs best.

Languages and language environments did not offer enough packaged facilities or tools to enable rapid prototyping; extra work would have to be done to implement all of the control and knowledge representation features which the application requires. If such facilities can already be purchased in toolkits then this can save a lot of effort and increase programmer productivity.

An expert system shell would have been too restrictive for the necessary knowledge representation and control, even though it might have given the project a quick start at attempting that representation. Flexibility was something that we wished to maintain throughout and we did not want to find that a shell design constraint or operational restriction later emerged with drastic consequences for our project.

It was hoped that a knowledge representation environment would give the best of both worlds—reasonable freedom of representation and control (unlike a shell) without the overhead of developing it from the basic building blocks which a language provides.

In the summer of 1985 only the ART and KEE toolkits were readily available and so these were the only two which we were able to consider in detail.

ART was chosen because it appeared to have more real time facilities than KEE (in the sense that

it could make faster decisions). This is due to the compiled aspect of the knowledge base. It also seemed to have less imposed structure than KEE (which had some fairly rigid hierarchies). A third point was that it contained the Viewpoints mechanism which allowed switching between different contexts (only added to later versions of KEE). KEE, on the other hand, clearly had a superior input/output library with better human interface facilities including icons, windows, better graphics interface and the TellAndAsk language. However, we considered this less important for our application than the other points.

It should be noted that other similar toolkits are now available. KnowledgeCraft is a very interesting system, especially in its use of frames. Although its rules are less well-integrated with the system than those of KEE or ART, its frame representation language is more powerful and expressive.

5 Experiences of using ART

5.a Description of toolkit contents

ART is a complex system and it is an advantage to be familiar with the concepts of frames, inheritance and production rules before attempting to use it. We cannot comment on how ART would appear to someone unfamiliar with these AI concepts. The documentation does have some tutorial material and the training course is very helpful. The Reference manual is large but ill-structured and the section on Viewpoints would benefit from more examples. Viewpoints allow multiple knowledge base states to be maintained simultaneously for hypothetical reasoning or perhaps reasoning about time.

Schema (frame) hierarchies can be quickly established and the facilities for defining slots and relations are not difficult to understand. It would have been useful if there had been a facility to state exactly how many values a slot can or must have or even the range or type of value the slot can hold. This can be done by writing a rule or defining a contradiction but it was felt that this sort of information should be held within the definition of the slot. Also it is not permissible to state which slots should be inherited via an inheritance relation. Once a slot is defined as inheritable then it is generally inherited via all the inheritance relations. Sometimes we wanted to say that a slot could be inherited via some relations but not others. Again it is possible to do this with rules but it would be less complex if this could be done in the relation definition (as it can in KnowledgeCraft).

One feature of ART that we were originally attracted to is that the notation for writing forward chaining and backward chaining rules is very similar. KnowledgeCraft for instance reverts to Prolog for its backward chaining facility and OPS5 for its forward chaining. ART has the advantage that another language does not have to be mastered for backward chaining. However it was disconcerting to have goals generated even when there was no backward chaining rule to satisfy them. The manufacturers of ART admit in the ART course material that coping with unexpected goals "is a major part of the programmer's task". We never had serious problems with this as we did not have many backward chaining rules and it is possible to check the goals generated over the user interface, but some complex applications could find this a major problem.

The user interface proved invaluable for prototyping. The lack of graphics facilities had been a major drawback of the RTL/2 implementation of AFFIRM. As the system grew it became more and more difficult to maintain a clear picture of the state of the knowledge base; particularly when experimenting with inheritance of knowledge between frames. Time had to be spent writing routines in RTL/2 to display the contents of the knowledge base in a user-friendly format. For this reason the graphical representation of the Schema and Viewpoint networks, called lattices in ART, was very useful. The Schema lattice was important for showing inherited features of our EQUIPMENT frames; the Viewpoint lattice displayed our EVENT hierarchy. We like the way practically everything is mouse driven and there is a lot of help available on-line. However, the database query menu option operates on compiled facts and so demands knowledge of how ART compiles its data.

A program may be inspected at run-time with the Watch facility. For example, we watched the ART Agenda, during tests, to ensure our rules were firing in the correct order. It is possible to watch

facts and goals being asserted and retracted which is useful for testing individual rules. It is easy to suspend a program, inspect and change the state of the database and resume execution. Again we needed to write RTL/2 code to provide similar facilities in the original implementation.

Another feature that is very important for our application is the ability to interrupt ART. We intend to have a system that will be able to diagnose and recover from errors in the robot work-cell. Therefore the system must be able to react to error signals quickly. The ART run command has the key-word: async-function which can be used to specify a LISP function to be tested after every rule has fired. In our case the function would poll the sensor communication channels to discover whether an error had occurred. This is much simpler interrupt handling mechanism than many other LISP environments provide and a lot easier to use than the interrupt handler, based on the operating system RT-11, in the RTL/2 version of AFFIRM.

What we were unable to implement in ART was written in LISP. LISP has the advantage over RTL/2 in that functions can be defined and manipulated at run-time; using RTL/2 we had to recompile and relink modules to create new procedural knowledge. Many of the ART functions can be used in LISP code so it is very easy to define new procedural knowledge in the form of an ART rule at run-time. LISP can be used in an ART rule but the interface is non-orthogonal because there are different requirements for using LISP on the left-hand and right-hand sides of a rule. Initially the distinction between an ART sequence and a LISP list was confusing because they are typographically identical but programmatically distinct.

5.b Implementing AFFIRM in ART

The functional specification of AFFIRM may be implemented in ART in a number of ways. AFFIRM uses frames similar to ART schemata; data is inherited through an EVENT frame hierarchy in the same way ART does via inheritance relations; there is a pattern matcher and an agenda. However some of these similarities proved more superficial than actual.

Some aspects of AFFIRM were very easy to translate into ART. We had to experiment with the representation of EVENT frames for the following reason:

The concept of frames in AI allows for "procedural attachment"; the filler for a slot can be a procedure to generate a value, provide a desired side effect or to modify control of the system. The pre and post conditions in EVENT frames held this type of procedural knowledge and resembled an ART rule. The language for specifying the conditions in AFFIRM had been limited whereas the language available in ART rules is extensive. Could we harness this for our purposes? If an EVENT frame became a schema, how could we represent procedural knowledge in a slot? LISP procedures can appear in ART slots but we wanted "rule attachment"; ART rules as fillers for slots. This was not feasible because the rules depend on certain macros that are difficult to bypass. Consequently we experimented with using rules for the representation of EVENT frames.

Translating AFFIRM EVENT frames into ART rules caused two problems. An EVENT frame inherits by default the values for some slots from its parent EVENT frame. This meant that in ART we had to know which rule had caused a rule to fire to know from where these values should be "inherited". There are many ways to do this in ART and we settled for using Viewpoints mainly because we did not want any control patterns in our rules for EVENTs. We generated a simple Viewpoint tree and did not need nested contexts so it was relatively easy to do. It is a good facility to have available; we used it a lot but we did not use the more complex Viewpoint structures that can be generated.

The other problem is the general one of using a production rule representation for an application like ours. Robot programming requires a defined sequence of EVENTs to be performed in a specific order. AFFIRM makes no attempt to evaluate and change the EVENT sequence specified by the programmer. By defining the EVENTs as rules we created the problem of ensuring the rules fired in the correct order. Again we used Viewpoints to do this but there are simpler methods. For example ART rules can be given more or less priority by changing its "salience" from the default value of zero; the rule with the highest priority is fired first.

5.c Final comments on ART

Any system when reimplemented in a different environment will change to suit the facilities inherent in that environment. Besides using rules for EVENT frames we generally resisted altering the AFFIRM specification to suit the ART representation. However, some of the recent advances in our research have evolved from using ART but we feel that ART is not as versatile as other programming environments and expect to write in LISP to tailor ART for our application.

ART has proved to be particularly useful for developing and testing ideas largely because of the excellent user interface it provides. In future it will be necessary for us to use multiple viewpoint levels and a criticism of ART is that this important feature is so badly documented and insufficiently integrated with the frame representation.

In conclusion, ART is a very useful tool and a new ART user who is familiar with AI concepts can quickly build a naive system. We did not use all of the facilities ART has to offer, but now have a clearer understanding of where we can use ART within our research. It is feasible to use ART within a flexible manufacturing system but we doubt whether we will continue with the production rule representation of EVENT frames because of the overheads required to control a sequence of events; a procedural language would be sufficient. However we believe that ART will be used extensively for the more fundamental concept modelling and reasoning behind the diagnosis of errors occurring in the robot workcell.

6 Summary—lessons learnt

Unfortunately it is not easy to read through a manual on AI tools and discover how useful they will be. The necessary experience requires considerable use and experimentation with a system. At present the cost of such systems, in the region of $50,000, prohibits speculative purchase in order to evaluate them but the main problem is that the amount of time it takes to understand these systems is greater than any reasonable time allowance for evaluation and selection of a suitable one. This is because it is essential to try out a tool before one can really gain a good feeling of how it operates. There are relatively few sites with these systems installed and consequently there is very little general consensus on how they compare or on the criteria for selection for different applications. The target applications tend to be complex and highly varied so it is likely that one user's experience of using the provided structures is not directly applicable to another user's application. Although the novelty of AI tools brings selection problems, we can argue that this effect will eventually diminish as user clubs become established and the programmer has access to a human resource of expertise about the efficient use of such systems.

A more serious problem, and one which impedes the exchange of information, is that the notations used to describe the systems are "created" by the manufacturers, so that the features of the systems appear to be different custom-built facilities. The basic AI concepts that are used are referred to differently in different products. Even worse, the techniques used are not made explicit but are sometimes hidden within the customisation of a particular package. The control methods are sometimes mixed in with the superstructure of a package and it is not easy to separate out the basic AI concepts and control issues that are being used from some of the more superficial interface techniques and other facilities provided. Currently the larger packages are undergoing evolution, each manufacturer being aware of the latest offering from his competitors. This means that the versions tend to converge as each new issue from a manufacturer offers facilities that his competitors previously offered as an advantage over him. It is not clear whether this evolutionary convergence will reach a stable state with a range of packages that are all fundamentally the same. The only rigorous way to get over this problem would be for everyone to adopt a standard notation for AI concepts which could be used when analysing two or more systems in a comparative evaluation.

What is needed is a catalogue of AI toolkit features described in an explicit and impartial form. Although there is a general AI catalogue (Bundy), a very precise definition of language facilities and control structures is necessary to compare packages. Such notations exist for hardware systems (Bell), but it is very difficult to produce such a comparative regime for software.

Recently the UK's Alvey Information Technology Initiative has set up the Knowledge Representation Systems Trials Laboratory (KRSTL) at Edinburgh's AI Applications Institute. Now it is possible to visit KRSTL, try out different AI toolkits (and languages) on a range of hardware, and spend some weeks evaluating your application on their systems. This service, which has only just started, shows signs of being valuable in the future.

References

Attardi, G., Corradini, A., Diomedi, S., and Simi, M. "Taxonomic Reasoning" in *Proc ECAI'86*.

Bonner, S. and Shin, K.G. "A Comparative Study of Robot Languages" in *Computer* 15(12): 82–89, 1982.

Bundy, A. (Ed.) "Catalogue of Artificial Tools", Springer Verlag, 1984.

Bell, C.G. "Computer Structures: Readings and Examples", C.G. Bell and A. Newell, McGraw-Hill, 1971.

Keen, M.J.R and Williams, G. "Expert System Shells Come of Age" in *Research and Development in Expert Systems*, Cambridge University Press, 1985.

Levitt, R. and Kunz, J. "Using Knowledge of Construction and Project Management for Automated Schedule Updating", *Project Management Journal*, December 1985.

Lozano-Perez, T. "Task Planning" in *Robot Motion, Planning and Control*, Ed. Brady M. et al., MIT Press, 1982, pp. 473–498.

Minsky, M. "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, Ed. Winston, P. H., McGraw-Hill, 1975, pp. 211–277.

Newell, A., Simon, H.A. "Human Problem Solving", Prentice-Hall, 1972.

Appendix A

RTL/2 was run on an LSI 11/23 with 64 kByte (later 256 kBytes) of memory. The operating system was RT-11 and a slightly enhanced multi-user version TSX which allowed flexible, simple multitasking. Each job was restricted to less than 64 kBytes with overlaying but no virtual memory. ART was run on a Symbolics 3670 with 4 Mbyte of real memory, a virtual memory of 1–2 Mbyte and lots of advanced facilities including a bit-mapped graphics console, mouse, and multiple windows allowing several tasks to be done in parallel.

Appendix B

Glossary

Component—Part of the product being assembled.

Device—An object in the workcell that does not form part of the product being assembled.

Expectation—Usually a sensory expectation, i.e. the value a sensor should have at a particular stage in the assembly.

Frame—A data structure representing a stereotype. Details of the stereotype, which may be adapted, are held in slots in the form of factual or procedural information.

Intention—The purpose of an action performed in the workcell typically described as a relation between components and/or devices.

Plan—An ordered set of actions, expectations or intentions.

Procedural Attachment—Assigning procedural information to a slot in a fame.