Programs that Model Themselves

Iain D. Craig
Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK EU
email: idc@dcs.warwick.ac.uk

January 25, 1998

Abstract

Work on reflection has frequently concentrated on programming languages and operating systems. This work either involves the use of tower reflection or meta-object protocols. The subject of this paper is a different form of reflection that is, we believe, more powerful and fundamental than the other kinds. Self modelling is a process that has not received attention in the literature; indeed, we believe that we are the first to study the topic in a systematic fashion. We will explain the concept of a self-modelling program and describe some example programs that construct and manipulate models of themselves. The programs that we consider are: a learning program, a conceptual modelling system and a production rule interpreter. We also consider the benefits that are to be gained from engaging in self modelling and we show how the programs that we have described benefit from the additional work required of them. We indicate how additional power can be derived by engaging in self modelling.

1 INTRODUCTION

Work on computational reflection has concentrated on three central issues:

- Tower reflection in programming languages [**?**, 24, 25];
- Meta-Object Protocols (MOPS) [17];
- Open implementations [5].

Tower reflection is, historically, the oldest form and was introduced in Smith's 3-LISP [24]. In tower reflection, when a problem is encountered at some level, structures are reified by the level immediately above and processing continues at that level until the problem is resolved. In programming language terms, the problem is typically a matter of performing some operation that requires access to some of the language processor's internal structures (e.g., environment or continuation). There has been work on reflection involving reflective towers as well as varieties that do not (or do not explicitly) use towers (for example, see the work on Brown [25]). Research in tower reflection concentrates, almost invariably, on languages. The SOAR cognitive architecture engages in a similar form of reflection [23].

Meta-Object Protocols (MOP) were introduced in order to allow users to extend the language they are using. Initially, MOPs were introduced for the Common Lisp object system, CLOS [6], in order to allow users to extend the range and behaviour of classes in a way reminiscent of LOOPS [2]; it also allows users to implement different kinds of slots of different search methods to be performed when engaging in multiple inheritance. In order to do this, access had to be granted to aspects of the CLOS implementation. For example, users can directly call the routine which initialises an object's slot; they can also directly access the arguments of a generic function. Users can also override many CLOS mechanisms by means of the introduction of subclasses of CLOS classes, and by changing the meta-class of a given class. A detailed description of the MOP method can be found in [17].

MOPs have been introduced for a number of object-oriented programming languages, including TeΛoΣ, the object-oriented component of Eulisp [21].

The concept of an Open Implementation arose from the work on MOPS. The method for producing MOPs requires constrained access to the internals of the language's processor. In a sense, this makes the underlying implementation open to manipulation by user programs. A new variety of compiler has been developed experimentally at Xerox PARC, and an open variant on C++ has appeared [5]. C++, unlike Lisp, is a compiled language, and the compiler makes the parse tree produced by C++ available to users so that they can manipulate the call and other structures of programs. Meta-classes are defined which introduce new functions by means of this manipulation.

Open implementation was independently used by the author (who termed it the 'tool kit approach') in a series of experiments, beginning in 1983, in which a number of production rule interpreters was developed, each with increasingly powerful reflective capabilities. The sequence ended with the ELEKTRA interpreter [7, 11, 12]. The open implementation technique allows rules to access every significant structure in the underlying interpreter. This allows rules to affect the behaviour of the interpreter in various ways. ELEKTRA is the subject of a later section of this paper because, although it employs open implementation in its construction, it also displays other characteristics which are particular to a new kind of program.

As noted at the outset, the three techniques just discussed dominate research into reflective programs. Much of the work reported in the 1996 US workshop on reflection falls into these three categories. In a sense, research into reflective programs should be re-termed research into reflective *programming*. There is an alternative to the above three approaches that has been less studied, even though it has a longer history. The alternative is *self-modelling*, an approach investigated by Lenat [18, 19], Bakker [1] and Braspenning [3] and by the author in both published and unpublished work [8, 9, 11, 12].

The purpose of this paper is to outline the approach of building programs that model themselves and to indicate where it might be of benefit. The approach, at first sight, appears impossible because a self-modelling program must necessarily be larger than itself and must contain components that are responsible for modelling the model (because the model is part of the program and must be modelled).

As will be seen, this is far from the truth. We will conclude by presenting four examples in varying detail: the general approach adopted by Lenat in his EURISKO [18, 19], Bakker [1] and Braspening [3] in their INCA, our ELEKTRA [11, 12]. In addition, we will present an extra case study in which we examine how a hypertext program can be given a model of itself and how such a program would benefit from this augmentation.

Acknowledgements A paper on "Programs that Model Themselves" has been my intention since 1990. During the intervening period, concepts have become considerably clearer, and the paper on agents [10], helped in this clarification process. The current paper has been written as a direct result of two visits to Maastricht, during which Peter Braspenning and I discussed reflection in great detail. I am more than pleased to acknowledge Peter's influence on this paper. I would also like to thank my wife Margaret for her constant support during dark days and her insistence that one should not be swayed by fashion, particularly in intellectual matters where only the highest standards should be tolerated.

This paper is dedicated to my father on the occasion of his sixty-fifth birthday.

2 WHAT ARE SELF MODELS AND HOW ARE THEY POSSIBLE?

The concept of a 'self' model might seem a little implausible at first sight. Certainly, people are able to construct models of themselves (we have discussed this in [10] at some length), but such models are not particularly detailed, and are, in some sense, different from the subject matter. Here, we have a structure which is supposed to contain a description of itself. Since the model is in the structure, it, too, must be modelled. The model of the model must be modelled, and so on, *ad infinitum.* This appears to be a completely reasonable line of reasoning. It is, however, incorrect; it is quite possible, although rather difficult, to construct programs that model themselves; indeed, we have already constructed some, as have others (e.g., Lenat [18, 19], Braspenning [3] and Bakker [1]). We will describe some programs that contain self models in the next section.

In this section, we will explain what we mean by a self model and we will outline one method by which they can be realised. We must emphasise that, at this stage, we do not have any single, guaranteed method for constructing self models, nor do we know how the process might be completely automated.

By a self model, we mean a structure which contains a description in mutable and inspectable form of the system or program which contains that structure as a proper component. The self model describes the structure of its program and it also describes its content. Every significant component of the containing program is described in the self model. (Unfortunately, the adjective 'significant' must remain an uninterpreted term for the time being, for we have no real handle on how to determine what is significant and what is not; we must rely upon intuition to guide us in determining which components are truly significant.)

We need to distinguish between a static and a dynamic model. The reason being that static models describe the structure and function of programs, while dynamic models describe what happens at runtime. As will be seen, in ELEKTRA, we have a static model that describes the contents of the rules in the program, and we have a dynamic model that is constructed in working memory as the program runs. The dynamic model might record decisions that have been made and rules that have been fired; usually, the dynamic model is maintained as the assertions that have been added by the action of rules. The static model allows the system to reason about how it is organised, what it contains (in terms of data and data types), and what it is intended to do. The dynamic model records how the system is doing this job. Both can be the subject or processing (reasoning).

It is possible to extend the scope of both types of model. The static model might, for example, contain information about the expected outputs of components; this can be used in fault detection and diagnosis (but note that we strongly believe that diagnosis can only be performed when a program has been designed for it—it is not something, we believe, that can simply be added at a whim). It might also be possible to include something about the 'purpose' and design of each component; this could form the basis for the on-line synthesis of new components that can replace faulty ones, or even new components that are more liberal in their inputs than components that are already included. This might be of use, for example, if we wanted a program to accept a different input type from the one it already handles—this is an important and pressing problem in software construction and the source of many unwarranted errors; this is a kind of problem that appears often during system extension and maintenance, and is the cause of much more trouble than it should. The synthesis of new components is a goal rather than a reality at present, unfortunately, but a goal worth striving for, we believe.

We must also discuss the roles of dynamic and static self models because, not only do they differ in their content, they can differ in their role; it should be noted that our view of self-modelling programs is a little unusual. This can be seen from the fact that we believe that a dynamically assembled model of a self-modelling program can be used by the program when it is not engaged in useful work; our view of a self-modelling program is more akin to a complex system or to a form of organism that is required only occasionally to do work, but which can have an existence in addition to this and when it is not engaged in work. Making use of previously created dynamic models is one such off-line use; however, benefiting from dynamic models can also occur when the program is at work and can be implemented in various ways

Now that we have seen that self modelling is possible, we can ask about the benefits that we can expect from self-modelling programs. This is an important question for, it would seem, the effort involved in constructing a program of this kind is only worthwhile if one can derive considerable benefit in return. We believe quite strongly that self modelling gives us a way of improving the quality of programs and of improving their reliability.

Although it is entirely possible that other benefits will be accrued, the following can immediately be discerned.

- Better user help and interfaces.
- Error detection and self repair.
- Self extension.
- Tolerance to unexpected or new data types.
- Continued operation in novel surroundings.
- Learning.
- Self-awareness.

We will briefly consider these examples in order. It should be noted that some of these properties are somewhat more speculative than others.

Below, we will see an example of a self-modelling hypertext program. This (hypothetical) program makes use of its self models to present information dynamically to the user. The program responds to the user's requests for navigation and the user's queries as to page content, relatedness, and so on. This hypertext system is intended to be more flexible than current versions and also to provide more context-specific help than is currently possible. The reasons why this can be done are rooted in the descriptions of the system which the system has access to. This is, in general, an extremely important point: the system has access to descriptions of itself. Clearly, the system's response will only be as good as its self descriptions and the quality of the processes that operate on it. In order to make better interfaces, the interface needs to have access to information about the user and about the functioning of the underlying system; it needs to know what is possible within the system and what is not permitted; when the functionality of the system changes, because of upgrade or maintenance, the interface must also change and self descriptions are of clear use here, too.

By modelling the functioning within the system of its own components, user help can be improved. Again, this depends upon the existence of a self model that is explicitly available to the system. Help can be provided by answering the user's questions and by executing some of the system's functions in a modelling mode. The latter has always been difficult because of the effects of the action on the system and because of the need for live data; in a self-modelling system, the model is a description of the system which is connected to it, but which can be run in a separate space; because the data is also modelled, it becomes easier to access it in this 'what if?' mode.

The existence, within a system, of a model of itself is also important when errors occur. The correct functioning of the system is what is described by its static self models. When something goes wrong, the model will be at variance with what is the case; this discrepancy allows the cause of the error, at least in principle, to be determined. The repair part of this process is, of course, somewhat more complex, for it implies the synthesis of program code at runtime. To know the cause of an error is different from knowing how to make a repair and from knowing the exact nature of the repair to make. However, a self model can be adjusted in such a way that the error does not occur; depending upon the nature of the model, the construction of program code can be a more or less complex task. The existence of self models is again seen to be of considerable benefit here.

If a system can repair itself by constructing new code, it can also, at least in principle, extend itself according to new requirements imposed upon it. As self extension proceeds, the self models the system maintains of itself must be updated. Again, a possible mechanism upon which self extension might be based depends upon self models. What might happen is that the models are updated to reflect the extension and code is generated from them. This approach depends upon the intervention of a programmer or modelling agent who updates the model. In a more futuristic scenario, self extension might result from the system's noticing that a particular function is required but not present. The system might then gather information about the nature of the function and then model it.

We should note at this point that there is, of course, a difference between a model and an implementation. Models are, by their very nature, more abstract than what they model. Moreover, there need be no connection between the form of the model and the form of the thing that is being modelled (differential equations are quite unlike fluid flow; chemical formulae do not resemble molecules). The connection between the model and the modelled is a complex one, and the choice of form and content crucially impacts upon the utility of the model. It is the case, too, that different models serve different purposes more or less adequately; there is no such thing as a model that is universally useful.

Programs often break because someone has tried to extend them using types that are different from those expected when the program was originally built. When messages of the wrong type are sent to modules of a distributed program, errors will occur. Similarly, if the circumstances within which a program operates alter, failure can occur. Self modelling can assist in both of these cases because it makes the expectations built into the program explicit. In the case of a new data or message type, the structure of the new type can be inspected and relationships between it and existing components can be discovered. This might lead to a half-way position in which information can be extracted from the new type. Completely integrating the new type into the program will require the intervention of a programmer or modeller because information not available to the model must be provided. However, if the models can be updated, and if code can be produced from the model, updates are rendered more tractable.

Learning is an area in which we have hard data. The EURISKO program was designed as a learning program. It achieved fame for some of the things it discovered. The real nature of EURISKO was to learn new heuristics, not to discover dramatic concepts; that it achieved fame shows the strength of the approach. EURISKO, as will be seen, contained models of itself. Lenat (*personal communication*, 1982) claimed that any program that is really to learn must contain a model of itself. The role of self models in learning is now established.

Finally, we reach the issue of self awareness. A self-aware program would be able to monitor its environment, detecting and responding to changes; it would also be able to monitor itself, making optimisations where appropriate and incorporating new functions as needed. Self awareness is possibly a pre-requisite for consciousness, but its manifestation in a program remains a conceptual exercise.

3 EXAMPLE SELF-MODELLING PROGRAMS

In this section, we will describe some example programs that model themselves. That such programs have existed for the better part of the last twenty years might come as a surprise to many; it might be a surprise to realise that most relational database systems engage in a little self-modelling via their system catalogues (unfortunately, the catalogue is highly restricted in content and is not used to the full). The examples that we will describe here are:

- EURISKO;
- INCA, and
- ELEKTRA.

In addition, we will consider the case of a hypertext system and how it can be made to model itself. We will also consider the benefits of making this step, both for users and for the system itself. In each of the cases, we will explain (where appropriate) what self-modelling adds to the program and how it improves the functioning of the program.

3.1 EURISKO

In EURISKO, a slot and filler representation is employed. The principal distinction is between units and slots; there is a single name space in which all objects are stored. In both systems, slots are represented by units (an approach that dates back to some of the earliest frame-based systems).

However, in these systems, the amount of information associated with slots is considerably greater than in other systems. A unit describing a slot contains or inherits information about what constitutes an acceptable filler, where the slot may legally appear (in terms of which classes of unit it makes sense for), the definition of the concept represented by the slot, a predicate to test whether the filler of a slot is of the correct type, and so on. The slot is defined in terms of a concept and how it is defined and tested. In addition, a slot is equipped with slots that define how to put values into units, how to add values to complex slots, and how to extract values from units. This cluster of operations allows the definition of complex slots that are composed of many sub-slots.

An important class of slot is that which deals with inheritance. The standard inheritance slots are defined, as are the operations that support the inheritance of values. This makes inheritance explicit, and it also allows users to define new inheritance modes if they so wish. For example, an inheritance mode which skips every second generation can be defined with relative ease, and then installed into the units of a system. In an identical fashion, it is easy to introduce multiple inheritance in different forms by the definition of slots and inheritance functions; it is straightforward to have CLOS [17] inheritance, LOOPS [2] inheritance as well as FLAVORS [4] inheritance all in the same program.

The slots of EURISKO are complete concepts in their own right. The units that are composed of slots are also treated in a similar fashion. As noted above, there is only a single name space, so naming conflicts can occur between slots and units. It is impossible, without resorting to complex name manipulation, to define meta units with the same names as the units they describe (Lenat has argued in a number of places that the distinction between a unit and meta unit is meaningless, and therefore such difficulties should not arise).

In EURISKO, everything is represented by means of units and their component slots. EURISKO operated on a hierarchical agenda of tasks, each of which was implemented as a unit (strictly, an instance of a unit class, but this is never made clear). Task units specified the task that was to be performed (specified the goal, in other words), and also contained information required to order the agenda as a function of the importance of the task. Each task required a collection of slots for its implementation, and they are all to be found in the unit space.

In a similar fashion, the rules that EURISKO used to implement tasks and to perform other forms of reasoning were implemented as units. EURISKO permitted many kinds of rules to be present in the system at any time. The simplest rule consisted of a condition predicate and an action procedure. This scheme can be augmented by the addition of predicates that perform tests that are of greater or lesser complexity and of greater or lesser computational cost; by introducing different kinds of predicate, it is possible to make rules of increasing or decreasing particularity—rules that are satisfied in fewer or more situations. Moreover, the processes of generalisation and specialisation are made easier by the alteration of predicates and actions; a mechanism that depends, basically, upon substitution of the rule's predicate by predicates that occur at higher or lower positions in the inheritance hierarchy for rule predicates (the location depending upon the concepts with which they deal).

In order to execute the various rule types, different rule interpreters are defined. A rule interpreter consists of a unit that selects and executes the various slots in a rule. One simple way to construct a new rule interpreter is to take an existing one and add a slot that interprets a slot newly added to the rule. Other ways are also possible, particularly those dealing with specialisation and generalisation.

The basic operation of EURISKO depends upon the execution of rules. Every rule is represented by a unit, as we have seen, and every rule contains information about its utility (this is an aspect of the meta-information denied independent representation by Lenat). Every time a rule is considered for execution, its meta information is updated by a basic interpreter; when rules fall below the norm for their class, they are subjected to modification processes of various kinds.

An interesting property of EURISCO's interpreters is that they can be constructed, more or less, on demand and to fit the needs of the problem. In fact, matters were probably less dramatic and interpreters were formed as a result of learning that new kinds of rule were required. This can be performed in a relatively simple way by having tasks that notice when new rules are created with additional slots. The slots are *active* concepts (as opposed to *passive* concepts like numbers or sets) and have little interpretative methods associated with them; the construction of a rule interpreter consists of the modification of an existing one so that the newly added slot is evaluated in the appropriate place. The clever part is spotting that the slot has been added.

The representation of objects in EURISKO in terms of slots makes modification relatively easy. It is necessary either to replace the information associated with a slot and make appropriate changes to every appearance (instance) of that slot in the units of the system, or to define a sub-class (sub-unit, but interpreted in terms of inheritance) of the slot and make appropriate changes. Both of these operations involve little more than an edit and an iteration. Small changes can have large effects in a system such as this.

The self-modelling aspects of EURISKO are relatively easy to see, yet whose implications are relatively hard to determine. Each unit is a collection of slots, and each slot is a complete unit. Associated with each slot is a considerable amount of descriptive information, as we have seen. The collection of slots that comprise a unit provide basic information about that unit; this principle applies to *every* unit in the system. In addition, each unit can have descriptive information associated with it, as we have seen in the case of rules. Here, the number of times a rule has been matched, the number of times it was found to fail, the number of times it was found to be appropriate and the number of times that it was chosen, all contribute to a rough estimations of the rule's worth or appropriateness. Whenever a rule is selected either for testing or execution, the super-ordinate concept, the rule concept, is updated, so totals are available. (It would be necessary to try to represent more information about the system state at the time of matching, consideration or execution before an arguably better metric could be provided.) Whenever a change is made to part of the system, that change is reflected in various ways on the rest. For example, the following cases are clear:

- When a new unit is defined, its existence is recorded in a slot of a very general unit (the *AllExamples* slot of the unit *Concept*);

- When a new slot is defined, it is added to the *AllExamples* slot of the *Slot* unit.

- When a slot is added to a unit, a new unit should be created, thus specialising the old one. Both parent and offspring units remain in memory.
- When a slot is removed from a unit, its relationship to its parent(s) and siblings should be reviewed.
- When a new slot is defined, all instances of its parent should be examined to see whether the new slot is more appropriate,
- etc.

Note that active objects (functions) can also be defined and added, so the behaviour of the system is altered by operations similar to those in the above list. In EURISKO, provision was made for units to contain mechanisms for the modification of Lisp code, so new functions could be constructed on the fly. In order to do this, a type system had to be defined. The simple syntax of Lisp (which is interesting because the concrete syntax *is* the abstract syntax) greatly assists this, although it should be possible in languages with more complex syntax provided the interpreter and compiler are available to be called at runtime.

There is an extremely close connection between self-model and program functioning in EURISKO. One reason for this is that EURISKO was a learning (actually, a discovery) program (Lenat, *personal communication*, has argued that learning programs *cannot* be constructed without self models). Another reason is that the representation chosen by Lenat closely couples description and program.

4 INCA

INCA [1, 3] is an object-oriented tool intended for the modelling of a variety of systems. INCA has a strong basis in Artificial Intelligence, a property shared by most of the examples in this paper. INCA is, in its structure, similar to EURISKO's representation, in that it is object-oriented; however, it is considerably more principled in its approach.

The core of INCA is its ontology [1]. Unlike other object-oriented proposals, INCA takes the concept of *ontology* seriously and uses it to classify the principal concepts in the INCA system. The major distinctions are between sorts and objects (called *primaries*). Objects and sorts can both be instantiated; the instantiation of a sort is an object, while the instantiation of an object is an instance. INCA also permits meta-sorts and objects as concepts with equal status; indeed, it requires the description of objects in terms of meta structures of various forms. INCA goes considerably further, however, in characterising all of its concepts in terms of name spaces that contain inheritance graphs. This allows for the classification of all objects in a principled way, and in a name space that is appropriate. Sorts, therefore, are collected within their own name space, as are meta-sorts.

The use of inheritance as a classificatory mechanism is of considerable subtlety. It is necessary to define sorts as objects, and this is done by making the

object concept the root of an inheritance graph with **sort** as one of its child objects. This effectively classifies instances of **sort** as objects in their own right, and it does so in an explicit fashion. In a similar fashion, objects that possess attributes (which are further classified, see below), are collected in a name space that holds objects with descriptions. Inheritance is also reified as an object and is collected in an inheritance graph that relates it to **object** (inheritance is an immediate descendent of **object**).

These and other similar connections make INCA a subject matter in its own right, and a subject matter for itself. This clearly justifies the emphasis on the system's ontology.

Unlike other systems of this kind, INCA treats slots and attributes rather differently. Indeed, it divides slots into a number of kinds, each of which is described by objects in their own name spaces. Slots are strongly typed and are divided into a number of kinds:

- Properties
- Attribues
- Links, and
- Relations

Each of the above kinds is distinct from the others. In particular, relations and links are introduced in order to represent what one might term essential and contingent connections between objects. By this, we mean that relations are intended to denote connections that are of some significance, perhaps in the definition of the object, while links represent connections that just happen to be made.

Inheritance can be applied to any slot kind. However, INCA restricts inheritance to single inheritance. This is for a number of reasons, the most important of which is that it is hard, ontologically speaking, to give multiple inheritance a convincing interpretation. Instead, INCA allows users to view objects in different ways by providing perspectives. By allowing an object to have multiple perspectives (implemented in terms of membership of more than one inheritance graph), the benefits of multiple inheritance can be had for less cost. The participation of an object in multiple inheritance graphs is an example of the need for an object to be named in more than one way (see below).

In [1], Bakker defines a number of different inheritance mechanisms, each of which can be applied to any object. These different forms of inheritance concern the kinds of information that can be derived from a super class. Inheritance takes place within a name space. Name spaces are a mechanism of considerable importance to INCA; they also have a slightly strange behaviour. Every object resides in a name space; objects can be named in different ways, but each name refers to a unique object. When an object is created, it is assigned a system-defined name; users may provide their own names for objects, but this kinds of name is not required by INCA. A distinction is made between the empty object (or *null* object) and the error object. The empty object is uniquely defined for each name space and denotes the object about which nothing whatsoever is known; when describing slots, the empty object refers to the slot about which we have no information. The error object, which is, again, unique in each name space, denotes an object whose use immediately causes an error. When a name space is created, it is immediately equipped with the empty and error objects; all name spaces are *inhabited*.

Name spaces are used to separate inheritance graphs. In other words, each name space contains a collection of related objects and these objects do not interfere with any others. References between name spaces are essential for the functioning of the system. As noted above, different names can be associated with the same object, so different relationships exist between name spaces. Different names arise because of the naming convention adopted for INCA objects. It is based upon the path from the root of the inheritance graph in which the object is defined to the object. Thus, an object which inherits from the classes **pc**, **personal-computer**, **digital-device**, **electronic-device**, **physical-object**, **sort**, will have the name:

sort.physical-object.electronic-device.digital-device.personal-computer.pc

The name reflects the path from the object to the root of its inheritance graph. Because an object can be viewed in many ways, the names with which it is associated can vary. For example, all sorts are instances of the meta sort, sort. Thus, any sort also has a name that reflects this fact. Similarly, individual sorts are primaries in their own right because they can be instantiated, and they have a name that reflects this property, too. The way in which an object is viewed determines the way in which references to it are made. The name space in which one is currently operating determines (in part) how one views an object, both in the current name space and in related ones.

Not all name spaces contain objects with attributes, however; this is an important distinction. Attributes belong to objects that are specifically defined to have them. Attribute sets are objects in their own right (another aspect of self-description) and are described within the INCA ontology. The particular structures that are used in the implementation of attribute sets are outside of the system (in a default name space that holds implementation-dependent objects which are not named inside INCA). The structure of each kind of attribute (slot) is described by an object in a name space that defines part of the description (so a slot can be an object and an object component, as well as a sort).

INCA contains its own description, as does EURISKO. However, the INCA self description is much more conceptual in orientation and much more developed. The INCA self description is of greater subtlety than that employed in the other systems. The principles for self description in INCA are related to those in Lenat's systems, but is much more refined. In particular, the principle of reification is consistently applied, as is the principle of separating implementation from organisation and semantics. In Lenat's systems, there is a considerable amount of what might be described as implementation detail, while in INCA, the emphasis is on correct placement in inheritance hierarchies; for INCA, the location of an object in space and in an inheritance graph within a space is the object's defining characteristic.

Furthermore, the perspective from which INCA structures are considered is different from that in Lenat's systems. The INCA perspective is that of defining an object and relating it to others, while Lenat emphasises the programming aspects of each object (how to check fillers, how to perform inheritance, etc.)

4.1 ELEKTRA

ELEKTRA is a production rule interpreter that has undergone a number of implementations. Initially written in 1983 in Prolog, it has been subsequently re-implemented in Scheme and Lisp, and a version in Java is under construction. The definitive description of the Lisp implementations ELEKTRA is to be found in [11, 12]; a formal specification in Z is to be found in [7].

ELEKTRA is outwardly a relatively conventional forward-chaining production rule interpreter and can be used as exactly that. The true power of the system derives from the fact that it supports a rich variety of behaviours, in particular behaviours implemented in terms of *meta rules*. The meta rules in ELEKTRA massively extend those proposed by Davis [14] and, we believe, this system represents the most powerful and systematic application of meta rules to date. The ELEKTRA system allows condition elements to be composed of executable code which can bind variables as a by product of operating on data. The system extends the range of objects available to rules for matching and operation and it also contains a pre-processor for handling rules. The former property is used in conjunction with an open implementation (which the program has had since 1983 when we referred to the property as 'implementation in terms of a tool kit'); open implementation allows rules to access parts of the rule interpreter that normally remain hidden from use.

The reason why an open implementation was adopted from the start was that we wanted to allow meta rules to operate in the system and part of the role we saw for meta rules was that of interpreting other rules. In order to do this, we initially permitted rules to have access to the condition-matching and action-executing components of the underlying interpreter; fairly soon, we saw that it was necessary to permit access to the conflict set and to the rule instances that it contained. We also augmented the matching and execution repertoire, allowing single conditions to be matched and single actions to be executed in various contexts, including those derived from currently instantiated rules. These features allowed us to write meta rules that could interpret object rules, and also to test conditions from rules in order to determine whether a rule was likely to be of relevance to the current partial solution.

From the very start of the work on ELEKTRA, our goal was to construct a system that could execute rules that would act as rule interpreters; that this was possible was guaranteed by the fact that forward-chaining production rules are just a variation on Post systems, and are, universal computing machines is computable by Turing Machine can be computed by a Post system, so any such function can also be computed by a forward-chaining production rule system. We were able to show, and have documented in [12], how rules can be used to implement a forward-chaining rule interpreter (this requires two rules). In that same paper, we also showed how to implement a backchaining interpreter as a set of forward rules; the backchaining interpreter as published is a little restricted because there are details of syntax that we did not care to handle in the paper. The paper also shows how a number of so-called 'content-directed' reasoning systems which motivated Davis [13, 14] to introduce meta rules in the first place. To enable the swift operation of these various interpretational strategies, the pre-processor is of central importance.

The pre-processor is used to convert rules into assertions. Initially, the reason for this was that the first version of ELEKTRA actually destructured rules in production memory when it needed to reason about content. This turned out to be an expensive operation which restricted the system's extensibility (if, for example, rules were to include an additional condition, say an *unless* part, the routines which accessed rule structure would need to be rewritten). Furthermore, access to rule components required the rule to be retrieved from production memory and the component accessed; this took a considerable time. It was decided that, apart from allowing direct read/write access to production memory, rules in the later versions of ELEKTRA should be converted into a form which allowed them to be matched in exactly the same way that working memory elements are. This would increase the time required to access components while maintaining the ability of rules to access other rules as intact structures.

It is the pre-processor's task to take rules and to generate working memory elements from them. The working memory elements that are produced describe the rule in various ways. The length of the condition part is recorded in terms of the number of elements it contains, as is the length of the action part. For each condition element, its predicate is recorded as being used or mentioned (depending whether it appears in a condition element or in an action) by the particular rule; constant and function symbols—the representation is that of atomic formulæwith Skolem functions—are treated similarly. Actions are also subjected to the same analysis; it is possible to query working memory, for example, to find the names of all those rules that execute the working memory addition action. The order in which various elements appear is also recorded so that the following queries, for example, can be answered by working memory:

- What is the nth element of rule x's condition part?
- What is the nth element of rule x's action part?
- Does constant c appear in the kth argument position of relation r in any rule?
- Which rules mention relation r?

Additional operators allow the computation of such things as set cardinality, unions, intersections and differences; operations on lists and bags are similarly provided. Set, bag and list operations are polymorphic and can be applied to various things, for example variable binding sets (they are reified and made available to rules) and the conflict set.

The pre-processor is applied to every rule as it is added to the system (it need only be done once and the resulting assertions can be stored in a file for later use); thus, meta rules are also subjected to the analysis. The availability of information about all rules in the system allows us to write rules that *exactly* describe the functioning of the forward-chaining interpreter itself (this is the main result in) [12]), so we can have a rule system that interprets *itself*, a result we find to be of some considerable significance. (The two rules described in [12] are capable of interpreting the other rules in the system and themselves; they take turns in interpreting themselves and the other rules.)

The working memory elements output by the pre-processor describe the structure and content of each rule; quite clearly, we have a form of self model in this collection of working memory elements. This is a *static* description of the organisation of the primary components of a rule. Rules are divided into meta levels by means of an integer-valued tag, a fact which allows other rules to take the level of metaness into account when performing various operations. The description of the rules in a program allows considerable amounts of reasoning about what the program contains; for a more dynamic account, the fact that the system contains meta rules must be employed.

The ELEKTRA system is, as was noted above, intended to be used with meta rules. The meta rules are to be used in a number of ways, for example:

- Implementing control structures;
- Interpreting rules;
- Generating inheritance hierarchies;
- Defining terms.

Of course, meta rules have full access to working memory; they become candidates for firing because of the presence or absence of particular (equivalence classes of) elements in working memory (this is true even when they are interpreted by other rules). Meta rules have access to conflict sets and other control structures, either those provided by default by the ELEKTRA interpreter implementation or by the structures defined for use by control rules. By means of the addition and deletion of working memory elements, rules (meta rules) can keep track of what is happening in the system as it runs. In other words, by appropriately recording in working memory descriptions of what is being done.

In the Blackboard Control Architecture [15], control decisions are in principle recorded on the architecture's control blackboard. The qualification 'in principle' is employed because, for real applications, the amount of store required is excessive and system performance degrades as a consequence; it should not be impossible to construct a mechanism for such recording which does not have such a negative impact. In the ELEKTRA case, decisions can be represented by much smaller objects than required by blackboard systems, so the demands imposed by such recording is lower.

Whatever the mechanism, it is possible to record decisions in working memory. These decisions relate to rules and to variable bindings, as well as to the context within which they were made. For example, a simple alteration to the basic matching routines allows them to record the working memory elements that have been matched (it is also possible to assign each working memory element a unique identifier so that they can be referenced in subsequent processing). Such additions make possible the recording of more of the context within which a rule was matched and within which the decision to fire it was made.

The fact that a working memory has been removed requires an additional recording by means of an addition to working memory, but this can be performed with relative ease (it requires another trivial modification to the system to make working memory elements completely available to rules—at present, only the assertion that they contain is accessible). By this means, all the information that is required to characterise a control decision can be made available to meta rules. (An addition to the system's fundamental structure that might be considered is that of a truth maintenance system of some form; this would allow the recording of context on an automatic basis, but its impact on the ability of the system to reason about itself remains to be investigated.)

We can conclude this discussion by stating that dynamic models of the program as it is in operation and generated by the system in a form that can be manipulated and reasoned about by rules is not impossible in the slightest. Precisely what should be included in such models (and we should not rule out the possibility of maintaining several models) is a subject requiring considerably more work. Nevertheless, we have, we hope, shown that it is *at least in principle possible* to engage in dynamic modelling. The next question is: what use is such modelling? We will address this when we have a more pacific view of the modelling process.

We have seen that a reflective production rule system has a self model and uses that self model in its normal processing. The model consists of a set of assertions held in working memory that describe the contents of the rules that constitute the program. These assertions can be augmented by other assertions that describe such things as inheritance hierarchies, class membership, applicable operations, and so on. This model is then used by rules at various levels in performing many tasks, the most dramatic of which is self interpretation. We have also seen how the production rule architecture allows us to record control and context information as it is engaged in processing. This dynamic modelling can be used for a variety of purposes, the most immediately clear of which is performance improvement. Below, we will consider other ways in which such dynamic self models can be exploited. Before moving on, it is important to note that the dynamic models that we have discussed in this sub-section cannot be constructed without the mechanisms required to implement the static models produced by the pre-processor and the attendant classificatory structure; nor, equally, can the dynamic models be interpreted by rules without such supplementary information. The two aspects of the modelling process are inseparable.

Dynamic modelling is an aspect of ELEKTRA that we have not yet discussed in public although it has been implicit in much of what we have written and thought over the years. Capturing information about a program in a form that can be put to good effect by that same program as it continues to run is essential to the project of constructing self-modelling programs. So far, only ELEKTRA has been shown to be capable of exploiting static and dynamic models to a great extent (the Blackboard Control Architecture can do it in principle, but lacks the specification of inference mechanisms and cannot support the kinds of distinction between object and meta levels that we require); EURISKO used representations that were restricted to the conduct of individual rules and to the overall conduct of all rules in the system, an approach that appears unable to make the kinds of fine-grained distinctions that we require.

In the next sub-section, we will consider how a hypertext system could be implemented that contains and manipulates its own static self model and that constructs and maintains a dynamic model of itself.

5 A Hypothetical Hypertext System

The purpose of this sub-section is to show that self-modelling can be adopted in other systems and that it brings benefits that actually matter to users. The fact that a system such as the one described below has not yet been constructed implies that we cannot know of *all* the benefits that reflection will actually bring; we can, however, be confident of some. We tackle the introduction of a self model into the hypertext system in stages. First, we will consider what a basic hypertext system is. Next, we will extend it in a number of ways so that a model is constructed. Next, we will show that what we have derived is, in fact, a static self model; we will describe what it can do in *a priori* terms. Then, we will turn our attention to the dynamic model that should accompany the static one; we will determine some of its properties and benefits also.

A hypertext system is composed of the following components:

- Pages, which can be text, graphics, images or Multi-media;
- Links between pages (see below);
- A history stack to record navigation.

The basic idea is that pages contain information to be communicated to the reader, links connect pages, and the history stack records which pages the user has visited in a session and can revisit by selecting them. Notice immediately that the history stack, as interpreted, for example, in most World-Wide Web browsers and existing hypertext systems records only the *page* that was visited and not the link which took one there; by recording the link that was traversed, it is conceivable that valuable information will be retained for use by the system. The reason that links are not often stored is that most pages are related only by one link, so the link's identity, should it ever be needed, can be inferred by examination of the source page.

If we were to model this account of a hypertext system, we would have little to bother with. Each page might be represented by an assertion and each link might be represented as an assertion containing two page references. If we so cared, we might introduce type links between pages and assertions about their type; it will probably be the case in any real hypertext system that the vast majority of the pages are multi-media pages, so we might assume this as a default. This is clearly a highly static model; all the dynamism is contained in the history stack which merely contains visited page names.

We need to look carefully at the links that are present. We might divide them into three categories:

- Navigational links: they take use from one place to another. These links include the *up*, *next*, *home* buttons with which we are all familiar. The *quit* button would also be included in this category.
- Organisational links which define the relationship between a page and the encompassing context within which it occurs. For example, a document might be divided into sections, each of which has an introduction, a body, and a set of pages. An organisational link will relate any particular page to one of these structural components. We might also include such matters as *zoom* and *pan* links; the former zoom in on some aspect of the current page, and pan does the opposite by panning to a more general context. We could also include links to indices and glossaries.
- Buttons, or procedural links. These links are intended to do things (e.g., cause a bit of animation to be displayed, some sound to be produced).

The existence of these link types makes the representation a bit richer, but they are still not very exciting! We can type the links emanating from a page and, using the organisational links, we can start to represent the structure of the overall document. This information might be use to an interactive browser to display the text structure. We have, though, enriched the model of our hypertext documents.

We can now extend our network in three ways:

1. Add attributes to pages.
2. Add attributes to links.
3. Add extra 'objects' into the system.

The intention is to introduce opportunities for adding information that is not already present in links and pages. For example, a page can have an author, a revision number, a last update date, a history of pages that it supersedes; it might also have a role to play in the overall document structure which is recorded in an attribute. A page might also refer to a summary of itself or to a collection of annotations for use by authors/maintainers. In a similar fashion, links are introduced by someone, the connection being made on a particular day, and as a consequence of some observation of structure or semantic significance. The role of the additional objects will become clear below; meanwhile, let us be content with observing that pages of all kinds, as well as links, can be regarded as objects of a particular kind—indeed, we might even want to regard them as being active in some fashion.

With the introduction of attributes, we can record information about the *content* of pages. This might be expressed in terms of objects and relationships. The information about page content can refer to what is entirely contained with in a page: a representation of what is on the page (and recall that pictures and animated graphics might be there also, so there is, in principle, a relatively difficult representational problem). There is also the sense of representing the way in which the text relates to the external context, that which is outside or off the page; this might be represented, taking the form of inferential links to other pages.

Links can now be further subdivided, because we have their role as representation-bearing items to take into account. We might, for example, want to assign links the following new categories:

- Relational links. This class of link relates something in the conceptual content of a page to something in the representation of another page.

- Inferential links. One page might contain something that implies or entails the content of another page.

(Links of these kinds need not be stored in the page itself, but in an object that is related to it.)

If we permit ourselves to allow navigation using all link types, we can move from one page to another because they are in some relationship. Buttons can now express relationships between pages as well as being there to make things happen. The history stack can now be augmented so that it records the relationships that have been exploited in traversing the network, as well as the purely navigational links.

The next addition is that of rules. We can add rules to make inferential relations more active. Moreover, because rules contain variables, it is possible to use a single rule to cover many ground instances of conditions. Rules can be used to state navigational relationships, for example:

if the user wants to go back, go to X

This opens up new perspectives for navigation even without the collections of relations that we will discuss in a moment or two.

First, we must change the interpretation of buttons so that, instead of activating links, they cause assertions to be made. When assertions are made, they can be detected by rules. Rules can, of course, examine much more than the assertion that was made by pushing a button; in particular, they can inspect the history stack and any other information that is to hand. This has the immediate consequence that we can separate buttons from any *a priori* given meaning; we can, indeed, make the interpretation of buttons relative to a context (this would require re-labelling of the button when a new context has been inferred and if the button's meaning is to change), the context being, in part, determined by the contents of the user's history stack. Thus, when a button is pressed, rules can be fired to interpret it in the current context. This permits us, *inter alia*, to change what it means to go to the next page or to go back to the start (we might want to do this in, for example, tutorial text, or when a start node has been specially set up for the particular reader). We can record all the relevant information and thus make it available to the inference system, so interpretation can still be made in a reliable (and, let us hope, sound) fashion.

Rules can be applied to reason about the contents of pages so that, for example, a set of next pages can be proposed and even predicted. Contextual information provided by an augmented history stack provides part of the input to this reasoning. Rules are objects and have a content, so we can represent them in the system's memory, as well. Thus, at a particular page, we might need to determine which rules are most relevant to it; this can be done in a manner not too dissimilar from that performed by ELEKTRA. (The reader should note that we consider rules to be as much part of the self model as are the assertions that represent its data.)

What we are doing is to produce an underlying subject matter for the model that the system maintains of itself. The model is one description we are giving in addition to the display form. The additional structure can be used to provide the following, and in a dynamic fashion:

- Provide views (some links might be forbidden, or might be dynamically replaced);
- Provide dynamic navigation (and this might be coupled to additional criteria such as pedagogical ones).
- Constructing browsers.
- Providing context-sensitive help.
- Answering queries of many kinds. For example:
  - Which pages mention relation r?
  - Which pages mention an internal relation r?
  - Which pages mention a relation r as a link?
  - Which page mention concept k?
  - Which pages mention all of the concepts $\{k_1, \ldots, k_n\}$ (or alternatively, which pages mention at least one of the concepts $\{k_1, \ldots, k_n\}$?)
  - Which pages are related to page P?
  - Which rules are applicable to page P?
  - Which concepts are mentioned on page P?
  - Is concept k mentioned on page P?
  - Which pages are implied by page P?
  - Which pages are implied by the current page?
  - How do I go from P_1 to page P_2?

In addition to these queries, the system is now able to engage in behaviours that previously would have required considerable coding effort. For example, it is now possible to have multiple versions of a page in the system at the same time, the version actually seen by the user depending upon the attributes of the versions. Rules can be used to determine which version to use. Furthermore, it is also possible now to engage in a kind of self-repair behaviour. If a page is withdrawn from the system (say for editing or revision), the normal hypertext model will simply raise an error and terminate (how many times has the reader had a 404 reply from a Web page request!) If the reflective proposal is adopted, the fact that a page has been withdrawn is recorded in the system and alternatives can be proposed; moreover, this behaviour is a natural consequence of the content of the system because, should the withdrawn page have been selected on the basis of its content, it is possible to arrange for there to be rules that will suggest another, closely related page, should one exist; it is conceivable that a best-first search be performed when looking for pages, so there will be alternatives available to any request (for very small hypertext systems, of course, there will never be enough pages to do this, but we are considering systems containing hundreds or even thousands of pages). It is also possible for an absent page to lead to a request for information from another site, be it a similar hypertext site or a Web site. We can introduce a distributed element into the hypertext system and do it transparently, just as we can introduce an index or a thesaurus.

Furthermore, the introduction of new pages into the system should be somewhat easier than for current (non-reflective) hypertext systems. The reasons for this derive from the existence of the self model and the queries which it permits. The writer of a new page can determine from the system which pages are similar to the one being introduced and can determine which rules might lead the user to reach the new page. The page needs to be annotated and added to the type hierarchy, its content needs to be encoded, but the process of addition itself can be eased by the existence of the internal model. In order to maximise the utility of the system, we have to represent much more than we initially started with. In particular, we have to represent the buttons that are used to move from one place to another.

Buttons are here to be construed in all their forms, from the up, back and quit buttons, to those which present the animation of some items, to those buttons which move based upon semantic criteria. The fact that a button has been pressed is recorded in the user's history stack for subsequent processing. Reflection adds considerably to the power of the hypertext system, permitting new and dynamic kinds of navigation. A dynamic model is already constructed by hypertext systems in the form of the history stack which records the pages of the pages the user has consulted. In an extended form of system, the history stack must record more than this, for it must also contain references to the aspect of the content of pages that are most relevant to the traversal; in the simplest case, the button pressed in order to go to the next page must be recorded and, because buttons are objects that are represented like everything else in the system, information about the button can be accessed when processing focuses upon the button press.

It might be argued that the representation we have proposed is additional to the representation of the hypertext. We argue that there is no distinction because we are describing a system in which text and other presentation items are just types of constant symbol within a more general and richer representation.

We are arguing that we can extract the conceptual content and represent (encode?) it in a form that is amenable to the reasoning we described above. We think of this in terms of the ELEKTRA model for the reason that it is the most familiar to us, and we can perform all of the above tasks in an ELEKTRA-based system. By adopting an approach based on the principles expounded in the sub-section on ELEKTRA, we bring all of the power of ELEKTRA to bear upon the hypertext system; ELEKTRA becomes a framework within which to represent hypertext concepts and to engage in reflective reasoning.

Above, we have emphasised the use of the self model. Even if we do not use an underlying system as powerful as ELEKTRA, we still have a system that contains its own model because the system has a representation of its own content and structures (pages, links, buttons, history stack—the latter representing itself, perhaps) which, as argued above, it can consult and modify itself. The consultation and modification of the self model is driven by user interaction and the relationship between the self model and the system is, perhaps, rather hard to discern at first.

It is inevitable that one should ask what are the difference between the above proposal and proposals such as Rada's *Expertext* [22]. Rada conceives of expertext as a way of enriching hypertext and making navigation context sensitive. Here, we have context-sensitive navigation, but we can also make predictions about what the user will do in the future, and we can dynamically reconfigure the system. For, example, the user might ask for all pages dealing with a particular topic, or all pages written by a particular person. By means of inferences made about the representation, e.g. inferences about transitivity of relations, about equivalence of concepts, the range of pages available to the user can be altered. In our proposal, the system knows when something is missing or inadequate because its representation is lacking or deficient; missing pages can be detected with relative ease. In our proposal, we can take the models, both static and dynamic, and make inferences using their contents, and we can operate on them in various ways. This ability is missing from the expertext proposal, and from all proposals of its kind (including those wanting to employ 'intelligent' hypertext as a component in an 'intelligent' database system). An important property of self models is that they form a context within and *about* which to engage in reasoning and other kinds of processing.

6 CONCLUSIONS

In this paper, we have analysed the concept of a self-modelling program. A self-modelling program is one that maintains and manipulates a model of its static organisation and content and the dynamic organisation and derivation of results. We have indicated ways in which self-modelling programs can be an improvement over conventional ones. We then examined three existing self-modelling programs and presented an account of a hypertext system that makes and uses models of itself. We showed areas in which a self-modelling hypertext system would be an improvement over other proposals.

We believe that self modelling is a powerful technique and that it provides important insights into the process of reflection. We also believe that it sheds light on the fundamental processes of reflection, doing so in a more balanced and natural light than tower reflection, and with greater analytic depth than meta-object protocols. The exploitation of the techniques of self modelling will also, we believe, lead not only to more robust and 'intelligent' programs, but also to new ways of construing the programming process and the role of programs.

Operating systems such as Microsoft's Windows95 and many of the programs that run under it show a new perspective on programming and programs. Programs, for Windows95, are document-handling objects that are integrated to a considerable extent with the environment provided by the operating system. A C++ program running under Windows95 can access many tools provided by the system, and can make use of many classes that form the *Foundation Class* library employed by the system itself. The construction of programs in this environment is now something that is considerably more automated than many professional programmers in other environments might think; it consists more of pressing buttons and gluing components together. This is a very different process from the one expected under Unix and in which everything must be constructed from the bottom up; instead, there is a process of assembly which relies more upon knowing what services are provided than how to implement them. The relationship between this kind of programming and self modelling is clear; models of components, subsystems, and so on, can be constructed and can direct the programmer or application builder in the assembly of components. Under this view, the nature of programming languages is something that must be called into question.

We can expect, we believe, new attitudes, techniques and approaches to develop from a systematic enquiry into self modelling that would otherwise never have been conceived.

References

[1] Bakker, H., *Object-oriented Modelling of Information Systems*, PhD thesis, Department of Computer Science, Rijksuniversiteit Limburg, Maastricht, 1995.

[2] Bobrow, D. and Stefik, M., *The LOOPS Manual (Draft Edition)*, Xerox Parc, Palo Alto, CA, 1983.

[3] Braspenning, P. J., A Reflexive Representation System for Objects, *Proc. 1990 Summer Computer Simulation Conference*, (eds. Svrcek, B. and McRae, J.), The Society for Computer Simulation, San Diego, CA, pp. 854-859, 1990.

[4] Cannon, H. I., *Flavors, A Non-hierarchical Approach to Object-Oriented Programming*, 1982.

[5] Chiba, Shigeru, A metaobject protocol for C++, *Proc. OOPSLA*, pp. 285-299, 1995.

[6] Steele, G. L., (ed.), *Common Lisp: The Language*, Digital Press, 1990.

[7] Craig, I. D., *The Formal Specification of ELEKTRA*, Research Report No. 261, Department of Computer Science, University of Warwick, 1994.

[8] Craig, I. D., *SeRPenS A Production Rule Interpreter*, Research Report No. 97, Department of Computer Science, University of Warwick, 1987.

[9] Craig, I. D., *The BB-SR System*, Research Report No. 94, Department of Computer Science, University of Warwick, 1987.

[10] Craig, I. D., Agents that Model Themselves, *Perspectives on Cognitive Science*, Vol. 1, New Bulgarian University, Sofia, Bulgaria, 1994.

[11] Craig, I. D., A Reflective Production System, *Kybernetes*, Vol. 23, No. 3, pp. 20-35, 1994.

[12] Craig, I. D., Rule Interpreters in ELEKTRA, *Kybernetes*, Vol. 24, No. 3, pp. 41-53, 1995.

[13] Davis, R, *Teiresias: Applications of Meta-Level Knowledge*, Ph. D. dissertation, Department of Computer Science, Stanford University, also appears in Lenat, D. B. and Davis, R., *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, 1982.

[14] Davis, R, Meta-rules: Reasoning about control, *Artificial Intelligence Journal*, Vol. 15, 1980.

[15] Hayes-Roth, B., A Blackboard Architecture for Control, *Artificial Intelligence Journal*, Vol. 26, 1985.

[16] Keene, S. E., *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

[17] Kiczales, G., des Riviéres, J. and Bobrow, D. G., *The Art of the Metaobject Protocol*, MIT Press, 1992.

[18] Lenat, D. B., The Nature of Heuristics II, *Artificial Intelligence Journal*, Vol. 12, 1983.

[19] Lenat, D. B., The Nature of Heuristics III, *Artificial Intelligence Journal*, Vol. 12, 1983.

[20] Maes, P. and Nardi, D., (eds.), *Meta-level Architectures and Reflection*, North-Holland, 1988.

[21] Padgett, J. A. (ed.), *Eulisp V0.99*, School of Mathematics, University of Bath, 1995.

[22] Rada, R., *Hypertext: From Text to Expertext*, McGraw-Hill, London, 1991.

[23] Rosenbloom, P., Laird, J. and Newell, A., *Meta-levels in SOAR*, in [20], pp. 227- 240, 1988.

[24] Smith, Brian C., *Reflection and Semantics in a Procedural Language*, Ph.D. Dissertation, Report no. MIT/LCS/TR-272, Laboratory for Computer Science, MIT, 1982.

[25] Wand, M. and Friedman, D. P., The Mystery of the Tower Revisited: a non-Reflective Description of the Reflective Tower, *Lisp and Symbolic Computation*, Vol. 1, no. 1, pp. 11-38, 1988.