Chapter 1

A classification of meta-level architectures

Frank van Harmelen

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
frankh@uk.ac.ed.aiva@nss.cs.ucl.ac.uk

Abstract

The goal of this paper is to categorise the meta-level systems in the literature on the basis of their crucial architectural features, concentrating on the relation between meta-level and object-level interpreter. We will discuss the communication between these two components and we will distinguish a number of typical architectures. After that we will discuss a number of other, independent properties by which these systems can be distinguished, although they are of secondary importance. Subsequently, we will compare the different types of architectures, and on the basis of this comparison we argue in favour of one particular type of meta-level system, the so called bi-lingual meta-level inference systems.

1.1 Introduction

The use of meta-level architectures as the basis for building reasoning systems has become widely accepted. The past 10 years have seen the advent of many different systems that in one way or another claim to be a meta-level architecture. The main benefits of meta-level architectures are twofold: meta-level architectures allow the separation of domain knowledge (what does the system know), from control knowledge (how does the system use its knowledge), and they allow the explicit representation of this control knowledge, as opposed to implicitly building this knowledge into the implementation of the system, or mixing it with the
domain knowledge. This separate and explicit representation of control knowledge has a number of advantages, as argued by numerous authors:

- A system with explicitly and separately represented control knowledge is more modular, and therefore easier to develop, debug and modify ([7], [4] and [1]).

- It becomes possible to use the same domain knowledge for multiple purposes ([4]).

- The system can generate explanations of its own behaviour on the basis of the explicit control knowledge ([23]).

- Finally, the separation of control knowledge from domain knowledge allows domain knowledge to be purely declarative in nature. While formulating domain knowledge we do not have to worry about efficiency, only about the “representational adequacy” (in the words of McCarthy). In the control knowledge on the other hand, the efficiency of the problem solving process is the most prominent aspect (“computational adequacy”).

Many systems described in the literature provide ways of explicitly representing meta-knowledge, and of controlling the inference process. The systems that were used as the basis for the following categorisation are (in chronological order) Golux [13], FOL [25], Teiresias [7], Press [20], Neomycin [5], the Prolog systems by [10] [11] and by [2], MLA [12], 3-LISP [19], S.1 [8], BB1 [14], KRS [16], PDP-0 [15] and Socrates [6]. Where relevant, we will briefly describe some features of these systems.

The goal of this paper is to develop a classification of this multitude of different architectures, based on their essential features. Perhaps surprisingly, the diversity of meta-level systems found in the literature can be classified into a limited number of typical architectures. The advantage of such a classification is that it will allow us to compare the meta-level systems at a high level, using only their essential architectural properties, and abstracting away from incidental features.

1.2 Classification of meta-level architectures

The essential characteristic of meta-level architectures is, of course, that they consist of two levels, the object-level and the meta-level. Each layer can be seen as an individual system with a representation language and an interpreter for expressions in that language. The purpose of the object-level is to perform reasoning in the application domain of the system, while the goal of the meta-level is to control the behaviour of the object-level1. The system as a whole can at any moment

1Notice that we restrict our interest in meta-level architectures to those where the meta-level is used to control the search at the object-level. Meta-level architectures
1. A classification of meta-level architectures

be active at one of the two levels; either it is interpreting object-level expressions (using the object-level interpreter), or it is interpreting meta-level expressions (using the meta-level interpreter). This leads us to the notion of the locus of action (using a phrase coined in [24]): the place in the system which is active at any one point in time. This locus of action can then be either the object-level or the meta-level. It is exactly this locus of action that will form the basis for our main classification of meta-level architectures. We will distinguish a spectrum of systems, with at one end of the spectrum systems where the locus of action is almost all the time at the object-level, and where meta-level activity takes places only occasionally. At the other end of the spectrum we will see systems where the converse is true: almost all (or sometimes: all) system activity takes place at the meta-level, and (almost) no activity takes place at the object-level. The systems in the middle of this spectrum exhibit equal amounts of object-level and meta-level activity. This classification, plus further subdivisions, is shown in figure 1.1 and we will discuss each of the types of system in this classification in the following subsections.

can be used for many purposes, such as extending the set of results computable at the object-level, to perform meta-theoretic reasoning concerning consistency and redundancy, learning etc, but these other uses of meta-level architectures are not our primary interest in this paper.
1.2.1 Object-level inference systems

On one extreme of the classification shown in figure 1.1 are the systems where
the main activity is at the object-level. In fact, these systems do not have a
proper meta-level interpreter (ie. an interpreter for meta-level expressions), but
only an object-level interpreter that takes the meta-level expressions into account
during its computational cycle in order to adjust its behaviour. As a result,
the object-level interpreter executes two types of instructions: firstly, the object-
level expressions it is supposed to interpret, and secondly the meta-level expres-
sions that affect its behaviour. Typically, the object-level interpreter performs a
fixed computational cycle, and the meta-level expressions concern certain fixed
points within this cycle. Systems in this category are the Prolog system by Gal-
laire/Lasserre and golux.

For instance in the Prolog system developed by Gallaire and Lasserre, a num-
ber of meta-predicates can be defined that handle both clause selection and con-
junct ordering. These predicates are then used in an interpreter-loop to determine
the behaviour of the system. A fixed vocabulary of meta-predicates is available
that can express a variety of properties of the object-level propositions that com-
pete for execution. These properties include among others the number of literals
in a clause, the presence of a particular literal in a clause, the value of any ancestor
of a clause, and the invocation depth of the clause. The object-level propositions
that are present in the knowledge base can be specified in the meta-rules by their
position in the knowledge base, or by a (partial) specification of their contents.
In the Prolog syntax that is used in the system, a clause like

\[
\text{order}(p(X,Y), [N1, N2, N3, \ldots, Ni]) :- \\
C1, C2, C3, \ldots, Ck.
\]

states that for the resolution of literals that are an instantiation of \(p(X,Y)\) the
clauses numbered \(N1, N2, \ldots, Ni\) will be used in that order, provided the con-
ditions \(C1, \ldots, Ck\) are met. Notice that the variables \(X, Y, N1, N2, \ldots, Ni\) can be used in the conditions \(Ci\). An example of this would be

\[
\begin{align*}
\text{order}(p(X), [1, 2, 3]) & :- \text{cond1}(X). \\
\text{order}(p(X), [1, 3, 2]) & :- \text{cond2}(X). \\
\text{order}(p(X), [3, 1, 2]) & :- \text{cond3}(X).
\end{align*}
\]

This specifies a different order for the clauses for \(p(X)\) for different conditions on
the argument \(X\).

An example of content directed conflict resolution is

\[
\begin{align*}
\text{before}(p(_), \text{Clause1}, \text{Clause2}) & :- \\
& \text{length(Clause1,\_,N)}, \text{length(Clause2,\_,M)}, \\
& N < M.
\end{align*}
\]
1. A classification of meta-level architectures

which states that for the resolution of literals of the form $p(\_)$ shorter clauses will be used before longer clauses. Replacing $p(\_)$ with either a variable or a more specific term (e.g. $p(\[\])$) would enlarge or reduce the scope of this heuristic. This enables the formulation of both domain dependent and domain independent strategies.

The system also provides a mechanism for conjunct ordering. In earlier work along the same lines [10], further facilities were proposed to

- assign priority numbers to competing clauses,
- block backtracking over specified clauses (corresponding to dynamic cut introduction)
- inhibit the execution of literals until they reach some degree of instantiation.

The important point for the classification of meta-level systems is that all the meta-predicates are annotations that are used by a predefined and unchangeable interpreter. In other words, the behaviour of the interpreter (in this case the Prolog interpreter) is parametrised over the definition of meta-level annotations as described above, but cannot be redefined to any further extent. For instance, it would be impossible to specify a forward chaining interpreter in the Gallaire-Lasserre systems, since the hardwired object-level interpreter presumes a backward chaining control regime. It is important to realise that the use and meaning of all the meta-level predicates are fixed by the system by the way they are used in the predefined and hardwired object-level interpreter. It is not possible to extend this set. Thus, object-level inference systems provide only a limited amount of flexibility, and cannot be redefined beyond the scope of the annotations that the object-level interpreter takes into account.

1.2.2 Mixed-level inference systems

In the middle of figure 1.1 we find systems where the computation takes place at both the meta- and the object-level. Object-level and meta-level computations are interleaved, and some mechanism is provided for switching between the two. The computation at the object-level is monitored by the meta-level. We can further subdivide this category of systems in the middle of the spectrum on the basis of the criterion that is used for switching between object- and meta-level, as shown in figure 1.1.

Reflect-and-act systems

Sometimes the meta-level is called very frequently, before or after every object-level step. This organisation has been called a reflect-and-act loop, since the object-level “acts”, the meta-level “reflects” on the object-level actions and these
two together are chained together in a continuous loop. Systems with this architecture are the production rule system TEIRESIAS and the blackboard system BB1.

The flow of control in such reflect-and-act systems can be described as in figure 1.2. The object-level interpreter finds the set of all applicable object-level rules and passes this conflict-resolution set on to the meta-level interpreter. The meta-level interpreter uses its control knowledge to select one of these applicable rules, which is then handed down again to the object-level interpreter, which applies this rule. In a system like TEIRESIAS the control knowledge for conflict resolution is written down in meta-rules such as:

```
if $rule1$ mentions the $current-goal$
and $rule2$ does not mention the $current-goal$
then $rule1$ should be used before $rule2$
```

Crisis-management systems

Sometimes the meta-level is called only if a crisis or an impasse occurs in the object-level computation, for example when too many or not enough steps are possible at the object-level. PDP-0 for instance, is a program that models the behaviour of human problem solvers in the domain of thermodynamics. When the system is given a problem to solve, it selects a problem solving strategy on the basis of characteristics of the input problem. This strategy will be executed by the object-level problem solver. When the program comes to a dead end, for example because none of the known strategies is applicable, or because of the unexpected failure of an applied strategy, this will be noticed by a supervising
1. A classification of meta-level architectures

The flow of control in crisis-management systems is summarised in figure 1.3. The object-level interpreter uses the domain knowledge to solve a particular problem, and only hands over control to the meta-level interpreter if some kind of crisis occurs which prevents the object-level computation from continuing. The meta-level interpreter then uses its strategic (meta-level) knowledge base to try and solve this crisis. If some kind of solution has been found it is handed down to the object-level interpreter which can then proceed with the computation. Different kinds of crises can occur. One example of a crisis is when no object-level rules can be found that apply to the current subgoal. The meta-level interpreter then has to find a different subgoal for the continuation of the object-level computation. In this way we could implement user-directed backtracking, rather than the built-in standard behaviour of a system like Prolog. Another example of a crisis is when more than one object-level rule applies to the current subgoal. The object-level interpreter then turns to the meta-level for conflict resolution. In this way a reflect-and-act system can be simulated in an efficient way by a crisis-management system (efficient since the meta-level is only called if there is indeed more than one applicable object-level rule, rather than in every loop, as in reflect-and-act systems).

Subtask-management systems

Yet another approach is where the meta-level knowledge is used to partition the object-level task into a number of subtasks. In such a system, the meta-level interpreter decides on a task to be done, and this task will then be executed by the object-level interpreter. After completion of this object-level task (be it successful or not), the meta-level decides on the next subtask for the object-level. This approach is taken in s.1, NEOMYCIN and MLA.
The flow of control in subtask-management systems is described in figure 1.4. The meta-level interpreter first decides upon a subtask to be solved by the object-level interpreter. The object-level interpreter then tries to solve this subtask, and only returns to the meta-level when it has either found a solution or when it has established that it cannot solve the subtask. On the basis of this result the meta-level interpreter can then try to find a new subgoal to be solved. Subtasks are like conventional subroutines in that they can call each other, resulting in a stack-based scheduling of tasks. Subtasks can be either very simple knowledge base partitions that try to solve a particular subgoal, or they can have a more elaborate structure. The subtask concept from NEOMYCIN is a good illustration. Subtasks in NEOMYCIN consist of:

- The focus: this is the argument with which the task is called. This often represents the object to which the task is applied.
- Three sets of meta-rules: the DOBEFORE, DODURING and DOAFTER rules that represent the prologue of the task, the main body and the epilogue respectively.
- The goal that is recorded to show that the task has been accomplished. This can be seen as the result of the task if it exits successfully.
- The end-condition that may abort the task when it becomes true.

As can be seen from figures 1.3 and 1.4 the architectures for crisis- and subtask-management systems are very similar. The main differences are in the kind of data that is passed between the object- and the meta-level, and in the place where the computation starts: the systems communicate either in terms of crises and their solutions or in terms of subtasks and their solutions; furthermore, crisis-management systems initiate their computation at the object-level, while subtask-management systems start their computation at the meta-level.
1. A classification of meta-level architectures

1.2.3 Meta-level inference systems

On the right side of the spectrum in figure 1.1 we see systems where the computation mainly takes place at the meta-level. In these systems the behaviour of the object-level is fully specified at the meta-level. Using this description of the object-level, the meta-level can completely simulate the object-level inference process. This means that there is no longer a need for an explicit object-level interpreter. As a result, the object-level interpreter is no longer present in the system, and its behaviour is completely simulated by the execution of its specification at the meta-level. Systems from this category are PRESS, 3-LISP, KRS, the amalgamated Prolog system by Bowen and Kowalski, and Socrates.

As an example of meta-level inference systems we consider PRESS. This system performs symbolic algebraic manipulations. One of the main features of PRESS is that the system proceeds in its problem solving process by trying to prove theorems at the meta-level, producing object-level proofs as a side effect. The key idea of meta-level inference is that strategies are considered to be at the meta-level of the domain. That is, the strategies are axioms of a meta-theory. For example, consider the following meta-level axiom from PRESS

\[
\text{singleocc}(X, L=R) \land \\
\text{position}(X, L, P) \land \\
\text{isolate}(P, L=R, Ans) \rightarrow \\
\text{solve}(L=R, X, Ans).
\]

This can be considered procedurally as a strategy to solve an equation: to solve an equation \( L=R \) in \( X \) giving answer \( Ans \), satisfy the subgoals on the left. However, it also has a declarative meaning in the meta-theory. The declarative meaning is

"If \( L=R \) contains exactly one occurrence of \( X \), and the position of this occurrence in \( L \) is \( P \), and if the result of isolating \( X \) in \( L=R \) is \( Ans \), then \( Ans \) is a solution to the equation \( L=R \), with \( X \) as the unknown."

Note that this description refers to properties such as position and number of occurrences of \( X \). These are meta-theoretic syntactic features. A meta-level axiom as the above would get used by a typical call to the \text{solve}-procedure of PRESS, as in for example:

\[
?- \text{solve}(\log(e,x+a)+\log(e,x-a),x,Ans).
\]

The inference of PRESS occurs at the meta-level. Some of the meta-level predicates are of the form

\text{New} \text{ is the result of applying Rule to Old.}

To satisfy such a predicate, the rule \text{Rule} is applied to the expression \text{Old} to produce \text{New}. As a result of this, an algebraic transformation has occurred at
the object-level, the expression Old has been transformed into New: the object-
level strategies are executed by performing inferences in the meta-theory. Thus,
search at the object-level is replaced by search at the meta-level. This works well
because, as described in [18], the meta-level search space is much better behaved
than the object-level space. In particular, the branching rate of the meta-level
space is much lower, and most wrong choices lead to dead ends rapidly. The use
of meta-level inference moves the search process from the object-level to the meta-
level, and thereby transforms an ill-behaved search space to a better behaved
one. If the meta-level space is still too complex, it is possible to axiomatise
the control of this level, i.e. to produce a meta-meta-level. This process can in theory
be continued until the control process of the highest level becomes trivial. This
usually happens very early, so only two levels are needed.

The important difference between meta-level inference systems and object-
level inference systems as discussed above is that the meta-level predicates of a
system like PRESS are interpreted by an independent meta-level interpreter. The
set of meta-level predicates (as singleocc, isolate etc) is therefore fully ex-
tendible and redefinable, unlike the meta-level predicates in object-level inference
systems, which are a fixed set.

The problem of choosing between the various strategies that are axiomatised
at the meta-level and that are applicable at a point in the problem solving process
now depends on the proof procedure that is used for the meta-level interpreter.
The systems mentioned here use different techniques for this task. PRESS relies
on the built in, fixed behaviour of a Prolog interpreter for this task. The system
described in [21] is a re-implementation of PRESS. It does allow reasoning about
the selection of strategies, rather than using a hardwired interpreter.

An important subdivision of meta-level inference systems can be made on
the basis of the relation between the object-level language L and the meta-level
language M used by the system. On the one hand, there are the systems that
we will call monolingual. In these systems M and L are the same language, and
no syntactic distinction is made between object-level and meta-level expressions.
Examples of these systems are 3-LISP and the Gallaire/Lasserre system. In these
systems LISP and Prolog is used for both M and L. On the other hand there are
the bilingual systems that support two strictly separate languages L and M. In
order to provide upwards and downwards communication between L and M, the
languages are related via a naming convention which translates sentences, sets of
sentences and other linguistic entities of L into variable free terms of M. This
choice between monolingual and bilingual systems has a profound effect on the
structure of meta-level interpreters. Taking a Prolog system as an example, we
can program a simple meta-level Prolog interpreter in Prolog as follows:

```prolog
solve([],).
solve([G|Gs]) :-
    clause(G, B),
```
append(B, Gs, NewGs),
solve(NewGs).

This is a monolingual meta-level interpreter which relies on the mixing of object-level terms and meta-level terms for the communication between the two levels. Object-level predicates are regarded as meta-level function symbols, but variables at both levels are represented as Prolog variables, thereby not making a distinction between object-level variables (ranging over object-level terms) and meta-level variables (ranging over object-level formulas). As a result, the built-in unification algorithm of Prolog can be used at both levels. A bilingual version of the same Prolog interpreter in Prolog would look like

solve([], []).
solve([G|Gs], S) :-
    clause(G, C),
    rename_vars(C, [G|Gs], C1),
    head(C1, H1),
    unify(G, H1, S1),
    body(C1, B1),
    append(B1, Gs, NewGs),
    instantiate(NewGs, S1, NewGsPlusS1),
    solve(NewGsPlusS1, S2),
    compose(S1, S2, S).

This version of the Prolog meta-interpreter in Prolog allows for a separate representation of object- and meta-level variables. Only meta-level variables are represented by Prolog variables, and object-level variables correspond to particular kinds of variable free meta-level terms. As a result, an object-level formula such as $\forall x \forall y f(x, y)$ can be represented as the ground meta-level term all(var(1), all(var(2), f(var(1), var(2)))). Simpler naming relations are also possible, such as the standard quoting relation, where each object-level term is represented by a single meta-level atom: "$\forall x \forall y f(x, y)"$. This is the naming relation used in Socrates. A third approach is presented in [2], and consists of amalgamating $L$ and $M$. In this approach $L$ and $M$ are in fact the same language, but the naming convention is employed as above, so that each object-level expression has a variable free term as its name associated with it. (Since this variable free term is itself again a syntactically correct object-level expression, since $L=M$, it again must have another variable free term as its name, ad infinitum).

This concludes our classification of meta-level architectures according to their locus of action. This distinction corresponds roughly to a distinction made by Silver [18] between object-level driven and meta-level driven systems. Systems at the left end of the spectrum shown in figure 1.1 are object-level driven, since their main computation takes place at the object-level, and systems at the right end of the spectrum are meta-level driven.
1.3 Other properties of meta-level architectures

In this section we will discuss some more properties of meta-level architectures that can be used to further classify them, beyond the classification on the basis of the locus of action as described in the previous section. Some of the properties discussed in this section are found implicitly in the literature. The aim of this section is to sharpen these criteria, and to make them explicit. Figure 1.5 at the end of this section will summarise the position of all the systems mentioned above with respect to all of these properties.

1.3.1 Linguistic relation between levels

The distinction between monolingual, bilingual and amalgamated systems used to subdivide meta-level inference systems in the previous section can in fact be used more generally, to apply also to object-level inference and mixed-level inference systems.

Object-level inference systems can have their meta-level instructions to the object-level interpreter expressed in a language that is either the same as or different from the object-level language. The Gallaire/Lasserre system relies on the two languages being the same, whereas GOLUX separates the two languages, providing an explicit quoting mechanism to give meta-level names to object-level expressions (using this mechanism, the ground terms at the meta-level that are used as the names of object-level expressions are always atomic constants).

The distinction between monolingual, bilingual and amalgamated systems also applies to mixed-level systems. TEIRESIAS is a bilingual system: although both object-level and meta-level language are production rule languages, they are quite different languages, properly separated; the languages are of the same type, namely production rule languages, but they are not identical. This is similar to GOLUX, where both meta-level and object-level language are of the same type (first order predicate calculus), but are in fact separate languages. The same remark holds for MLA. NEOMYCIN and S.1 are also bilingual systems. but in both these systems the meta-level language is not only different from the object-level language, but also of a different type: Both NEOMYCIN and S.1 use production rules at the object-level, but use some special purpose language at the meta-level: the task-language in NEOMYCIN and so called control-blocks in S.1.

1.3.2 Declarative or procedural meta-language

The control knowledge of the meta-level can be either expressed in a declarative or a procedural language. Expressions in a procedural language can only be understood in terms of the behaviour of the meta-level interpreter, the actions it takes, the order in which it does things etc., whereas a declarative language states true facts that can be understood without reference to the behaviour of the meta-level interpreter. For example, the meta-level expressions of GOLUX are
purely declarative descriptions of properties of object-level proof trees, whereas something like the control knowledge of §1 is purely procedural in nature, talking about the order in which to perform actions, sequences and loops of instructions etc. Yet other systems (such as PRESS) have a meta-level language that has both a declarative and a procedural reading.

1.3.3 Partial specifications

An important property of some of the systems mentioned above is that they allow the partial specification of meta-level knowledge. Such systems (like GOLUX, Gallaire/Lasserre, MLA, 3-LISP, KRS and Bowen/Kowalski) provide a default specification of the behaviour of the system that can be totally or partially overwritten by the user to modify the systems default behaviour. In some of these systems the default definition is explicitly available for inspection in the system (3-LISP, KRS, Bowen/Kowalski), whereas the other systems (GOLUX, Gallaire/Lasserre, MLA) only contain an implicit definition of their default behaviour.

1.3.4 Completeness and strictness

As mentioned in the introduction, one of the main purposes of having a meta-level architecture at all is to allow the object-level to be purely declarative, without having to worry about procedural aspects. Thus, for any given query, the object-level (implicitly) specifies a set of answers. It is the task of the meta-level interpreter to determine which of these possible answers is going to be actually computed, and in which order. Furthermore, in some systems, it is possible for the meta-level to extend the set of answers derivable from the object-level theory, using meta-theoretic devices like reflection principles [9], implemented for instance in FOL [25]. We call a meta-level architecture complete if it computes all the results derivable from the object-level theory (i.e. the meta-level does not suppress any object-level results). We call a meta-level system strict if it computes only results derivable from the object-level theory (i.e. the meta-level does not extend the object-level results). Notice that it is possible for a meta-level architecture to be both incomplete and unstrict, namely if the meta-level suppresses some of the object-level results, but also computes extra ones. We use the term exact when a meta-level system is both complete and strict: an exact system computes all and only those results derivable from the declarative object-level theory.

With these definitions it is clear that completeness of a meta-level system is not always a desirable property. The whole point of a meta-level architecture is often to prune parts of the object-level search space, thereby suppressing certain object-level results because they are too expensive to compute. [22] used the term

\[\text{This is most clear is the object-level consists of a logical theory plus a set of logical inference rules, but similar notions exist for other declarative representation languages.}\]
“positive heuristic” in connection with the concept of completeness: a positive heuristic is

“a heuristic that prefers certain object-level computations over others, but that does not prohibit certain computations altogether.”

In other words, a system that only allows positive heuristics is automatically complete. Whether strictness of a meta-level system in the above sense is always a desirable property is less clear. In the context of using meta-level systems for control, we probably do not want to extend the results of the object-level theory, but one possible use of a meta-level architecture (although not our interest here) is exactly to try and extend the results of the object-level theory (FOL is an example of this).

A system like the Gallaire/Lasserre Prolog system is one of the few systems mentioned above that is complete: its only concern is the ordering of clauses and literals, thereby affecting only the order in which the object-level computation takes place, but not its ultimate outcome\(^3\). A system like GOLUX is not complete: for instance we can force proofs in GOLUX to be deterministic, thereby pruning alternative solutions. However, GOLUX is strict in the above sense: no new object-level results can be introduced through meta-level computation. This is not true in meta-level inference systems like KRS and 3-LISP: since their meta-levels completely specify the object-level computation, it is well possible to extend the object-level behaviour in order to produce extra results.

This concludes our discussion of different properties to distinguish meta-level architectures. The table in figure 1.5 summarises the position of all the systems mentioned above on these properties.

1.4 Comparison of the different architectures

In this section we will compare the architectures discussed above, and we will conclude that the one based on meta-level inference is most promising. The other approaches all have major problems associated with them. Meta-level inference does not suffer from these problems, while it offers several advantages.

The most obvious problem is associated with the object-level-inference systems. The meta-level does not have a separate place in the architecture of these systems, and is not stated as explicitly as would be necessary in order to achieve the advantages of a meta-level architecture (better explanation, re-usability and ease of development and debugging). The main structure of the control strategy of these systems is only implicit in the system. Although possibly available for inspection, it is never available for modification, and only a restricted number of aspects of the control strategy can be changed. For instance in the Gallaire/Lasserre

\(^3\)Strictly speaking, this is only true for those versions of their system that do not allow dynamic cut-introduction. This corresponds to dynamically removing object-level backtrack-points, thereby potentially suppressing certain object-level results.
## 1. A classification of meta-level architectures

<table>
<thead>
<tr>
<th>System</th>
<th>Architect. type</th>
<th>linguistic relation</th>
<th>declarative/ procedural</th>
<th>partial spec.</th>
<th>strict &amp; complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEIRESIAS</td>
<td>reflect-act</td>
<td>bi-ling.</td>
<td>decl.</td>
<td>yes</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>S.1</td>
<td>task-man.</td>
<td>bi-ling.</td>
<td>proc.</td>
<td>?</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>BB1</td>
<td>reflect-act</td>
<td>bi-ling.</td>
<td>proc.</td>
<td>yes</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>MLA</td>
<td>task-man.</td>
<td>bi-ling.</td>
<td>decl.</td>
<td>yes</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>KRS</td>
<td>meta-inf.</td>
<td>mono-ling.</td>
<td>1</td>
<td>yes</td>
<td>C– ; S–</td>
</tr>
<tr>
<td>NEOMYCIN</td>
<td>task-man.</td>
<td>bi-ling.</td>
<td>proc.</td>
<td>2</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>PRESS</td>
<td>meta-inf.</td>
<td>bi-ling.</td>
<td>decl.&amp;proc.</td>
<td>2</td>
<td>; S+</td>
</tr>
<tr>
<td>GOLUX</td>
<td>object-inf.</td>
<td>bi-ling.</td>
<td>decl.</td>
<td>yes</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>Gallaire</td>
<td>object-inf.</td>
<td>mono-ling.</td>
<td>decl.&amp;proc.</td>
<td>yes</td>
<td>C+ ; S+</td>
</tr>
<tr>
<td>Bowen</td>
<td>meta-inf.</td>
<td>amalgam.</td>
<td>decl.&amp;proc.</td>
<td>yes</td>
<td>C– ; S+</td>
</tr>
<tr>
<td>3-LISP</td>
<td>meta-inf.</td>
<td>mono-ling.</td>
<td>proc.</td>
<td>yes</td>
<td>C– ; S–</td>
</tr>
<tr>
<td>Socrates</td>
<td>meta-inf.</td>
<td>bi-ling.</td>
<td>decl.&amp;proc.</td>
<td>no</td>
<td>C– ; S–</td>
</tr>
</tbody>
</table>

**Legend:**
- ? = unknown, cannot be determined from available literature.
- n = not applicable, see note n.
- C+/– = completeness enforced/not enforced.
- S+/– = strictness enforced/not enforced.

**Notes:**

1. It is unclear how the object-oriented paradigm relates to the distinction declarative vs. procedural.

2. The possibility of partial specifications does not occur in this system since it is a program built for a particular task, containing a full specification of one appropriate control regime.

3. The system can be made unstrict because the confusion between object-level and meta-level language (monolingual) allows the meta-level predicates to introduce arbitrary bindings for the object-level variables.

4. The system is only complete without the facility of dynamic cut-introduction.

---

Figure 1.5: Properties of meta-level systems
system, it is possible to change the clause- and literal-selection strategies, but the fact that the system is chronologically backtracking and backward chaining is hardwired.

The mixed level systems do not suffer from this problem, but they have other problems associated with them, which can best be discussed using the subcategories from section 2:

A problem that is associated with both the crisis-management systems and the reflect-and-act systems is that the search in the solution space is still performed at the object-level. As a result, the meta-level knowledge is only used as a preference criterion over the separate object-level search space, whereas in systems that are more meta-level driven the meta-level knowledge is used to completely specify the whole structure of the search space of the system. This means that no full advantage is taken from the fact that the meta-level search space is better behaved than the object-level search space.

A problem associated with the task-management systems is what could be called the black-box effect: after the meta-level has decided on a task to be performed by the object-level, the object-level is no longer under the control of the meta-level, and again no full benefit is gained from the differences between meta-level and object-level search.

None of the mixed-level inference systems makes all the control knowledge in the system explicit: reflect-and-act systems only deal with conflict resolution strategies, crisis-management systems know how to solve impasses in the computation and subtask-management systems represent the selection of goals and subgoals, but none of them contains a full description of the object-level computation.

Meta-level inference systems do not suffer from these problems. In meta-level inference systems, the meta-knowledge is not just used as preference criteria over the separate object-level search space, but it is used to completely specify the whole structure of the search space. Meta-level inference systems perform their search in the meta-level search space and thereby gain the full benefit of the nicer properties of the meta-level search space. Furthermore, meta-level inference systems contain a full specification of the inference strategy of the system, thereby allowing the user to change any part of this strategy, and not just only a few predefined aspects of it.

This leaves us with the choice between the different subtypes of meta-level inference systems: monolingual, bilingual and amalgamated systems. A number of reasons can be given why it is important for the meta-level language to be separated from the object-level languages, thus ruling out both the mono-lingual and the amalgamated systems. First of all, there is an epistemological reason. [17] and [3] argue that different domains require different representation languages. Since the object-level and the meta-level deal with widely different domains (the object-level deals with the application domain of the system, while the meta-level deals with the issue of controlling the object-level), it follows that these two
levels of the system do indeed need different representation languages to suit their different needs.

A second argument concerns the modularity of the system. One of the advantages of separating control knowledge from domain knowledge is that the two can be changed independently. However, this ability to vary the two levels independently would be greatly reduced if control knowledge and domain knowledge were represented together in one and the same language.

The third argument is one about explanation. As argued in [23], the explanations given by reasoning systems should not only include what the system is doing, but also why it is doing a particular action and not another one. In order to enable the system to include control knowledge explicitly in its explanations, it is important that control knowledge can be syntactically distinguished from domain knowledge.

A further problem associated with the amalgamated approach is the recursive application of the naming convention. Each ground term in the meta-level language that is the name of an object-level formula is itself again an object-level formula (since the two languages are the same), and thus has some ground term as its name, etc. This introduces the possibility of self-referential sentences, which is necessary for introspection, or for incompleteness proof à la Gödel (the self-referential capability is used in exactly this way in [2]). However, if we are not interested in these aspects of meta-level reasoning this added complexity is not needed, and we can get away with the much simpler construction of separate languages.

Thus having narrowed our choice down to bi-lingual meta-level inference system, we still have to discuss the best position on the other properties of meta-level systems described in the previous section: the possibility of partial specifications, a declarative versus a procedural meta-level language, and enforcing strictness and completeness on the meta-level.

The possibility of writing only partial specifications of the control strategy of the system is obviously attractive for the development of a system. We can gradually refine the control strategy of the system, without overcommitting ourselves at any point, postponing decisions until we understand enough of the domain. However, a high price needs to be paid for this possibility, resulting in a severe restriction of the system’s architecture. In order for the system to be able to “fill in the gaps” of the partial specification of the control regime by the user, it is necessary that this partial specification is of a particular format, so that it is possible for the system to identify which parts of the control regime are underspecified, and need to be filled in with default values. This restricts the possible range of control regimes that can be formulated by the user.

Concerning the issues of strictness and completeness we can say the following: For reasons discussed before, we would not want to enforce completeness on a meta-level architecture, since often the whole point of having a meta-level is to be able to avoid the expensive computation of certain object-level results. Whether
strictness, not allowing the meta-level to extend the set of results that can be computed by the meta-level, is a desirable property, is less clear in general, and depends on the purpose for which the architecture will be used.

1.5 Conclusion

We have categorised the meta-level systems described in the literature, and have distinguished the following types:

- object-level inference systems
- mixed-level inference systems, which can be divided into
  - reflect-and-act systems
  - crisis-management systems
  - subtask-management systems
- meta-level inference systems, which can be divided into
  - mono-lingual systems
  - bi-lingual systems
  - amalgated systems

Furthermore, a number of secondary properties of meta-level architectures were identified:

- Is the meta-level language declarative or procedural.
- Does the system allow partial specifications of the control regime.
- Does the system enforce strictness and completeness of the control regime.

We have compared these systems, and have argued in favour of bi-lingual, meta-level inference systems.

Acknowledgements

I am grateful to Han Reichgelt and Fausto Giunchiglia for comments on earlier versions of this paper. The work reported here was made possible by the Alvey/SERC grant “A flexible toolkit for expert systems”, grant no. GR/D/17151. The author is currently supported on grant no. GR/D/44270.

Bibliography


