Knowledge-level Reflection*

Frank van Harmelen[†] Bob Wielinga[†]

Bert Bredeweg[†] Guus Schreiber[†] Werner Karbach[‡]
Martin Reinders[§] Angi Voß[‡] Hans Akkermans[§]
Brigitte Bartsch-Spörl[¶] Erik Vinkhuyzen[†]

[†]UvA [‡]GMD [§]ECN [¶]BSR

Abstract

This paper presents an overview of the REFLECT project. It defines the notion of *knowledge level reflection* that has been central to the project, it compares this notion with existing approaches to reflection in related fields, and investigates some of the consequences of the concept of knowledge level reflection: what is a general architecture for knowledge level reflection, how to model the object component in such an architecture, what is the nature of reflective theories, how can we design such architectures, and what are the results of our actual experiments with such systems?

Keywords: Knowledge-based systems, reflection, meta-reasoning.

1 Problem statement

1.1 Definition

Current knowledge-based systems lack the capabilities to perform a number of functions which are generally attributed to human experts: competence assessment, knowledge base maintenance, knowledge refinement, sensible explanation, adaptive interaction, performance validation, verification of internal knowledge consistency, etc. The fundamental premise that underlies this paper is that such advanced functions of knowledge-based systems can only be realised by creating a *reflective system*, i.e. a system that is able to reason about itself, its own knowledge of a problem domain, its problem solving strategies, the scope of its competence and its history in terms of solved cases. Reflective

*The research reported here was carried out in the course of the REFLECT project. This project is partially funded by the Esprit Basic Research Programme of the Commission of the European Communities as project number 4178. The partners in this project are SWI, University of Amsterdam (Amsterdam, The Netherlands), the Netherlands Energy Research Foundation ECN (Petten, The Netherlands), the National German Research Centre for Computer Science GMD (St. Augustin, Germany) and BSR Consulting (Munich, Germany).

reasoning requires a self-representation, a mechanism for reasoning about and with such a self-representation, and a means of controlling the interaction between the object problem solver and the reflective component. Through reflective reasoning the system is able to know why it can or cannot solve a particular problem. It is capable of indicating where gaps in its knowledge base exist and how these gaps may be filled. Also, a reflective system is able to act upon the object problem solver, e.g. to adapt its behaviour in case a difficult problem is encountered or when an impasse in the problem solving process occurs. The goal of this paper is to investigate how more flexible and robust knowledge-based systems can be constructed using the concept of reflection.

Before we can elaborate this goal, we first have to establish precisely what we mean by reflection and reflective systems and what we consider to be the scope of reflection.

1.2 Context

Reflection in its common sense meaning refers to the activity of pondering about oneself, about one's own behaviour and about one's relation to the outside world. In recent AI research the term reflective system is used to identify a class of systems which in some way or other have knowledge about themselves and which are able to reason about and act upon themselves. In this view reflective systems are closely related, but not identical to meta-level systems [Maes & Nardi, 1988].

Maes [Maes, 1987] defines a reflective system as a computational system which has not only knowledge and data about some part of the external world, the object domain, enabling it to reason about and -sometimes act upon- that domain, but which also has the ability to reason and act upon itself. Reflection requires a self-representation, which is causally connected [Smith, 1982] to the object system such that actions in the object system are reflected in the self-representation and changes in the self-representation are reflected in the object system. Maes distinguishes reflective systems from meta-systems. etc In this paper we take a broader view on reflection: A reflective system is a system that has knowledge about some part of itself: the object component. This knowledge is called the *model of the object component*. In addition a reflective system contains meta-knowledge which allows it to reason about and act upon the object component. In principle the object component and the meta component could coincide, in which case the system is a fully reflective system. In many cases, however, the knowledge about the object part of the system is only partial. In that case the system can be viewed as consisting of a *meta-system* and an *object system*. In this view, systems that perform meta-level reasoning as Maes defines it [Maes, 1987], are considered a subset of (partially) reflective systems.

Questions addressed in this paper

- How can we realise flexible and robust behaviour of knowledge-based systems using reflection?
- How do various views on reflection relate to each other? (section 2)
- What is the nature of the object model needed to perform a variety of reflective tasks (sections 3 and 4).

- What meta-knowledge needs to be represented in order to achieve reflective tasks (section 5).
- What architectural choices do we have when designing reflective systems and what are the consequences of these choices (section 6).

Section 7 presents some examples of reflective systems that have been implemented.

2 Approach

2.1 Background

In this section we will discuss work on reflection in two distinct, though related areas of science: Logic and Computer Science. For each of these areas we will give a brief summary of the major topics, problems, approaches and solutions concerning reflection, and we will indicate what the impact of this work is on the approach to reflection described in this paper.

2.1.1 Reflection in Logic

Both building formal systems that describe other formal systems, and building systems with self-referential capabilities are activities with a long standing tradition in logic and meta-mathematics. The purpose of this section is to present briefly the most important results achieved by logicians in this area.

Encoding truth and provability

The earliest, and probably most widely known work on self-referential systems is that by Gödel [Godel, 1931]. He devised a way of *encoding* certain meta-statements concerning the syntax and proof theory of his object-formalism (Peano Arithmetic) as statements *in* that same object-formalism. Thus, predicates stating, for instance, that one formula concerning natural numbers followed from another by some rule of inference, were represented by object-predicates using natural numbers. Using this construction, Gödel could prove certain properties of such self-encoding systems: any theory in which such encoding is possible must be incomplete (first incompleteness theorem), and no consistent theory can prove its own consistency (second incompleteness theorem). The proofs of these (mostly negative) results relied on the possibility of constructing *self-referential sentences*, which encoded their own unprovability, such as the sentence $\phi : \neg Prov(\overline{\phi})$ (where $\overline{\phi}$ stands for the encoding of ϕ).

A closely related result was obtained by Tarski [Tarski, 1936]. He proved that if we explicitly add to a theory a *truth predicate* defined by $True(\overline{\phi}) \leftrightarrow \phi$ (known as the *Tarski biconditionals*), then the theory becomes inconsistent. Again, this proof relies on the construction of a self-referential sentence, the liar sentence $L : \neg True(\overline{L})$, which leads to a contradiction. Attempts have been made to avoid the negative results of Gödel and Tarski. Kripke [Kripke, 1975] suggested a construction in which $True(\overline{\phi}) \vee True(\overline{\neg\phi})$ only holds for some formulae ϕ. A closely related suggestion is made by Perlis [Perlis, 1985]. The recent book by Turner [Turner, 1990] provides an excellent and thorough exploration

of these and other attempts at encoding truth and provability, but makes it clear that many of the issues arising from self-reference are still unresolved.

Inter-theory inference

It is in the work of Feferman that the term *reflection principle* originated [Feferman, 1962; page 274]:

> "By a reflection principle we understand a description of a procedure for adding to any set of axioms A certain new axioms whose validity follows from the validity of the axioms A and which formally express, within the language of A, evident consequences of the assumption that all the theorems of A are valid."

A somewhat different use of the term "reflection principle" is found in the work on logic in AI. Originally, in [Weyhrauch, 1980], we find reflection principles defined as: "a statement of a relation between a theory and its meta-theory" which is then formalised as an inference rule with its premise and conclusion in different theories (namely the object- and meta-theories of the system). The standard examples of this form of reflection principle as an inter-theory inference rule are the rules *up* and *down*, reflecting between an object-theory $\mathcal{O}$ and a meta-theory $\mathcal{M}$:

$$\frac{\vdash \Delta O\phi}{\vdash \Delta MProv(\overline{\phi})}\ up \qquad\qquad \frac{\vdash \Delta MProv(\overline{\phi})}{\vdash \Delta O\phi}\ down.$$

This situation is close to that described by the Tarski biconditionals, although the difference is of course that here M and O can be different theories, thus avoiding the contradiction-generating self-referential sentences.

These rules *up* and *down* and the associated two-theory framework have been implemented in FOL [Weyhrauch, 1980]. The system imposes no constraints on the definition of *Prov* as given by the user, and FOL can thus be used to implement either truthful, enlarged or partial reflection, which are defined in the literature review [Giunchiglia & Smaill, 1988] as follows: truthfulness only allows the extension of the $\mathcal{O}$ with results consistent with $\mathcal{O}$; in enlarged reflection truthfulness is dropped, and partial reflection only allows the extension of $\mathcal{O}$ with results already derivable in $\mathcal{O}$ itself (ie. definitional extensions).

Although the rules *up* and *down* are the reflection rules most often mentioned in the literature, they are by no means the only possibility [Giunchiglia & Serafini, 1990] generalises the notion to arbitrary inference rules that link two theories.

Naming

In the above, we have written $Prov(x)$ and $True(x)$ without explicitly stating the syntactic status of x. If we want to stay within the context of first order logic, and thus require *Prov* and *True* to be first order predicates (as opposed to, say, modal operators or second order predicates), then x must be a first order term, in particular the *name* of a first order formula. This implies that sentences of our object-language must be named by terms which can be used as argument to the meta-predicates. Logicians have employed two types of naming [Tarski, 1936], namely *quotation-mark names* and *structural descriptive names*. Quotation-mark names associate with a formula ϕ a constant as its name, often

written as $\overline{\phi}$ or "ϕ". Gödel's encoding, described above, is an example of this type of naming. Structural descriptive names are arbitrarily complex ground terms which reflect the structure of the sentence they name.

Both these types of naming relations are entirely "syntactic", in the sense that names are determined only by the syntactic structure of the object-expression. [vanHarmelen, 1992] on the other hand argues that naming can also be defined on non-syntactic grounds (semantic, pragmatic), and that this opens up the possibility of exploiting the name relation to encode more information about the object-theory than is possible with purely syntactic naming.

The relation between languages

As discussed in the above, it is of crucial importance whether the meta-predicates such as *Prov* and the names such as $\overline{\phi}$ belong to the same theory as the sentences ϕ themselves. Gödel's system, where formulae about natural numbers were encoded as (or: named by) natural numbers, the formulae and their names belonged to the same theory, whereas the bi-lingual framework of FOL separates the two, as is indicated by the subscripts M and O to the derivability symbol in the rules *up* and *down* above and thus avoids the problems caused by self-reference (since we have $\phi \in O$, $\overline{\phi} \in M$ and $O \cap M = \emptyset$).

In the context of the rules *up* and *down*, if all objects (i.e. formulae and their names) belonged to the same language (i.e. $\phi \in O$, $\overline{\phi} \in O$, and $M = O$), also called the *amalgamated* approach, the rule *up* could be proved as a derived inference rule (in other words, it is a form of partial reflection), whereas the rule *down* is an enlarged reflection principle.

The rules *up* and *down* are also presented in [Bowen & Kowalski, 1982], where the authors start from an object-theory O, describe an associated meta-theory M, and then "amalgamate" the two theories.

Summary

In the above, we have given a brief overview of the work on reflection, encoding and self-reference in logic.

Fundamental is the work of Gödel and Tarski, leading to mostly negative results about the properties of self-encoding systems (incompleteness, inconsistency).

Work by Kripke and Perlis attempts to avoid these negative results by introducing variants of the truth predicate. Other attempts at avoiding the inconsistencies of self-encoding systems are the iterated extensions of axiom sets by Turing and Feferman, where the word "reflection principle" originated.

A third approach, aimed at avoiding self-reference altogether, is to separate object- and meta-theory, as proposed by Weyhrauch. This leads to viewing reflection principles as inter-theory inference rules.

Fundamental in all this is the notion of naming formulae by ground terms, so that the encoding of an object-theory in a meta-theory can be done within a first order framework.

A final strand in the work on reflection in logic has been on the encoding of belief, in contrast with the work described above which is concerned with encoding truth and provability. Work on encoding belief in logic has been based on either modal or first order formalisms, but both approaches suffer from problems similar to those encountered in encoding truth and provability.

2.1.2 Computational Reflection

The term reflection has been used in computer science to denote computational systems that, in addition to their computation on aspects of the world, perform computation on aspects of themselves, i.e., their data, program or execution process. A reflective system has a meta-level architecture. The part of the system that is reasoned about, the object-level, is represented at the meta-level. This self-representation and the object-level are *causally connected*, a term that is informally described in [Maes, 1987] as a link between a representation and a system that it represents, such that if one of these changes, the other changes accordingly (the representation is still truthful). In this context [Smith, 1982] introduces the notions *introspective integrity* and *introspective force*. Introspective integrity concerns the question whether any significant property of the representation at the meta-level is in accordance with its content (the represented object-level). Since integrity is a quality of the representation at the meta-level, it is realised by the causal connection upwards. Introspective force involves the realisation of a system's reflective goals in the object-level through the causal connection downwards.

The implementation of a reflective computational system is according to a particular *reflective architecture* that determines the representation (languages and interpreters) of, and the control flow between the system components and provides a mechanism implementing the causal connections. A *reflective programming language* is a language that supports programming reflective systems. Such a language has facilities that support the writing of computational systems that are "able to access and manipulate causally connected representations of themselves during computation" [Maes, 1987]. Although reflective systems need not be programmed in reflective languages, most work in computational reflection is focussed on the definition of reflective languages.

Reflective languages

Most reflective computation in languages that constitute landmarks in the brief history of computational reflection concerns *control* of the object-level computation.

In 3-LISP [Smith, 1984] this is realised by reflective functions that are interpreted by meta-circular interpreters. A meta-circular interpreter is an explicit representation of the interpreter in the language itself which is used to actually run the language. The meta-circular interpreter makes explicit the the variable binding environment and the continuation of the interpreter at the lower level which can then be inspected and influenced by the reflective function.

3-KRS [Maes, 1987] is an object-oriented programming language. Reflective computation is specified by means of meta-objects that represent reflective information about and that can access and modify the computation performed by other objects. Again these are evaluated by a meta-circular interpreter.

TEIRESIAS [Davis, 1980] is an early example of a rule language that allows meta-rules to order and prune a set of object-rules[1] that are plausible for reaching an object-level goal. Meta-rules in TEIRESIAS describe the domain specific content of plausible object-rules, but do not mention them explicitly. Abstracting away from the domain, meta-rules

[1] [Davis, 1980] uses the term knowledge source in order to indicate that any form of encoding can be used, not only rules.

in HERACLES refer to knowledge roles rather than specific domain objects [Clancey, 1983].

SOAR [Laird *et al.*, 1987] is an attempt to provide an architecture for general intelligence. It is dedicated to the problem space hypothesis: all goal-oriented behaviour is based on search in problem spaces. Although not originally conceived as a meta-level or reflective architecture, SOAR can be specified as a multi-level architecture that can reflect upon its own problem solving behaviour [Rosenbloom *et al.*, 1988]. The architecture has three distinct meta-levels: (1) a problem space level that contains operators, states and goals in contexts; (2) a production level that applies productions (condition-action pairs) from long-term memory to the objects in working memory in order to implement operators; (3) a preference level that selects between alternative candidate objects resulting from productions or that signals an impasse, *i.e.*, the impossibility to select an object or apply an operator. In the latter case a subgoal is setup that makes explicit the impasse and why and where it occurred. The subgoal to resolve the impasse is carried out by search in a new problem solving context for which the same three-level architecture is applied (and which may lead to new impasses that are resolved at higher levels). For example, an impasse in operator application is resolved by searches for a state that allows execution of the operator. A selection impasse is resolved by look-ahead search. In this manner the full capacity of SOAR can be applied to basic operations as well as to resolving impasses by reflecting upon the problem solving capabilities itself.

An architecture for computational reflection more oriented towards logic is FOL [Weyhrauch, 1980], that distinguishes theories and meta-theories in first order logic. The notable purpose of reflection here is to change theorem proving in the object-theory into evaluation in the meta-theory. Such a meta-theory may for example consist of subsidiary deduction rules that shorten a proof.

Recent work on meta-programming in logic programming resulted in various proposals for meta-level extensions to logic programming languages in the form of meta-interpreters, following the seminal work of [Bowen & Kowalski, 1982]. In particular, GÖDEL [Hill & Lloyd, 1991] is a serious attempt to provide meta-logical facilities with a declarative semantics. The approach here is to make a clear distinction between the language at the object-level and its representation via a syntactic naming relation.

Summary

Reflection in computational systems refers to the computation of a program on aspects of the program itself. Its purpose in existing architectures has been mainly control of object-level computation. In order to achieve that purpose a variety of architectures is conceivable. These differ on the self-representation of a system, the causal connection between the self-representation and the object-level, the computation needed when performing reflection and the locus of action between object- and meta-level.

2.2 Definition of the approach: Knowledge level reflection

From the literature discussed in the previous section, a number of topics seem to re-occur as issues concerning reflection in each of the three subfields. In section 2.2.1 we first describe these (five) re-occurring issues and devote a subsection to each. In 2.2.2, we subsequently define our own approach to building reflective systems in terms of specific

positions that we take on each of these issues, again devoting a subsection to each of the issues.

2.2.1 Issues in building reflective systems

The nature of the object-model

By *object-model* we mean the representation, or model, that the meta-system has of the object-system, and through which it reasons about the object-system. In logic, the object-model is created through applying the (syntactically defined) naming relation to object-expressions, resulting in the meta-terms which name the object-expressions, and which are the subject of the meta-theory.

In many of the computational reflective systems, the object-representation captures various degrees of abstractions of the computational mechanisms of the object-system: 3-LISP's meta-theories deal with function-call-continuations and bindings environments, those of 3-KRS deal with message passing and inheritance, and TEIRESIAS deals with conflict resolutions sets, which shows that all of these reflective systems deal with computational aspects of their object-system.

Cognitive science has little consensus of the nature and structure of the model we have of our own knowledge. The SOAR architecture, which does propose a specific model, does indeed originate in cognitive science, but does not in any way represent an emerging view in the field.

Separation vs. amalgamation

This issue concerns the relation between meta-and object-system. Are they two separate systems that communicate with each other, or is one a subpart of the other? In logic, where this issue corresponds to the relations between the languages used to express object- and meta-theory, both approaches have been investigated. In computer science, both approaches are also present, with e.g. FOL representing a separated approach, and 3-LISP and 3-KRS an amalgamated approach. In cognitive science this issue is found in the use/mention distinction that underlies reflective access.

Causal connection

The problems of connecting object- and meta-system is a third issue which occurs in each of the three subfields dealing with reflection. The reflection rules in logic, the causal connection in computational systems and the influence that meta-cognitive processes have on the way we solve a problem are all concerned with connecting object- and meta-component, and specify how information flows from one to the other, and how the two complements influence each other.

Switching the locus of action

A direct implication of our choice of separating the object-system from the meta-system is that we must decide which system is active at any one point. In the computer science literature, this is described as the switch of the locus of action, and in cognitive science

this amounts to switching between cognitive and meta-cognitive activities. The notion of switching is not directly found in logic, since logic deliberately abstracts from control.

The nature of reflective theories

All three approaches to reflection share an interest in the nature of the contents of the reflective components. How general can they be formulated, how independent from the object-system can they be, do they have a generic form, what topics of interest can be treated in such reflective theories, etc, are all questions concerning the general nature of meta-theories that are addressed in each of the three subfields.

2.2.2 Knowledge level reflection

In this section we will define our own approach to reflection, which differs from the existing approaches outlined in section 2.1. We will define our approach in terms of our position on each of the four general issues in building reflective systems that were discusses in the previous section. Our position on the first issue (the nature of the self-representation) is the one that is most markedly different from existing approaches. The choice that we make concerning this issue will then largely determine our position on each of the other general issues.

Abstract object-models

It is the treatment of the object-model which separates our approach to reflection most markedly from existing approaches in the literature. Both logic-based and computational approaches to reflection use object-models which make detailed commitments to the syntactic and computational aspects of their object-system: logic's syntactic naming mirrors the object-system's syntax in the meta-theory, and the meta-components of computational systems like 3-LISP and 3-KRS make detailed assumptions about the computational mechanisms of their object-systems.

In contrast with all of this, our own approach to reflection is based on the idea that a meta-theory should, as much as possible, be *independent* from such syntactic or computational details of its object-system. A meta-theory should be concerned with the *functional behaviour* of its object-system, independently from the syntactic and computational details of how this behaviour is realised. We have coined the term *knowledge-level reflection* form this idea, since our point of view resembles that advocated by Newell [Newell, 1982], who argued that the "knowledge level" would be the appropriate level for describing AI systems. The term "knowledge-level" is the cause of much confused debate in the AI community. In the context of this paper, no more should be read in this term than that a "knowledge-level representation" is an abstract representation that captures essential problem solving capabilities without defining how such capabilities should be computationally realised.

Thus, a meta-theory should be concerned with the kind of task performed by the object-system (such as diagnosis, monitoring, design, etc), and with the knowledge needed to perform that task, but *not* with the computational mechanisms that are used to perform that task or to represent that knowledge, be they logical deduction, messages between objects or production rules.

This choice for abstracting away in the meta-theory and object-model from computational and implementational details is motivated and justified by the type of reflective task that we want to tackle in the meta-theory. As discussed in the introduction, these are tasks like competence assessment, sensible explanation, adaptive interaction, etc. All of these tasks (and many more) are concerned with the "essential problem solving capabilities" of the object-system, quite independently from how these capabilities are realised.

Sections 3.1 and 3.2 will be devoted to discussing the structure and contents of both the object-models and the meta-theories required for such knowledge level reflection, but we will first investigate what the implications are of our commitment to abstract, knowledge-level object-models for the other three central issues in building reflective systems that were discussed in the previous section.

Separated systems

Our commitment to object-models that abstract from implementation details of the object-system necessarily implies at least a conceptual separation between the object-system and its model at the meta-level: the object-system is an implemented system, and must thus be concerned with implementational mechanisms ("symbol-level aspects", in Newell's terminology), whereas the object-model at the meta-level is the abstract ("knowledge level") representation of the object-system. In section 3.3.1, we will argue that it is in fact possible to use the same symbolic structures for the dual purposes of object-system (symbol-level computation) and object-model (knowledge-level representation), but, at least conceptually, the two should be treated as separate. An additional advantage of this separation between object-system and its representation at the meta-level is that we avoid the paradoxes arising from self-referential constructs.

Causal connection must make non-syntactic distinctions

Our commitment to abstract ("knowledge-level") object-models means that we must depart from the standard approach to base the object-model only on syntactic distinctions in the object-system. As will be discussed in section, 3.1, we will want our object-model to capture distinctions between different types of knowledge used in the object-system. For instance, in a diagnostic system, we may want to distinguish abstraction-rules, causal-rules, generalisation-rules, etc, even though all these rules may be represented in the object-system using the same syntax. Besides syntactic distinctions, we will also want to capture semantic and pragmatic distinctions between elements of the object-system: different types of knowledge and how these are used.

However, even though the object-model should be based on such categories that are not necessarily syntactically distinguishable, the causal connection is required to make a link between such functional categories in the object-model on the one hand, and the implementational structures in the object-system that underly these categories on the other hand.

[vanHarmelen, 1992] describes how such non-syntactic distinctions can be captured by extending the usual notion of naming between object- and meta-theory.

Locus of action

Our approach to reflection does not commit us to specific ways of switching the locus of action between object- and meta-system. Section 3.3.2 will investigate the pro's and con's of various switching paradigms, but these comparisons are made on architectural and not on conceptual grounds.

Generalised reflective theories

A major advantage of our approach to reflection is that our meta-theories are applicable to a wider class of object-systems. Because our object-models only capture the functional behaviour of the object-systems, and abstract away from implementation details, the meta-theories based on these object-models will be applicable to any object-system that realises a certain functionality (say abductive diagnosis, or design), irrespective of how this functionality is implemented.

In section 3.2, we will argue that a reflective e task is like any other problem solving task, with the exception that its application domain is somewhat different (it is not a part of the external world, but instead another problem solver), an we will investigate the consequences of this.

In section 4, we will see that it is also possible to apply multiple reflective theories to a single object-system (say one reflective theory that does competence assessment, and another that does sensible explanation). Similarly, it will turn out to be possible to apply a single meta-theory to multiple object-systems, thus requiring multiple abstract models at the meta-level. We introduce the term n:1-reflection for the first arrangement, en 1:n-reflection for the second. Of course, the fully general m:n variation is also a possibility.

The remainder of this paper will be devoted to considering in greater detail each of the five issues that were treated in this section. Section 3.1 deals with the issues concerning the self-model, section 3.2 investigates the general nature of our meta-theories, section 3.3.1 discusses the implementational consequences of our standpoint on amalgamation vs. separation, and sections 3.3.2 and 3.3.3 investigate the implementational issues related to our choices on the causal connection and the switching paradigm.

3 Results

3.1 Model of the Object-Component

3.1.1 Using KADS for modelling the object-component

As pointed out the previous section, the nature of the model of the object-component is the prime discriminating factor for knowledge-level reflection. The position taken in the previous section on the abstract, implementation independent, knowledge-level nature of the object-model raises the question of how we should realise such object-models. For an answer to this question, we can look to some of the developments in the field of knowledge-engineering, where, although for different reasons, the construction of KBS-models with similar properties has received much attention in the past decade.

The particular knowledge-level models we use in our experiments are the models used in the KADS approach to knowledge engineering [Breuker & Wielinga, 1989, Wielinga *et*

al., 1992]. KADS models contain the following three types of ingredients:

1. Knowledge describing which *inferences* are needed in an application. Inferences describe the basic reasoning steps that one wants to make in some domain and the *roles* that pieces of domain knowledge that are manipulated by the inferences play in the overall reasoning process. The set of inferences is often graphically represented in a diagram showing the input-output dependencies between inferences: the so-called "inference structure".

2. Knowledge about the *structure of the domain-specific knowledge* required to perform inferences. The basic reasoning steps that make up the inference process will assume certain types of domain knowledge to be present.

3. *Control knowledge* which is used to determine how inferences are sequenced in a particular situation. The notion of a *task* is used to structure this control knowledge. A task defines a typical decomposition into inferences and/or sub-tasks together with internal sequencing information.

The different categories of knowledge have particular relations to each other: control knowledge *invokes* an inference; an inference *applies* domain knowledge.

An important property of KADS models is that they are independent of the specific details of the domain of the object-system. The model only contains object-domain independent knowledge about how the application task will be realised in the object system. For example, in the case of an object-system for diagnosing faults in an audio-system, the KADS model only contains knowledge about diagnosing such devices in general and does not contain knowledge specific for audio systems. In knowledge engineering this property is exploited by reusing (parts) of such a model in domains with similar characteristics, thus preventing the knowledge engineer from reinventing the wheel each time a new application is being built. For the type of reflective systems we are aiming at it means that the self-model is much more general than the specific system for which it has been constructed.

3.1.2 Languages for representing the self-model

To be able to use knowledge-level models as a self-model in a reflective system, it is necessary to have a formal and interpretable representation of such models. The KADS framework as presented in the literature [Breuker & Wielinga, 1989, Wielinga *et al.*, 1992] uses a highly structured, but still partly informal modelling language and can thus not be used directly for knowledge-level reflection. In REFLECT, two different languages were developed for solving this problem:

Formal specification through (ML)^2 (ML)^2 is a formal specification language for KADS models of expertise. This language offers a many-sorted logic, a module algebra for defining multiple theories, and a meta-level organisation using user-definable naming relations between theories.[2] For a detailed description of (ML)^2 the reader

[2] As remarked in the previous section, the internal structure of KADS models is also of a meta-like nature, in particular the relation between inferences and domain structures. This meta-object relation *within* a KADS model should not be confused with the meta-object relation between reflective system and object-system.

is referred to [Akkermans *et al.*, 1992, vanHarmelen & Balder, 1992]. An example $(ML)^2$ model of a problem solving method can be found in [Schreiber *et al.*, 1992].

Operational specification through Model-K Model-K [Karbach *et al.*, 1991] is an operational language for implementing KADS models. Model-K provides language constructs that retain to a large degree the information present in the KADS model, in addition to the implementation details. In this way, model-K ensures that the code of the final system can be used to access the knowledge-level model from which it was built. This information-preserving property of the object system is an important facilitating requirement for building the type of reflective systems we are aiming at. We will address this issue concerning the relation between self-model and object-system in more detail in the next section.

Besides opting for the use of KADS models to describe the object-system's model at the meta-level, we can of course also use the KADS framework to describe the meta-theory itself. After all, the meta-system is a KBS in its own right, be it one that has another KBS as its application domain. We can thus describe the meta-theory as a KADS model, containing the three ingredients of domain, inference and control knowledge as specified above, with the specific constraint that the KADS model of the object-system is part of the domain knowledge of the meta-system. This gives rise to a model of the meta-theory as depicted in figure 1.

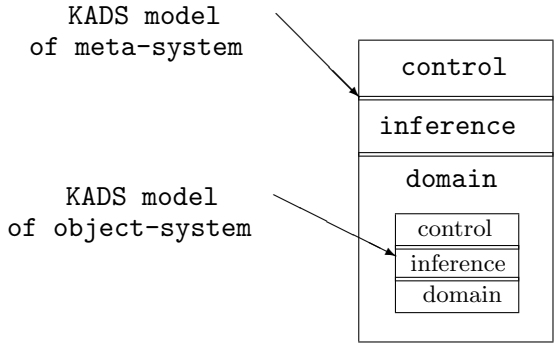


Figure 1: Use of KADS models in knowledge level reflection

3.2 The nature of reflective theories

The purpose of this section is to investigate the general nature of the contents of reflective problem solvers. Since reflective problem solvers perform many of the same tasks as ordinary problem solvers, we can expect that many of the general models that are available for ordinary problem solving will be equally applicable to reflective problem solvers. In this section, we will take the existing KADS model for a diagnose-repair task, and we will instantiate this model with the general types of knowledge required for a reflective theory that diagnoses malfunctions in its object-system.

3.2.1 Reflective Task and Inference Knowledge

We can ascribe various behaviors to the reflective component, for instance, predicting the effort of problem solving and the quality of the solution, recognizing impasses in the object problem solver or dynamically re-configuring the problem solver in cases of unsatisfactory behavior. Notwithstanding the variety, we can identify generic tasks that cover the specific instances of reflective reasoning. One way to model the reflective component is to regard it as consisting of a *diagnose* step where potential causes of incompetence are located and *repair* actions to overcome the incompetence. Although being slightly different, monitoring and control tasks can also be explained by this model.

Diagnose comprises inspecting resp. *analyzing* the object problem solver and *interpreting* the obtained *findings* in terms of potential *causes of incompetence*. Diagnosis is necessary for example, if an impasse in the object problem solver occurs. But even before an error actually occurs, an analysis of the object problem solvers can identify causes of a potential If the findings are compared to predefined norms, the interpretation step can expand to an entire assessment task or to a prediction task if the future behavior of the system is of interest.

Repair may be viewed as improving the competence depending on the outcome of diagnosis. The repair subtask consists of *proposing* repair actions and *applying* them to the object system. The repair step may be refined to an a priori configuration of the object system and the dynamic modification during problem solving. Usually, the *apply* subtask is the one that modifies the object system.

Figure **??** depicts the inference structure of the diagnose-repair model. Input to the *analysis* is the *object system model* or more precisely, an abstraction thereof. The analysis produces *findings* about the state of the object system, like a quantitative description of the complexity of the problem, the number of hypotheses or the set of competing actions. The findings are interpreted as *causes of incompetence* like "overcomplex problem" or "inconsistent state", but can also be the basis for predicting the behavior of the object system. The specific findings and causes will determine which repair actions are required to overcome the incompetence. Separating the proposal of repairs from their actual *application* allows to choose between incompatible proposals of different modules. Storing its own *experience* allows the module to *update* its knowledge and use it for analyzing and proposing repairs in the future. For instance, a reflective module decomposing a problem can accumulate the solutions of already solved subproblems as experiences.

figure missing:

An inference structure for reflective competence assessment and improvement

In the inference structure for diagnose-repair tasks of figure **??**, the "malfunctions" play a crucial role. Above we suggested that essentially six categories of malfunctions can be responsible for an object-system's unsatisfactory behaviour on a given problem: incompleteness, uncertainty, inconsistency, overcomplexity, irrelevancy and redundancy. – Further categories like sufficiency of computing resources or the possibility to parse are already situated at the implementation level.

3.2.2 Reflective Domain Knowledge

In contrast with the other two layers of the model, this layer is mostly independent of the choice of task for the reflective system. (i.e. diagnose-repair in our case). It is, instead, more dependent than the other two layers on the fact that we are studying a reflective system, which reasons about another problem solver. This is to be expected, since the domain layer is supposed to capture the properties of the domain of reasoning, which, in the reflective case, is another problem solver.

Domain knowledge of reflective systems consist of two parts: First there is the model of the object-system, which has been discussed in section 3.1. Beside the model of the object system there may be additional knowledge enabling the reflective component to perform its task. To a large extend this knowledge embodies the expertise of knowledge engineers and programmers evaluating their programs. The specific type of tasks performed determines the kind of knowledge to be expected at the reflective domain layer. General categories of knowledge that can be distinguished at the domain layer are:

- general knowledge about problem solving and reasoning
- knowledge abou the specific object task
- additional knowledge about the object domain
- additional knowledge about the object system realising this task in this domain

3.3 A space of architectures for knowledge level reflection

The preceding sections have all been concerned with the conceptual organisation of systems that perform knowledge-level reflection, but have barely touched upon the issue of how to implement such systems. That will be the topic of this section. We will lay out a *design space* for the actual construction of the reflective component and its connection to the object system. The dimensions of this design space follow from the specific nature of a reflective system and will concern (i) how to realise the model of the object-system, (ii) how to switch activity between object- and meta-system, and (iii) how to ensure a correspondence between the object-system and its model at the meta-layer.

Before discussing each of these three design dimensions in the next three subsections, we discuss the possibility of not implementing a reflective specification as a two-level system at all.

Single level systems In principle, every reflective specification can be realised within a single-level system: there is no inherent property of the reflective specification that makes it impossible to integrate it with the object-system. The major reason for choosing the single-level option is computational efficiency. However, there can be practical reasons for leaving the object system intact. Often it is difficult and sometimes it is impossible to modify the existing code of the object system. Even more important is that opting for a multi-level architecture offers a number of advantages: a high degree of conceptual clarity and modularity. The standard problems created through non-modularity are only amplified in the case of the complex systems investigated here. Three system development objectives that require a high level of conceptual clarity and modularity and thus lead to

favoring a multi-level solution above a single-level one are reusability of reflective modules, explanation and maintainability.

3.3.1 Realising a model of the object system

In the preceding sections, we extensively discussed the abstract nature of the object-model that the meta-level has of the object-model. It contains structural knowledge such as task-decompositions, primitive inferences and knowledge-roles. The emphasis for the model of the object system is on *describing* the knowledge structures, rather than *using* them as in the object system itself.

In this section we consider three alternatives for realisation of such an abstract model of the object system in a reflective architecture.

Access procedures In this approach a direct connection between the reflective inference layer and the language-structures of the object system is established by means of a set of procedures of read and write access. No separate meta-representation of the object-system is employed. The read procedures fill or update the contents of the reflective knowledge roles. The write procedures directly modify the language structures of the object system.

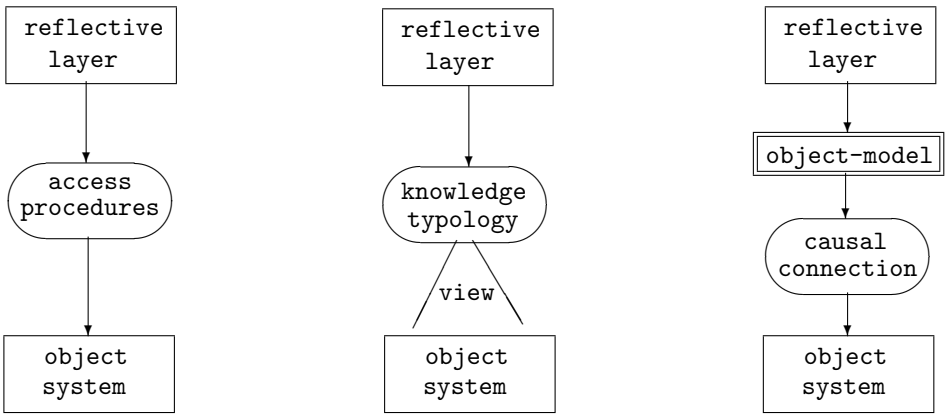
Knowledge typing The second approach introduces a facility for knowledge typing, i.e., a declaration of the conceptual types of knowledge in the object system. The knowledge typing mechanism serves as a *view* on the object system: it attaches knowledge types to language structures of the object-system and limits access to those parts of the object system for which this declaration is explicitly made. Again, in this case, no separate meta-representation of the object-system is employed.

Separate representation In the third approach the model of the object system is represented as a separate data structure and in a language that is different from the language of the object system. It results in an implementation of a reflective system that corresponds exactly to the conceptual description of such systems in the preceding sections. This approach requires a causal connection that ensures the integrity and force of the representation. (see Section 2.1.2). Because the language structures of the object system are not directly manipulated by the reflective system, it is possible to delay the synchronisation of representation and object system. This allows for multiple extensions and modifications of the self-model to be investigated by the reflective system at the same time, for example the comparison of alternative applicable methods in a reflective system that schedules object problem solving steps.

These three options for realising the object-model are summarised in figure 2.

Guidelines

Access procedures can only realise an abstract model of the object system in an ad hoc manner. A knowledge typing mechanism constitutes the model in a principled manner either as an integral part of the language of the object system or by imposing a suitable view on the object-system. The knowledge typing mechanism is a view on the object system in the sense that modifications to the model are directly effectuated in the object



(a) Access procedures (b) knowledge typing (c) separate representation

Figure 2: Different ways of realising the object-model

system. This is suitable for reflective tasks that react to the current situation by immediate actions, but would result in inconsistency of the object system when alternative extensions are considered by the reflective task. The third approach of using a separate representation allows for alternative extensions of the model. In combination with a knowledge typing mechanism it realises a self model in a manner that fully corresponds with the conceptual framework of the preceding sections. Another advantage is that the languages of object system and representation are disjunct, clarifying the distinction between mention (at the reflective level) and use (at the object level). The major drawback of the separate representation is that it requires a specification of the self representation and introduces the computational complexity of synchronisation as will be discussed in the next section.

3.3.2 Switching Paradigm

The conceptual view on reflection outlined in the previous sections presents an architectural problem, in that it represents two "active agents": the object-system and the meta-system. We have to implement a distribution of activity between these two parts of a reflective system.

This switching paradigm constitutes the second dimension of our design space for reflective systems. In [vanHarmelen, 1991] we laid out a number of design options for switching activity between object- and meta-layer which we will re-investigate in the context of the notion of knowledge-level reflection.

Preferred switching paradigms

An obvious choice for switching activity between object- and meta-layers is *subtask-management*: reflective knowledge is used to schedule the subtasks that make up the object-level. We have used subtask-management successfully as the switching paradigm in a number of experiments (section 4).

A rather different option (not occurring in [vanHarmelen, 1991]) is so-called *external*

switching. The idea here is to introduce a third module that is responsible for switching activity between the object- and reflective layers. We call such a third module a *scheduler*. It is a meta-module for both the object- and the reflective layer, and determines when either of these two should be active. The relation between the scheduler and the other two systems should then be one of subtask management: the scheduler gives subtasks to either the object- or the reflective layer. As with subtask-management, we have used this switching paradigm successfully in a number of experiments (section 4).

Inadequate switching paradigms

A design option for the switching paradigm that is explicitly advocated in [vanHarmelen, 1991] is *meta-simulation*. This ssumes that there will only be activity at the meta-level, and requires a complete description of the object-system in the object-model. This description can then be used to simulate the object-system at the meta-level.

However, it turns out that this option is unsuitable for realising the kind of reflective systems investigated in this paper. The major assumption underlying meta-simulation is that the representation of the object-system in the reflective layer is detailed enough to allow full simulation of the object-system by means of this representation. This puts very stringent requirements on the contents and amount of detail of the object-model: this representation must be detailed enough to allow a full simulation of the object-system. This requirement is in conflict with the major premise of the notion of "*knowledge level* reflection", as defined above which states that the object-model abstracts from the computational aspects of the object-system. It would seem unlikely that such an abstract object-model would enable a simulation of the object-system.

Adequate switching paradigms

There are two further options for the switching paradigm in our design space. Both these options would be adequate choices, although both suffer from the problem that they require some reflective knowledge as part of the object-system, instead of having all reflective knowledge at the reflective layer.

For so-called *crisis-management systems*, a switch of the locus of action takes place whenever a *crisis* occurs in the object-level computation. After the meta-system has resolved this crisis, the object-system can proceed with the computation. Examples of definitions of a crisis are when no or too much object-level knowledge is applicable, when the object-level has run out of resources, or when the object-level needs more information. The major drawback of this paradigm is that the definition of crisis needs to be inserted into the object-system while it conceptually belongs at the meta-level

Reflect-and-act systems are the final option along this dimensions of our design space. In reflect-and-act systems, the locus of action moves from object- to reflective layer whenever a predefined place in the object-computation has been reached. Two objections that can be raised against this option are that switching only happens at predefined, fixed places in the object-system's cod and the switching need to be defined in the object-system, rather than at the meta-level.

3.3.3 Synchronising object-model and object-system

If, among the design option for realising the self-model outlined in section 3.3.1 we would choose for an explicit representation of the object-model which is separate from the object-system itself, we must somehow ensure both integrity and force of this model with respect to the system it represents.

This section discusses a number of possible ways to realise this synchronisation, which can be distinguished on the basis of when the synchronisation takes place on how much information is copied between the object-system and its model.

When to synchronise

Eager synchronisation The most obvious option is to ensure that the two descriptions of the object-system are always synchronised: changes are immediately propagated. This type of synchronisation is required for the monitoring type of switching discussed above, but is technologically hard to realise efficiently.

Lazy synchronisation The opposite of eager synchronisation is the arrangement where changes in either representation are allowed to happen without propagation to the other representation: only when a read operation is performed on one of the representations, an update takes place. The advantage of this scheme is of course that it minimises the amount of synchronisation that is performed by the system.

Synchronise on switching A third option is to exploit the control switching paradigm that has been chosen. In particular, if a control paradigm has been chosen that involves an explicit switch of the locus of action (i.e. subtask-management, crisis-management or reflect-and-act), synchronisation can take place every time the locus of control is moved. The justification for this is of course that in between such switches, only one of the two representations is modified and read, so that we can delay the synchronisation until the point of switching. Again, this arrangement avoids some unnecessary updates, but not as many as lazy synchronisation.

How much to copy

A further option dimension in the design space for the causal connection is how much of a representation should be copied in the process of synchronisation.

Full copying The easiest option is of course to make a full copy of one representation, and translate it entirely into the other representation. Although conceptually very simple, this option is likely to lead to efficiency problems when both representations are very large and complicated.

Incremental copying - what is changed A much more efficient, but rather more complicated option is to copy only those parts of a representation that have been actually modified since the last synchronisation. This requires keeping track of modification times of parts of the different models. This option further requires that there is "structural correspondence" between the two representations of the object-problem solver, so that the

elements in one representation can be linked to the corresponding elements in the other representation. This is quite a strong requirement, which is not imposed by the "full copying" option from the previous paragraph.

Incremental copying - what is required A final option is to copy not all datastructures that have changed, but all structures that are needed (irrespective of whether they have been changed or not). This no longer requires timestamping datastructures, but possibly copies too many data. It is of course also possible to combine this option with the previous one to absolutely minimise the amount of data that is copied (at the expense of complicating the implementation of the causal connection).

4 Validation

To validate our approach we built several reflective systems exploring a broad conceptual and architectural spectrum. This section gives a survey of all experiments and presents some in greater detail. An exhaustive description can be found in [Bartsch-Spörl *et al.*, 1991].

4.1 A reflective assignment system

4.1.1 The object component

Our object was an assignment system that allocates components to so-called slots while satisfying certain requirements. We called it OFFICE-PLAN as we actually used it to allocate employees to office rooms. OFFICE-PLAN was rather incompetent. It tried to solve even obviously inconsistent problems, for instance when there were more employees than room for them in the offices. When it finally detected that there was no solution, it could not propose any compromises. It did not recognize that a problem was overcomplex so that it would need hours to solve it. In particular, it did not detect redundancies. For instance a person that must sit in a single room trivially satisfies any different-room constraints so that they may be dropped. It could not cope with underspecified problems, e.g by evaluating more restrictive constraints first. From the five types of malfunctions mentioned in section 3.2, uncertain knowledge was the only one that caused no problems, because the system operated with categorical knowledge.

To tackle these four types of malfunctions – overcomplex, undercomplex, incomplete, and irrelevant knowledge – we built and integrated ten meta-components on top of OFFICE-PLAN. They are generic so that they do not depend on the particular domain of office planning. To a certain extent, they are neither specific to the particular problem solving technique employed in the object component.

4.1.2 The ten meta-components and their integration

Roughly spoken, we have meta-components for feasibility studies, for resource management, and for problem modifications. The latter do simplifications, relaxations, or eliminate contradictions.

Feasibility studies are performed by comparing available and required resources. With this knowledge, we can prevent the object component from solving overcomplex or "apparently" unsolvable problems.

Contradictions are detected both wrt. the requirements in the original problem statement and wrt. the constraints in the internal representation, and relaxations are negotiated with the user.

Redundancies of four different types can be detected and removed: between pairs of requirements, pairs of constraints, between constraints and conditions, and between constraints and the possible value restriction of constrained variables. Removing these redundancies speeds up the solution process.

Decomposition and relaxation: One meta-component decomposes overcomplex problems and solves them stepwise, as much as possibly reusing previously found partial solutions. By suitably composing solvable subproblems it can return approximate solutions to overspecified problems. The component is rather quick, and its runtime increases only linearly with the number of subproblems stored in its library. Thus, problem solving time is substantially reduced. In contrast to feasibility studies, which only detect certain overspecifications, and in contrast to contradiction removal, which is specialized to pairwise inconsistencies, this meta-component will detect and solve any inconsistencies.

Resource limitations are introduced by limiting the time and the number of solutions desired. To satisfy them, this meta-component controls the generate-and-test kernel of OFFICE-PLAN by iteratively allocating time slices to the "generate" and "test" steps. Its purpose is to achieve optimal performance under the given limitations. Thus, we could specify the number of solutions we would be content to see, or we could specify the time we are willing to wait and the system will do its best to produce many solutions in that time, or even combine both.

Case-based reasoning allows to bypass the entire problem solving process by exploiting a library of complete cases. By incorporating problems with relaxed solutions, this meta-component can cope with inconsistent problems as well. However, it offers a quick but risky approach, since the case-library may not contain a similar case. Instead of including the case-library in the meta-component, we could have used a case-based reasoner as a second object-component. The meta-component would then only have to decide whether to invoke the case-based reasoner or OFFICE-PLAN.

In the reflective system, the functionality of the object component was extended, as inconsistent problems can now be solved by making compromises. Additionally, system utility was enhanced. The original system took an input problem, then fell silent for quite a while to finally come up with all solutions, i.e. none if the problem was inconsistent. In the reflective system, we can choose the number of solutions we want, the time we are willing to spend, and we can have relaxations. Figure 3 shows the three dimensions of utility and the values we can assign to each parameter. However, not all combinations can be served by the meta-components. For instance, you cannot have relaxed solutions or all solutions within a prespecified amount of time. To serve such impossible combinations we

introduced priorities: Relaxation is more important than time which is more important than the number of solutions. For instance, we cannot produce relaxed solutions within predefined time limits. In this case, we relax and ignore the time.

utility dimensions	values	OFFICE-PLAN	CASY	COMIC	TACKLE-TIME
relaxation	yes no	no	yes	yes	no
time	quick [min, max] nomatter	long	quick but risky	often quick, sometimes slow	[min,max]
no of solutions	few [min, max] all	all	all or none	all	[min,max]

Figure 3: Utility dimensions and how the specialists fit in

4.2 A spectrum of reflective systems

Altogether we developed 13 meta-components operating on three different object systems, and combined them into reflective systems varying in the number of meta- and object components. Beside the reflective OFFICE-PLAN system,

- we have a 1:1[3] reflective system [Bartsch-Spörl *et al.*, 1991; chapter 3] dealing with a bug in a qualitative reasoner. Under certain circumstances, the system computes multiple instances of a parameter resulting in different states which should actually be identified. The bug can be repaired in three ways. We can change the library of qualitative models before prediction which may be rather counter-intuitive. Or we can fix the bug in the code of the system, which would require a very intimate knowledge of some 10000 lines of Prolog code and probably a lot of debugging. So we built a meta-component that spots the multiple parameter instances, identifies them and merges the corresponding states.

- In another 1:1 reflective system [Bartsch-Spörl *et al.*, 1991; chapter 2] we deal with the problem of multiple diagnoses in abductive reasoners. Instead of trying to explain all symptoms by a multiple-fault diagnosis, we seek observations that explain most of the symptoms and relax the others. This is done by computing the specificity of the observations from the causal network and by considering the applicability of the hypotheses for the currently observed symptoms. Other than in multiple fault diagnoses, this approach allows to take into account dependencies between symptoms and physiological interactions between diseases.

- A 1:2 reflective system [Bartsch-Spörl *et al.*, 1991; chapter 4] combines the two object components already mentioned, the abductive reasoner and the qualitative one. If abductive diagnosis fails to produce an explanation or if it cannot differentiate between two hypotheses, causal connections are missing, which could be retrieved by qualitative simulation. We built a meta-component that decides which object component to invoke in which situation, and which controls and possibly repairs this initial strategy.

[3]We use the $n : m$ notation to indicate a system with n meta-modules and m object-modules.

With this collection of 1:1, 1:2, and n:1 reflective systems we covered a broad scope, including reflective diagnose&repair tasks, reflective resource management, and reflective strategy construction. We tackled four different malfunctions, inconsistencies, redundancies, incompleteness, and overcomplexity. Only uncertainty was not treated since it played no role in any of our object components. Our meta-components realize very different reflective behaviors in that they inspect or modify different kinds of knowledge in the object components. Table 1 summarizes the spectrum covered by our reflective systems.

dimension	options explored
type of reflective system	1:1, 2:1, n:1
type of reflective task	diagnose&repair, strategy construction, resource management
malfunctions	inconsistency, irrelevancy, incompleteness, overcomplexity
general inference structure	exact match, renamed, simplified, extended

Table 1: Spectrum covered by our experiments.

4.3 Architectural alternatives explored

To develop the reflective assignment system, we proceeded incrementally. Each meta-component was built so as to constitute –together with the object component – a 1:1 reflective system. As soon as a number of interesting meta-components was available, we integrated them into an n:1 system. We started by integrating the redundancy removing components, then added contradiction detection and feasibility studies, then resource handling, then decomposition and relaxation, and finally the case-based reasoner. To support this implementation technique, we had to modularize any aspects concerning the scheduling of the meta-components and their coordination with the object component. For this purpose, we introduced a scheduling layer on top of the task layers of the meta-components. Since control between different components is exercised at this layer, the suitable switching paradigm was the external subtask management regime.

Instead of implementing the ten meta-components in an general purpose language like LISP or Prolog, we developed a language that directly supports the REFLECT-approach. This language, called MODEL-K, represents one point in the design space laid out in the previous section. Any meta-components defined in this language submit to the same design decisions. This is the reason why in table 2 it is sufficient to juxtapose the options explored in the other experiments with the design decisions for MODEL-K. We will shortly comment the entries.

Representation and implementation languages: In the office allocation system, we used MODEL-K to model and to implement, both, the object and the meta-components. In the other experiments, the object and meta-components were modeled in $(\text{ML})^2$, a logic based language for formalizing KADS models [vanHarmelen & Balder, 1992], and implemented in Prolog.

Scope of model represented: In MODEL-K, the object system is an extension of its conceptual model by any operational aspects. Thus, we had the complete model

architectural dimension	options explored	
	in other experiments	in MODEL-K experiments
specification language	ML	MODEL-K
implementation language	Prolog	MODEL-K
switching paradigm	subtask management	external
object model:		
scope of representation	partial	complete
access	via representation, direct	transparent
synchronization:		
direction	up, down	up, down
scope	full	incremental
time	at switch	lazy (up), eager (down)

Table 2: Architectural options explored

available in the implementation. For object components modeled in (ML)^2, we explored two alternatives. We either represented only the parts actually being accessed by the meta-component, or we dropped the model completely, because the corresponding system was already written in Prolog in a structure preserving way.

Access: MODEL-K only supports a transparent access to the object model. In the other cases, we either accessed the explicit Prolog representation of the model, or directly the Prolog code of the object system.

Switching paradigm: For the 1:1 and 1:2 systems we used subtask management as the most straightforward alternative. For the n:1 systems we used the external switching facility built into MODEL-K.

Synchronization: Few of our reflective systems require only a synchronization upwards, because the object-component is not modified. Most require a synchronization upwards and downwards. We used full and incremental updates, and effected them eagerly, lazily, or at switching times. In MODEL-K, the update upwards is done when data are required and done downwards eagerly.

With our experiments we actually explored some quite different points in the design space defined in section 3.3. In particular, without the 10:1 experiment in office planning we probably would not have noticed the need for an external scheduler. We conclude that there is no unique set of design decisions, the guidelines we gave in that section were derived from our experiments.

Beside these differences, we made some common experiences. All meta-components somehow fit the inference structure shown in section 3.2. Thus, it should provide a good starting point for modeling a meta-component. All our meta-components operate on a very partial model of their object system. This may mean that the components are one on the one hand quite general, on the other hand not to complicated. Although we explored very different types of implemented the object model, realizing the causal connection was always straightforward. Even more, we never had to modify the object systems. This is very strong evidence for our approach of operating on knowledge level models of the object components - provided they are implemented in a structure preserving way.

5 Conclusions

5.1 The nature of reflection

Starting point of the REFLECT project was the notion of reflection as reasoning about one's own knowledge and capabilities. Through a study of the literature on reflection and a systematic investigation of the conceptual space spanned up by the various forms of reflective systems, we have identified a number of key issues that the designer of a reflective system must decide upon.

From our study of the literature on reflection it becomes clear that there exist a wide variety of conceptualisations of what reflection is. Despite the differences, the various approaches to reflection point to a number of central notions. First of all, there is the concept of a *self model*. The second common idea that we find in the various approaches is that of a separation between the object and meta system as a means of coping with the problems of self-referentiality, circularity and inconsistency.

Based on the observations sketched above, we have developed in the REFLECT project a new approach to meta-level and reflective systems. This approach is based on a multi-layered architecture where a sub-system located in one layer reasons about and operates upon an *abstract model* of a sub-sustem represented in another layer. This approach reflects two decisions. First, there is a clear separation of meta and object level. The main rationale behind this choice is to circumvent the problems related to self-referentiality. As a consequence the systems that we have investigated are not *fully* reflective in the sense that they do not have a full self-model but only a partial one.

A second choice that was made in the REFLECT project concerns the nature of the model of the object system. Rather than to choose a syntactic or computationally oriented representation of the object system, we have opted for a representation that is independent of the implementational details of the object system and that is *meaningful*, i.e. is expressed in an ontology that reflects the purpose of the reflective reasoning. The approach taken in REFLECT is very much in line with modern approaches to analyse and describe knowledge-based systems. Here too, a consensus is emerging that in order to understand knowledge-based systems, to re-use knowledge and to make more flexible systems, a model of the knowledge and reasoning processes is necessary that abstracts from implementation details and focusses on the knowledge content of a system [Wielinga *et al.*, 1992]. Newell [Newell, 1982] has introduced the term *knowledge level* for abstract, implementation independent descriptions of AI systems Hence, we use the term knowledge level reflection for reasoning about problem solving using abstract models.

5.2 The nature of the reflective task and associated meta-theories

Reflection in its common-sense meaning defines the object of the reasoning, i.e. the self, but not the purpose of reflection. In our study of reflection we have discovered that there are many different goals that reflective reasoning may serve. A common example of a reflective task is to *control* the reasoning in the object system through meta-level reasoning. This task has been extensively studied in the literature [vanHarmelen, 1991], but other types of reflective tasks have not received much attention until very recently. In the REFLECT project we have studied both reflective tasks concerned with controlling the object problem solver (e.g. *strategy planning* and reasoning under limited resources)

and other, non-control related reflective tasks. The latter include *competence assessment* (estimate whether a system can solve a given problem in principle, and how difficult this will be), *quality assessment* of the solutions produced by the object system, and *combining* multiple problem solvers. We have found that many of the reflective tasks that have been studied in prototype implementations have a similar structure. First an analysis is made of the current state of the problem solving process. If anything is not according to the expectations that the reflective component has, the discrepency is analysed and measures are proposed to remedy the problem. In order to perform such reasoning, the reflective component needs to be equipped with a meta-theory of the problem solving processes that take place in the object problem solver. A crucial role in such meta-theories is played by knowledge of deficiencies that can occur in the object problem solver. Five categories of such deficiencies have been identified: incomplete information, uncertain information, inconsistent information, overcomplex information and irrelevant information.

Although the meta-theories that were used in the various experiments are specific for the type of reflective task and to some extent dependent on the specifics of the application domain of the object problem solver, some general patterns begin to emerge in these meta-theories. Meta-theories for reflection are about problem solving and as such they contain general knowledge about problems, search spaces, solvability criteria, execution time etc. For example the meta-theory that is used in the resource-limited reasoning task is sufficiently general that it can be used for any system that is based on a generate-and-test method.

One might argue that the level of abstraction that is used for modelling the object problem solver may be too high in order to achieve interesting reflective behaviour. We have illustrated through a number of examples that the use of abstracted models of the object system still allows to formulate interesting and useful meta-theories. Additionally, the abstract and hence implementation-independent nature of the models serves as a basis for reusable meta-theories: meta-theories can be reused for different object systems which can be modelled by the same abstract model.

5.3 Architectural options

Given the conceptual approach that was developed in REFLECT, a number of architectural options still remain open. First there are several ways in which the model of the object system and the causal connection can be realised: a separate representational structure can be used, special access procedures can be defined which inspect and modify the object system directly and knowledge typing providing a certain view on the object system, can be used to access the object knowledge. The separate representation option, possibly in combination with the knowledge typing, appears the most flexible one, but has as a drawback that maintaining the consistency between the model and the actual object problem solver requires a complex synchronisation.

The *switching paradigm* is a second important choice point for the designer of a reflective system. The switching paradigm defines the distribution of activity between the object and meta component. A number of options were found to be more or less suitable, while others were not.

A third architectural problem is that of the synchronisation between the model of the object system and the actual object problem solver. We have identified three dif-

ferent strategies for synchronising: eager synchronisation (update whenever something is changed), lazy (update when a piece of information is needed) and on switching. The choice between these alternatives is a trade-off between efficiency and complexity.

Several of the architectural options were explored and tested in various prototype systems. From these experiments we derived the judments about usefulness, adequacy and feasibility of the various options. Choices with respect to some of the options will depend on taste, style and requirements of the reflective task.

5.4 Technical Advances

In addition to the conceptual and theoretical advances made in the project, there are some technical advances that the use of reflection offers. The first one is that of *evolutionary development of knowledge based systems*. The REFLECT approach allows to enhance the capabilities of an existing system by adding a reflective component equipped with an abstract model of the object system. Both the experiments with the office-plan system and with the GARP system show that little or no modification of the object system is necessary when the meta-component is added. The abstract model needed is often already available as a specification of the system, e.g. in the form of KADS models.

A second advance of the REFLECT approach is that of combinability of several object problem solvers without having to re-engineer these systems. For example, as was demonstrated in one of the experiments, several existing diagnostic systems can be combined into a much more powerful system by adding a reflective module, without the need to modify the individual object components. A second aspect of combinability is the possibility to add several additional capabilities, such as competence assessment, reasoning under limited resources and flexible control, as separate reflective modules to an existing KBS. The interactions between these reflecvtive components can then be controlled by a straightforward scheduling module.

A third advance is that of reusability. reflective modules implement general capabilities that are useful for a range of knowledge based systems. Through the approach taken in REFLECT, where the meta-theory is separated from the model of the object system, one may expect such modules to be reusable. For example modules implementing explanation, flexible control or competence assessment can be made largely independent of the nature of the object system.

In summary, we conclude that the approach developed in REFLECT has significant application potential in the field of knowledge-based systems. One could argue that many of the capabilities that we have described above, can be implemented in the object systems themsleves, and indeed they can. However, it is precisely the architectural framework developed in REFLECT that provides the modularity and combinability of software that is so much needed, but so seldom achieved.

5.5 Unresolved issues and future work

The REFLECT project has laid the foundations of a methodology for building modular, reusable and flexible knowledge-based systems. However, a number of issues remain to be studied. An important limitation of the results of the project is their scope. We will need proof that the approach tested in the various prototypes will scale up to real

life applications. Even though some of the experiments used object problem solvers of a significant size (e.g. GARP) more complex reflective systems need to be built.

A second area of further study is the nature of the meta-theories. As said above, some general patterns in the meta-theories begin to emerge, but much more research is needed to establish a general ontology and meta-theory of problem solving that can be a basis for the development of meta-theoretical extentions for specific reflective tasks. Most of the experiments that were performed in the project concern assessment and modification of the dynamic structures involved in solving a problem. Reasoning about the scope and capabilities of the static knowledge in order to assess the general competence of a problem solver is a very difficult problem. Recent insights in the theoretical foundations of diagnostic reasoning give however some hope that meta-theories can be developed that generate advanced reflective behaviours for certain classes of object systems.

References

[Akkermans *et al.*, 1992] J. M. Akkermans, F. van Harmelen, A. Th. Schreiber, and B. J. Wielinga. A formalisation of knowledge-level models for knowledge acquistion. *International Journal of Intelligent Systems*, 1992. forthcoming.

[Bartsch-Spörl *et al.*, 1991] B. Bartsch-Spörl, B. Bredeweg, C. Coulon, U. Drouven, F. van Harmelen, W. Karbach, M. Reinders, E. Vinkhuyzen, and A. Voß. Studies and experiments with reflective problem solvers. ESPRIT Basic Research Action P3178 REFLECT, Report IR.3.1,2 RFL/BSR-UvA/II.2/1, REFLECT Consortium, August 1991.

[Bowen & Kowalski, 1982] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 152–172. Academic Press, 1982.

[Breuker & Wielinga, 1989] J. A. Breuker and B. J. Wielinga. Model Driven Knowledge Acquisition. In P. Guida and G. Tasso, editors, *Topics in the Design of Expert Systems*, pages 265–296, Amsterdam, 1989. North Holland.

[Clancey, 1983] W. J. Clancey. From Guidon to Neomycin and Heracles in twenty short lessons. *The AI Magazine*, 7(3):40–61, August 1983.

[Davis, 1980] R. Davis. Metarules: Reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[Feferman, 1962] S. Feferman. Transfinite Recursive Progressions of Axiomatic Theories. *Journal of Symbolic Logic*, 27(3):259–316, September 1962.

[Giunchiglia & Serafini, 1990] F. Giunchiglia and L. Serafini. Multilanguage first order theories of propositional attitudes. Technical Report 9001-02, IRST, Trento, Italy, January 1990.

[Giunchiglia & Smaill, 1988] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming (META88)*, pages 123–140, Bristol, June 1988. MIT Press. Also: DAI Research Paper 375, Dept. of Artificial Intelligence, Edinburgh.

[Godel, 1931] K. Godel. Uber formal unentscheidbare satze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–98, 1931. English translation in *From Frege to Godel: a source book in Mathematical Logic, 1879-1931*, J. van Heijenoort (ed.), Harvard University Press, 1967, Cambridge, Mass.

[Hill & Lloyd, 1991] P.M. Hill and J.W. Lloyd. The Gödel report (preliminary version). Technical Report TR-91-02, Computer Science Department, University of Bristol, March (Revised September '91) 1991.

[Karbach *et al.*, 1991] W. Karbach, A. Voß, R. Schukey, and U. Drouwen. Model-K: Prototyping at the knowledge level. In *Proceedings Expert Systems–91, Avignon, France*, pages 501–512, 1991.

[Kripke, 1975] S. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 13(72), 1975.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Maes & Nardi, 1988] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*, Amsterdam, 1988. North-Holland.

[Maes, 1987] P. Maes. Computational reflection. Technical report 87-2, Free University of Brussels, AI Lab, 1987.

[Newell, 1982] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[Perlis, 1985] D. Perlis. Languages with self-reference I: Foundations. *Artificial Intelligence*, 25:301–322, 1985.

[Rosenbloom *et al.*, 1988] P. Rosenbloom, J. Laird, and A. Newell. Meta-levels in SOAR. In *Meta-Level Architectures and Reflection*, pages 227–240, Amsterdam, 1988. North-Holland.

[Schreiber *et al.*, 1992] A. Th. Schreiber, B. J. Wielinga, and J. M. Akkermans. Differentiating problem solving methods. In Th. Wetter, K-D. Althoff, J. Boose, B. Gaines, M. Linster, and F. Schmalhofer, editors, *Current Developments in Knowledge Acquisition - EKAW'92*, pages 95–111, Berlin/Heidelberg, 1992. Springer Verlag.

[Smith, 1982] B. C. Smith. Reflection and semantics in a procedural language. Technical Report TR–272, MIT, Computer Science Lab., Cambridge, Massachussetts, 1982.

[Smith, 1984] B. C. Smith. Reflection and semantics in Lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, 1984. also: Xerox PARC Intelligent Systems Laboratory Technical Report ISL–5.

[Tarski, 1936] A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936. English translation in *Logic, Semantics, Metamathematics*, A. Tarski, Oxford University Press, 1956.

[Turner, 1990] R. Turner. *Truth and Modality for Knowledge Representation.* Pitman, London, 1990.

[vanHarmelen & Balder, 1992] F. van Harmelen and J. R. Balder. $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1), 1992. Special issue: 'The KADS approach to knowledge engineering'.

[vanHarmelen, 1991] F. van Harmelen. *Meta-level Inference Systems.* Research Notes in AI. Pitmann, Morgan Kaufmann, London, San Mateo California, 1991.

[vanHarmelen, 1992] F. van Harmelen. Definable naming relations in meta-level systems. In A. Pettorossi, editor, *Proceedings of the Third Workshop on Meta-programming in Logic (META'92)*, Uppsala, June 1992. Springer Verlag.

[Weyhrauch, 1980] R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13, 1980. Also in: *Readings in Artificial Intelligence*, Webber, B. L. and Nilsson, N. J. (eds.), Tioga publishing, Palo Alto, CA, 1981, pp. 173-191. Also in: *Readings in Knowledge Representation*, Brachman, R. J. and Levesque, H. J. (eds.), Morgan Kaufman, California, 1985, pp. 309-328.

[Wielinga *et al.*, 1992] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1), 1992. Special issue 'The KADS approach to knowledge engineering'.