Meta-reasoning: a Survey

Stefania Costantini

Dipartimento di Informatica
Università degli Studi di L'Aquila,
via Vetoio Loc. Coppito, I-67100 L'Aquila, Italy
stefcost@univaq.it

Abstract. We present the basic principles and possible applications of systems capable of meta-reasoning and reflection. After a discussion of the seminal approaches, we outline our own perception of the state of the art, mainly but not only in computational logic and logic programming. We review relevant successful applications of meta-reasoning, and the basic underlying semantic principles.

1 Introduction

The meaning of the term "meta-reasoning" is "reasoning about reasoning". In a computer system, this means that the system is able to reason about its own operation. This is different from performing object-level reasoning, which refers in some way to entities external to the system. A system capable of meta-reasoning may be able to reflect, or introspect, i.e. to shift from meta-reasoning to object-level reasoning and vice versa.

We present the main principles and the possible applications of meta-reasoning and reflective systems. After a review of the relevant approaches, mainly in computational logic and logic programming, we discuss the state of the art and recent interesting applications of meta-reasoning. Finally, we briefly summarize the semantic foundations of meta-reasoning. We necessarily express our own partial point of view on the field and provide the references that we consider the most important.

There are previous good reviews on this subject, to which we are indebted and to which we refer the reader for a wider perspective and a careful discussion of problems, foundations, languages, approaches, and systems. We especially mention [1], [2], [3]. Also, the reader may refer, for the computational logic aspects, to the Proceedings of the Workshops on Meta-Programming in Logic [4], [5], [6], [7], [8]. Much significant work on Meta-Programming was carried out in the Esprit funded European projects Compulog I and II. Some of the results of this work are discussed in the following sections. For a wider report we refer the reader to [9].

More generally, about meta-reasoning in various kinds of paradigms, including object-oriented, functional and imperative languages, the reader may refer to [10] [11], [12].

Research about meta-reasoning and reflection in computer science has its roots in principles and techniques developed in logic, since the fundamental work of Gödel and Tarski, for which it may be useful to refer to the surveys [13], [14]. In meta-level approaches, knowledge about knowledge is represented by admitting sentences to be arguments of other sentences, without abandoning the framework of first-order logic.

An alternative important approach to formalize knowledge about knowledge is the modal approach that has initially been developed by logicians and philosophers and then has received a great deal of attention in the field of Artificial Intelligence. It aims at formalizing knowledge by a logic language augmented by a modal operator, interpreted as knowledge or belief. Thus, sentences can be expressed to represent properties of knowledge (or belief). The most common modal systems adopt a *possible world* semantics [15]. In this semantics, knowledge and belief are regarded as propositions specifying the relationship between knowledge expressed in the theory and the external world. For a review of modal and meta-languages, focused on their expressivity, on consistency problems and on the possibility of translating modal languages into a meta-level setting, the reader may refer to [16].

2 Meta-programming and Meta-reasoning

Whatever the underlying computational paradigm, every piece of software included in any system (in the following, we will say *software component*) manipulates some kind of data, organized in suitable data structures. Data can be used in various ways: for producing results, sending messages, performing actions, or just updating the component's internal state.

Data are often assumed to denote entities which are *external* to the software component. Whenever the computation should produce effects that are visible in the external environment, it is necessary to assume that there exists a *causal connection* between the software system and the environment, in the sense that the intended effect is actually achieved, by means of suitable interface devices. This means, if the software component performs an action in order, for instance, either to print some text, or to send an e-mail message, or to switch a light on, causal connection should guarantee that this is what actually happens.

There are software components however that take other programs as data. An important well-known example is a compiler, which manipulates data structures representing the source program to be translated. A compiler can be written in the language it is intended to translate (for instance, a C compiler can be written in C), or in a different language as well. It is important to notice that in any case there is no mixture between the compiler and the source program. The compiler performs a computation whose outcome is some transformed form of the source program. The source program is just text, recorded in a suitable data structure, that is step by step transformed into other representations. In essence, a compiler accepts and manipulates a *description* of the source program.

In logic, a language that takes sentences of another language as its objects of discourse is called a meta-language. The other language is called the object language. A clear separation between the object language and the meta-language is necessary: namely, it consists in the fact that sentences written in the meta-language can refer to sentences written in the object language only by means of some kind of *description,* or *encoding,* so that sentences written in the object language are treated as data. As it is well-known, Kurt Gödel developed a technique (gödelization) for coding the formulas of the theory of arithmetic by means of numbers (gödel numbers). Thus, it became possible to write formulas for manipulating other formulas, the latter represented by the corresponding gödel numbers.

In this view a compiler is a *meta-program,* and writing a compiler is more than just programming: it is meta-programming. The language in which the compiler is written acts as a meta-language. The language in which the source program is written acts as the object language. More generally, all tools for program analysis, debugging and transformation are meta-programs. They perform a kind of meta-programming that can be called *syntactic* meta-programming.

Syntactic meta-programming can be particularly useful for theorem proving. In fact, as first stressed in [17] and [18], many lemmas and theorems are actually meta-theorems, asserting the validity of a fact by simply looking at its syntactic structure. In this case a software component, namely the theorem prover, consists of two different parts: one, that we call the object level, where proofs are performed by repeatedly applying the inference rules; another one, that we call the meta-level, where meta-theorems are stated.

We may notice that a theorem prover is an "intelligent" system that performs deduction, which is a form of (mechanized) "reasoning". Then, we can say that the theorem prover at the object level performs "object-level reasoning". Meta-theorems take as arguments the description of object-level formulas and theorems, and meta-level proofs manipulate these descriptions. Then, at the meta-level the system performs reasoning about entities that are internal to the system, as opposed to object-level reasoning that concerns entities denoting elements of some external domain. This is why we say that at the meta-level the theorem prover performs "meta-level reasoning", or shortly meta-reasoning.

Meta-theorems are a particular kind of meta-knowledge, i.e. knowledge about properties of the object-level knowledge.

The object and the meta-level can usefully interact: meta-theorems can be used in order to shorten object-level proofs, thus improving the efficiency of the theorem prover, which can derive proofs more easily. In this view, meta-theorems may constitute *auxiliary inference rules* that enhance (in a pragmatic view) the "deductive power" of the system [19] [20]. Notice that, at the meta-level, new meta-theorems can also be proved, by applying suitable inference rules.

As pointed out in [21], most software components implicitly incorporate some kind of meta-knowledge: there are pieces of object-level code that "do" something in accordance to what meta-knowledge states. For instance, an object-level planner program might "know" that *near(b,a)* holds whenever *near(a,b)* holds,

while this is not the case for *on(a,b)*. A planner with a meta-level could explicitly encode a meta-rule stating that whenever a relation $\mathcal{R}$ is symmetric, then $\mathcal{R}(a, b)$ is equivalent to $\mathcal{R}(b, a)$ and whenever instead a relation is antisymmetric this is never the case. So, at the meta-level, there could be statements that *near* is symmetric and *on* is antisymmetric.

The same results could then be obtained by means of explicit meta-reasoning, instead of implicit "knowledge" hidden in the code. The advantage is that the meta-reasoning can be performed in the same way for *any* symmetric and antisymmetric relation that one may have. Other properties of relations might be encoded at the meta-level in a similar way, and such a meta-level specification (which is independent of the specific object-level knowledge or application domain) might be reused in future applications.

There are several possible architectures for meta-knowledge and meta-reasoning, and many applications. Some of them are reviewed later. For a wider perspective however, the reader may refer to [22], [23], [24], [25], [20], [26], [27], [28], [29], [30], [31], [32], [33] where various specific architectures, applications and systems are discussed.

3 Reification

Meta-level rules manipulate a *representation* of object-level knowledge. Since knowledge is represented in some kind of language, meta-rules actually manipulate a representation of syntactic expressions of the object-level language.

In analogy with natural language, such a representation is usually called a *name* of the syntactic expression. The difference between a word of the language, such as for instance *flower,* and a name, like *"flower",* is the following: the word is used to denote an entity of the domain/situation we are talking about; the name denotes the word, so that we can say that "flower" is composed of six characters, is expressed in English and its translation into Italian is "fiore". That is, a word can be used, while a name can be inspected (for instance to count the characters) and manipulated (for instance translated).

An expression in a formal language may have different kinds of names that allow different kinds of meta-reasoning to be made on that expression. Names are expressions of the meta-language.

Taking for instance an equation such as

$$a = b - 2$$

we may have a very simple name, like in natural language, i.e.

$$\text{“}a = b - 2\text{”}$$

This kind of name, called *quotation mark name,* is usually intended as a constant of the meta-language.

A name may be instead a complex term, such as:

$equation$
$(left\_hand\_side(variable(\text{“}a\text{”})),$
$(right\_hand\_side$
$(binop(minus, firstop(variable(\text{“}b\text{”})), secondop(constant(\text{“}2\text{”})))))$

This term describes the equation in terms of its left-hand side and right-hand side and then describes the right-hand side as the application of a binary operator (*binop*) on two operands ($firstop$ and *secondop*) where the first operand is a variable and the second one a constant. “a”, “b” and “2” are constants of the meta-language, they are the names of the expressions a, b and 2 of the object language.

This more complex name, called a *structural description name,* makes it easier to inspect the expression (for instance to see whether it contains variables) and to manipulate it (for instance it is possible to transform this name into the name of another equation, by modifying some of the composing terms).

Of course, many variations are possible in how detailed names are, and what kind of detail they express. Also, many choices can be made about what names should be: for instance, the name of a variable can be a meta-constant, but can also be a meta-variable. For a discussion of different possibilities, with their advantages and disadvantages, see [34], [35], [36].

The definition of names, being a relation between object-level expressions and meta-level expressions that play the role of names, is usually called *naming relation.*

Which naming relation to choose? In general, it depends upon the kind of meta-reasoning one wants to perform. In fact, a meta-theory can only reason about the properties of object-level expressions made explicit by the naming relation. We may provide names to any language expression, from the simplest, to the more complex ones. In a logic meta-language, we may have names for variables, constants, function and predicate symbols, terms and atoms and even for entire theories: the meta-level may in principle encode and reason about the description of several object-level theories. In practice, there is a trade-off between expressivity and simplicity. In fact, names should be kept as simple as possible, to reduce the complexity (and improve the readability) of meta-level expressions. Starting from these considerations, [37] argues that the naming relation should be adapted to each particular case and therefore should be definable by the user.

In [38] it is shown that two different naming relations can coexist in the same context, for different purposes, also providing operators for transforming one representation into the other one.

The definition of a naming relation implies the definition of two operation: the first one, to compute the name of a given language expression. The second one, to compute the expression a given name stands for. The operation of obtaining the name of an object-level expression is called *reification* or *referentiation* or *quoting.* The inverse operation is called *dereification* or *dereferentiation* or *unquoting.* These are built-in operations, whose operational semantics consists in applying the naming relation in the two directions.

In [39] it is shown how the naming relation can be a sort of input parameter for a meta-language. That is, a meta-language may be, if carefully designed, to a large extent independent of the syntactic form of names, and of the class of expressions that are named. Along this line, in [36] and [33] a full theory of definable naming relations is developed, where a naming relation (with some basic properties) can be defined as a set of equations, with the associated rewrite system for applying referentiation/dereferentiation.

4 Introspection and Reflection

The idea that meta-knowledge and meta-reasoning could be useful for improving the reasoning performed at the object level (for instance by exploiting properties of relations, like symmetry), suggests that the object and the meta-level should interact. In fact, the object and the meta-level can be seen as different software components that interact by passing the control to each other.

At the object level, the operation of referentiation allows an expression to be transformed into its name and this name can be given as input argument to a meta-level component. This means that object-level computation gives place to meta-level computation. This computational step is called *upward reflection,* or *introspection,* or *shift up. Upward* because the meta-level is considered to be a "higher level" with respect to the object level. *Reflection,* or *introspection,* because the object level component suspends its activity, in order to initiate a meta-level one. This is meant to be in analogy with the process by which people become conscious (at the meta-level of mind) of mental states they are currently in (at the object level).

The inverse action, that consists in going back to the object-level activity, is called *downward reflection,* or *shift down.* The object-level activity can be resumed from where it had been suspended, or can be somehow restarted. Its state (if any) can be the same as before, or can be altered, according to the meta-level activity that has been performed. Downward reflection may imply that some name is dereferenced and the resulting expression ("extracted" from the name) given as input argument to the resumed or restarted object-level activity.

In logical languages, upward and downward reflection can be specified by means of special inference rules (reflection rules) or axioms (reflection axioms), that may also state what kind of knowledge is exchanged.

In functional and procedural languages, part of the run-time state of the object-level ongoing computation can be reified and passed to a meta-level function/procedure that can inspect and modify this state. When this function terminates, object-level computation resumes on a possibly modified state.

A *reflection act,* that shifts the level of the activity between the object and the meta-level, may be: *explicit,* in the sense that it is either invoked by the user (in interactive systems) or determined by some kind of specification explicitly present in the text of the theory/program; *implicit,* in the sense that it is auto-

matically performed upon occurrence of certain predefined conditions. Explicit and implicit reflection may co-exist.

Both forms of reflection rely on the requirement of *causal connection* or, equivalently, of *introspective fidelity:* that is, the recommendations of the meta-level must be always followed at the object level. For instance, in the procedural case, the modifications to the state performed at the meta-level are effective and have a corresponding impact on the object-level computation. The usefulness of reflection consists exactly in the fact that the overall system (object + meta-levels) not only reasons about itself, but is also properly affected by the results of that reasoning.

In summary, a meta-level architecture for building software components has to provide the possibility of defining a meta-level that by means of a naming relation can manipulate the representation of object-level expressions. Notice that the levels may be several: beyond the meta-level there may be a meta-meta-level that uses a naming relation representing meta-level expressions. Similarly, we can have a meta-meta-meta-level, and so on. Also, we may have one object level and several independent meta-levels with which the object level may be from time to time associated, for performing different kinds of meta-reasoning.

The architecture may provide a reflection mechanism that allows the different levels to interact. If the reflection mechanism is not provided, then the computation is performed at the meta-level, that simulates the object-level formulas through the naming relation and simulates the object-level inference rules by means of meta-level axioms. As discussed later, this is the case in many of the main approaches to meta-reasoning.

The languages in which the object level and the meta-level(s) are expressed may be different, or they may coincide. For instance, we may have a meta-level based on a first-order logic language, were meta-reasoning is performed about an object level based on a functional or imperative language. Sometimes the languages coincide: the object language and the meta-language may be in fact the same one. In this case, this language is expressive enough as to explicitly represent (some of) its own syntactic expressions, i.e. the language is capable of *self-reference.* An interesting deep discussion about languages with self-reference can be found in [40] and [41]. The role of introspection in reasoning is discussed in [42] and [43]. An interesting contribution about reflection and its applications is [44].

5 Seminal Approaches

5.1 FOL

FOL [19], standing for *First Order Logic,* has been (to the best of our knowledge) the first reflective system appeared in the literature. It is a proof checker based on natural deduction, where knowledge and meta-knowledge are expressed in different contexts. The user can access these contexts both for expressing and for inferring new facts.

The FOL system consists of a set of theories, called contexts, based on a first-order language with sorts and conditional expressions.

A special context named META describes the proof theory and some of the model theory of FOL contexts. Given a specific context C that we take as the object theory, the naming relation is defined by attachments, which are user-defined explicit definitions relating symbols and terms in META with their interpretation in C.

The connection between C and META is provided by a special linking rule that is applicable in both directions:

$$\frac{Theorem(\text{“}W\text{”})}{W}$$

where W is any formula in the object theory C, “W” is its name, and $Theorem$(“W”) is a fact in the meta-theory. By means of a special primitive, called REFLECT, the linking rule can be explicitly applied by the user. Its effect is either that of *reflecting up* a formula W to the meta-theory, to derive meta-theorems involving “W”, or vice versa that of *reflecting down* a meta-theorem “W”, so that W becomes a theorem of the theory. Meta-theorems can therefore be used as subsidiary deduction rules.

Interesting applications of the FOL system to mathematical problems can be found in [17], [45].

5.2 Amalgamating Language and Meta-language in Logic Programming

A seminal approach to reflection in the context of the Horn clause language is MetaProlog, proposed by Bowen and Kowalski [46]. The proposal is based on representing Horn clause syntax and provability in the logic itself, by means of a meta-interpreter, i.e. an interpreter of the Horn clause language written in the Horn clause language itself. Therefore, also in this case the object language and the meta-language coincide.

The concept (and the first implementation) of a meta-interpreter was introduced by John McCarthy for the LISP programming language [47]. McCarthy in particular defined a universal function, written in LISP, which represents the basic features of a LISP interpreter. In particular, the universal function is able to: (i) accept as input the definition of a LISP function, together with the list of its arguments; (ii) evaluate the given function on the given arguments. Bowen and Kowalski, with MetaProlog, have developed this powerful and important idea in the field of logic programming, where the inference process is based on building proofs from a given theory, rather than on evaluating functions.

The Bowen and Kowalski meta-interpreter is specified via a predicate *demo,* that is defined by a set of meta-axioms Pr, where the relevant aspects of Horn-clause provability are made explicit. The *Demo* predicate takes as first argument the representation (name) of an object-level theory T and the representation (name) of a goal A. *Demo(“T”,“A”)* means that the goal A is provable in the theory T.

With the above formulation, we might have an approach where inference is performed at the meta-level (via invocation of *Demo*) and the object level is simulated, by providing *Demo* with a suitable description "T" of an object theory T.

The strength and originality of MetaProlog rely instead in the *amalgamation* between the object level and the meta-level. It consists in the introduction of the following *linking rules* for upward and downward reflection:

$$\frac{T \vdash_L A}{Pr \vdash_M Demo(\text{“}T\text{”}, \text{“}A\text{”})} \qquad \frac{Pr \vdash_M Demo(\text{“}T\text{”}, \text{“}A\text{”})}{T \vdash_L A}$$

where $\vdash_M$ means provability at the meta-level M and $\vdash_L$ means provability at the object level L.

The application of the linking rules coincides, in practice, with the invocation of *Demo*, i.e., reflection is *explicit*. Amalgamation allows mixed sentences: there can be object-level sentences where the invocation of *Demo* determines a shift up to the meta-level, and meta-level sentences where the invocation of *Demo* determines a shift down to the object level. Since moreover the theory in which deduction is performed is an input argument of *Demo*, several object-level and meta-level theories can co-exist and can be used in the same inference process.

Although the extension is conservative, i.e. all theorems provable in $L+M$ are provable either in L or in M alone, the gain of expressivity, in practical terms, is great. Many traditional problems in knowledge representation find here a natural formulation.

The extension can be made non-conservative, whenever additional rules are added to *Demo*, to represent auxiliary inference rules and deduction strategies. Additional arguments can be added to *Demo* for integrating forms of *control* in the basic definition of provability. For instance it is possible to control the amount of resources consumed by the proof process, or to make the structure of the proof explicit.

The semantics of the *Demo* predicate is, however, not easy to define (see e.g. [35], [48], [49], [50]), and holds only if the meta-theory and the linking rules provide an extension to the basic Horn clause language which is conservative, i.e., only if *Demo* is a faithful representation of Horn clause provability. Although the amalgamated language is far more expressive than the object language alone, enhanced meta-interpreters are (semantically) ruled out, since in that case the extension is non-conservative.

In practice, the success of the approach has been great: enhanced meta-interpreters are used everywhere in logic programming and artificial intelligence (see for instance [51], or any other logic programming textbook). This seminal work has initiated the whole field of meta-programming in logic programming and computational logic. Problems and promises of this field are discussed by Kowalski himself in [52], [53]. The approach of meta-interpreters and other relevant applications of meta-programming are discussed in the next section.

5.3 3-LISP

3–Lisp [54] is another important example of a reflective architecture where the object language and meta-language coincide. 3–Lisp is a meta-interpreter for Lisp (and therefore it is an elaboration of McCarthy's original proposal) where (the interesting aspects of) the state of the program that is being interpreted are not stored, but are passed by as an argument of all the functions that are internal to the meta-interpreter. Then, each of these procedures takes the state as argument, makes some modification and passes the modified state to another internal procedure. These procedures call each other tail-recursively (i.e. the next procedure call is the last action they make) so as the state remains always explicit. Such a meta-interpreter is called a meta-circular interpreter. If one assumes that the meta-circular interpreter is itself executed by another meta-circular interpreter and so on, one can imagine a potentially infinite *tower* of interpreters, the lowest one executing the object level program (see the summary and formalization of this approach presented in [55]).

Here, the meta-level is accessible from the object level at run-time through a reflection act represented by a special kind of function invocation. Whenever the object-level program invokes any function f in this special way, f receives as an additional parameter a representation of the state of the program itself. Then, f can inspect and/or modify the state, before returning control to object-level execution. A reflective act implies therefore the reification of the state and the execution of f as if it were a procedure internal to the interpreter. Since f might in turn contain a reflection act, the meta-circular interpreter is able to reify its own state and start a brand-new copy of itself. In this approach one might in principle perform, via reflection, an *infinite regress* on the reflective tower of interpreters.

A program is thus able to interrupt its computation, to change something in its own state, and to continue with a modified interpretation process. This kind of mechanism is called *computational reflection*. The semantics of computational reflection is procedural, however, rather than declarative. A reflective architecture conceptually similar to 3-Lisp has been proposed for the Horn clause language and has been fully implemented [56].

Although very procedural in nature, and not easy to understand in practice, computational reflection has been having a great success in the last few years, especially in the context of imperative and object-oriented programming [11], [12]. Some authors even propose computational reflection as the basis of a new programming paradigm [57].

Since computational reflection can be perceived as the only way of performing meta-reasoning in non-logical paradigms, this success enlights once more how important meta-reasoning is, especially for complex applications.

5.4 Other Important Approaches

The amalgamated approach has been experimented by Attardi and Simi in Omega [58]. Omega is an object-oriented formalism for knowledge representation

which can deal with meta-theoretical notions by providing objects that describe Omega objects themselves and derivability in Omega.

A non-amalgamated approach in logic programming is that of the Gödel language, where object theory and meta-theory are distinct. Gödel provides a (conservative) provability predicate, and an explicit form of reflection. The language has been developed and experimented in the context of the Compulog European project. It is described in the book [59]. In [60] a contribution to meta-programming in Gödel is proposed, on two aspects: on the one hand, a programming style for efficient meta-programming is outlined; on the other hand, modifications to the implementation are proposed, in order to improve the performance of meta-programs.

A project that extends and builds on both FOL and 3–Lisp is GETFOL [61],[62]. It is developed on top of a novel implementation of FOL (therefore the approach is not amalgamated: the object theory and meta-theory are distinct). GETFOL is able to introspect its own code (lifting), to reason deductively about it in a declarative meta-theory and, as a result, to produce new executable code that can be pushed back to the underlying interpretation (flattening).

The architecture is based on a sharp distinction between deduction (FOL style) and computation (3–Lisp style). Reflection in GETFOL gives access to a meta-theory where many features of the system are made explicit, even the code that implements the system itself.

The main objective of GETFOL is that of implementing theorem-provers, given its ability of implementing flexible control strategies to be adapted (via computational reflection) to the particular situation. Similarly to FOL, the kind of reasoning performed in GETFOL consists in: (i) performing some reasoning at the meta-level; (ii) using the results of this reasoning to assert facts at the object level.

An interesting extension is that of applying this concept to a system with multiple theories and multiple languages (each theory formulated in its own language) [63], where the two steps are reinterpreted as (i) doing some reasoning in one theory and (ii) jumping into another theory to do some more reasoning on the basis of what has been derived in the previous theory. These two deductions are concatenated by the application of *bridge rules,* which are inference rules where the premises belong to the language of the former theory, and the conclusion belongs to the language of the latter.

A different concept of reflection is embodied in Reflective Prolog [39] [64] [65], a self-referential Horn clause language with logical reflection. The objective of this approach is that of developing a more expressive and powerful language, while preserving the essential features of logic programming: Horn clause syntax, model-theoretic semantics, resolution via unification as procedural semantics, correctness and completeness properties.

In Reflective Prolog, Horn clauses are extended with self-reference and resolution is extended with logical reflection, in order to achieve greater expressive and inference power. The reflection mechanism is *implicit*, i.e., the interpreter of the language automatically reflects upwards and downwards by applying suit-

able linking rules called *reflection principles.* This allows reasoning and meta-reasoning to interleave without user's intervention, so as to exploit both knowledge and meta-knowledge in proofs: in most of the other approaches instead, there is one level which is "first–class", where deduction is actually performed, and the other level which plays a secondary role.

Reflection principles are embedded in both the procedural and the declarative semantics of the language, that is, in the extended resolution procedure which is used by the interpreter and in the construction of the models which give meanings to programs.

Procedurally, this implies that there is no need to axiomatize provability in the meta-theory. Object level reasoning is not simulated by meta-interpreters, but directly executed by the language interpreter, thus avoiding unnecessary inefficiency. Semantically, a theory composed of an object level and (one or more) meta-levels is regarded as an enhanced theory, enriched by new axioms which are entailed by the given theory and by the reflection principles interpreted as axiom schemata. Therefore, in Reflective Prolog, language and metalanguage are amalgamated in a non-conservative extension.

Reflection in Reflective Prolog gives access to a meta-theory where various kinds of meta-knowledge can be expressed, either about the application domain or about the behavior of the system. Deduction in Reflective Prolog means using at each step either meta-level or object level knowledge, in a continuous interleaving between levels. Meta-reasoning in Reflective Prolog implies a declarative definition of meta-knowledge, which is automatically integrated into the inference process. The relation between meta-reasoning in Reflective Prolog and modal logic has been discussed in [66].

An interpreter of Reflective Prolog has been fully implemented [67]. It is interesting to notice that Reflective Prolog has been implemented by means of computational reflection. This is another demonstration that computational reflection can be a good (although low-level) implementation tool.

An approach that has been successful in the context of object-oriented languages, including the most recent ones like Java, is the *meta-object protocol.* A meta-object protocol [68] [69] gives every object a corresponding meta-object that is an instance of a meta-class. Then, the behavior of an object becomes the behavior of the object/meta-object pair. At the meta-level, important aspects such as the operational semantics of inheritance, instantiation and method invocation can be defined. A meta-object protocol constitutes a flexible mean of modifying and extending an object-oriented language.

This approach has been applied to logic programming, in the ObjVProlog language [70] [71]. In addition to the above-mentioned meta-class capabilities, this language preserves the Prolog capabilities of manipulating clauses in the language itself, and provides a provability predicate.

As an example of more recent application of this approach, a review of Java reflective implementations can be found in [72].

A limitation is that only aspects directly related to objects can be described in a meta-object. Properties of sets of objects, or of the overall system, cannot

be *directly* expressed. Nevertheless, some authors [72] argue that non-functional requirements such as security, fault-tolerance, atomicity, can be implemented by *implicit* reflection to the meta-object before and after the invocation of *every* object method.

6 Applications of Meta-Reasoning

Meta-reasoning has been widely used for a variety of purposes, and recently the interest in new potential applications of meta-reasoning and reflection has been very significant. In this section, we provide our (necessarily partial and limited) view of some of the more relevant applications in the field.

6.1 Meta-interpreters

After the seminal work of Bowen and Kowalski [46], the most common application of meta-logic in computational logic is to define and to implement meta-interpreters. This technique has been especially used in Prolog (which is probably the most popular logic programming language) for a variety of purposes.

The basic version of a meta-interpreter for propositional Horn clause programs, reported in [53], is the following.

$demo(T, P) \leftarrow demo(T, P \leftarrow Q), demo(T, Q).$
$demo(T, P \wedge Q) \leftarrow demo(T, P), demo(T, Q).$

In the above definition, '$\wedge$' names conjunction and '$\leftarrow$' names '$\leftarrow$' itself. A theory can be named by a list containing the names of its sentences. In the propositional case, formulas and their names may coincide without the problems of ambiguity (discussed below), that arise in presence of variables. If a theory is represented by a list, then the meta-interpreter must be augmented by the additional meta-axiom:

$demo(T, P) \leftarrow member(T, P).$

For instance, query $?q$ to program

$q \leftarrow p, s.$
$p.$
$s.$

can be simulated by query $?demo([q \leftarrow p \wedge s, p, s], q)$ to the above meta-interpreter. Alternatively, it is possible to use a constant symbol to name a theory. In this case, the theory, say $t1$, can be defined by the following meta-level axioms:

$demo(t1, q \leftarrow p \wedge s).$
$demo(t1, p).$
$demo(t1, s).$

and the query becomes $?demo(t1, q)$.

The meta-axioms defining *demo* can be themselves regarded as a theory that can be named, by either a list or a constant (say d). Thus, it is possible to write queries like $?demo(d, demo(t1, q))$ which means to ask whether we can derive, by the meta-interpreter d, that the goal q can be proved in theory $t1$.

In many Prolog applications however, the theory argument is omitted, as in the so-called "Vanilla" meta-interpreter [35]. The standard declarative formulation of the Vanilla meta-interpreter in Prolog is the following (where ':−' is the Prolog counterpart of '←' and '&' indicates conjunction):

$demo(empty)$.
$demo(X) :- clause(X, Y), demo(Y)$.
$demo(X\&Y) :- demo(X), demo(Y)$.

For the above object-level program, we should add to the meta-interpreter the unit clauses:

$clause(q,\ p\&s)$.
$clause(p,\ empty)$.
$clause(s,\ empty)$..

and the query would be $:- demo(q)$.

The vanilla meta-interpreter can be used for propositional programs, as well as for programs containing variables. In the latter case however, there is an important ambiguity concerning variables. In fact, variables in the object-level program are meant to range (as usual) over the domain of the program. These variables are instantiated to object-level terms. Instead, the variables occurring in the definition of the meta-interpreter, are intended to range over object-level atoms. Then, in a correct approach these are meta-variables (for an accurate discussion of this problem see [34]).

In [35], a typed version of the Vanilla meta-interpreter is advocated and its correctness proved. In [46] and [65], suitable naming mechanisms are proposed to overcome the problem.

Since however it is the untyped version that is generally used in Prolog practice, some researchers have tried to specify a formal account of the Vanilla meta-interpreter as it is. In particular, a first-order logic with ambivalent syntax has been proposed to this purpose [73], [74] and correctness results have been obtained [75].

The Vanilla meta-interpreter can be enhanced in various ways, often by making use of built-in Prolog meta-predicates that allow Prolog to act as a meta-language of itself. These predicates in fact are aimed at inspecting, building and modifying goals and at inspecting the instantiation status of variables.

First, more aspects of the proof process can be made explicit. In the above formalization, unification is implicitly demanded to the underlying Prolog interpreter and so is the order of execution of subgoals in conjunctions. Below is a formulation where these two aspects become explicit. Unification is performed by a *unify* procedure and *reorder* rearranges subgoals of the given conjunction.

$demo(empty)$.
$demo(X) :- clause(H, Y), unify(H, X, Y, Y1), demo(Y1)$.
$demo(X\&Y) :- reorder(X\&Y, X1\&Y1), demo(X1), demo(Y1)$.

Second, extra arguments can be added to *demo*, to represent for instance: the maximum number of steps that *demo* is allowed to perform; the actual number of steps that *demo* has performed; the proof tree; an explanation to be returned to a user and so on. Clearly, the definition of the meta-interpreter will be suitably modified according to the use of the extra arguments.

Third, extra rules can enhance the behavior of the meta-interpreter, by specifying auxiliary deduction rules. For instance, the rule

$demo(X) :- ask(X, yes)$.

states that we consider X to be true, if the user answers "yes" when explicitly asked about X. In this way, the meta-interpreter exhibits an interactive behavior. The auxiliary deduction rules may be several and may interact.

In Reflective Prolog, [65] one specifies the additional rules *only,* while the definition of standard provability remains implicit. In the last example for instance, on failure of goal X, a goal $demo(X)$ would be automatically generated (this is an example of implicit upward reflection), thus employing the additional rule to query the user about X.

An interesting approach to meta-interpreters is that of [76], [77], where a binary predicate *demo* may answer queries with uninstantiated variables, which represent arbitrary fragments of the program currently being executed.

The reader may refer to [51] for an illustration of the meta-interpreter programming techniques and of their applications, including the specification of Expert Systems in Prolog.

6.2 Theory Composition and Theory Systems

Theory construction and combination is an important tool of software engineering, since it promotes modularity, software reuse and programming-in-the-large. In [53] it is observed that theory-construction can be regarded as a meta-linguistic operation. Within the Compulog European projects, two meta-logic approaches to working with theories have been proposed.

In the Algebra of Logic Programs, proposed in [78] and [79], a *program expression* defines a combination of object programs (that can be seen as theories, or modules) through a set of composition operators. The provability of a query with respect to a composition of programs can be defined by meta-axioms specifying the intended meaning of the various composition operations.

Four basic operations for composing logic programs are introduced: encapsulation (denoted by $*$), union ($\cup$), intersection ($\cap$) and import ($\lhd$).

Encapsulation copes with the requirement that a module can import from another one only its functionality, without caring of the implementation. This kind of behavior can be realized by encapsulation and union: if P is the "main program" and S is a module, the combined program is:

$$P \cup S^*$$

Intersection yields a combined theory where both the original theories are forced to agree during deduction, on every single partial conclusion.

The operation $\lhd$ builds a module $P \lhd Q$ out of two modules P and Q, where P is the *visible part* and Q the *hidden part* of the resulting module.

The usefulness of these operators for knowledge representation and reasoning is shown in [78]. The meta-logical definition of the operations is given in [79], by extending the Vanilla meta-interpreter. Two alternative implementations using the Gödel programming language are proposed and discussed in [80]. One extends the untyped Vanilla meta-interpreter. The other one exploits the meta-programming facilities offered by the language, thus using names and typed variables. The second, cleaner version seems to the authors themselves more suitable than the first one, for implementing program composition operations requiring a fine-grained manipulation of the object programs.

In the Alloy language, proposed in [81] and [82], a theory system is a collection of interdependent theories, some of which stand in a meta/object relationship, forming an arbitrary number of meta-levels. Theory systems are proposed for a meta-programming based software engineering methodology aimed at specifying, for instance, reasoning agents, programs to be manipulated, programs that manipulate them, etc. The meta/object relationship between theories provides the inspection and control facilities needed in these applications.

The basic language of theory systems is a definite clause language, augmented with ground names for every well-formed expression of the language. Each theory is named by a ground *theory term.* A theory system can be defined out of a collection of theories by using the following tools.

1. The symbol '$\vdash$' for relating theory terms and sentences. A *theoremhood statement,* like for instance $t_1 \vdash \lceil u_1 \vdash \Psi \rceil$ where t_1 and u_1 are theory terms, says that $\lceil u_1 \vdash \Psi \rceil$ is a theorem of theory t_1.
2. The distinguishes function symbol '$\diamond$', where $t_1 \diamond t_2$ means that t_1 is a meta-theory of t_2.
3. The *coincidence statement* $t_1 \equiv t_2$, expressing that t_1 and t_2 have exactly the same theorems.

The behavior of the above operators is defined by reflection principles (in the form of meta-axioms) that are suitably integrated in the declarative and proof-theoretic semantics.

6.3 The Event Calculus

Representing and reasoning about actions and temporally-scoped relations has been for years one of the key research topics in knowledge representation [83]. The Event Calculus (EC) has been proposed by Kowalski and Sergot [84] as a system for reasoning about time and actions in the framework of Logic Programming. In particular, the Event Calculus adapts the ontology of McCarthy and Hayes's *Situation Calculus* [85] i.e., actions and fluents [1], to a new task: *assimilating a narrative*, which is the description of a course of events. The essential

[1] It is interesting to notice that the fluent/fluxion terminology dates back to Newton

idea is to have terms, called *fluents,* which are names of time-dependent relations. Kowalski and Sergot however write $holds(r(x,y),t)$ which is understood as "fluent $r(x,y)$ is true at time t", instead of $r(x,y,t)$ like in situation calculus.

It is worthwhile to discuss the connection between Kowalski's work on meta-programming and the definition of the Event Calculus. In the logic programming framework it comes natural to recognize the higher-order nature of time-dependent propositions and to try to represent them at the meta-level. Kowalski in fact [86] considers McCarthy's Situation Calculus and comments:

> Thus we write
>
> $$Holds(possess(Bob,\ Book1),\ S0)$$
>
> instead of the weaker but also adequate
>
> $$Possess(Bob,\ Book1,\ S0).$$
>
> In the first formulation, $possess(Bob,\ Book1)$ is a term which names a relationship. In the second, $Possess(Bob,\ Book1,\ S0)$ is an atomic formula. Both representations are expressed within the formalism of first-order classical logic. However, the first allows variables to range over relationships whereas the second does not. If we identify relationships with atomic variable-free sentences, then we can regard a term such as $possess(Bob,\ Book1)$ as the name of a sentence. In this case $Holds$ is a meta-level predicate [...]

There is a clear advantage with reification from the computational point of view: by reifying, we need to write only one frame axiom, or inertia law, saying that truth of any relation does not change in time unless otherwise specified. Negation-as-failure is a natural choice for implementing the default inertia law. In a simplified, time points-oriented version, default inertia can be formulated as follows:

$$\begin{aligned} Holds(f,t) \leftarrow\ & Happens(e),\\ & initiates(e,f),\\ & Date(e,t_s),\\ & t_s < t,\\ & not\ Clipped(t_s,f,t) \end{aligned}$$

where $Clipped(t_s,f,t)$ is true when there is record of an event happening between t_s and t that terminates the validity of f. In other words, $Holds(f,t)$ is derivable whenever in the interval between the initiation of the fluent and the time the query is about, no terminating events has happened.

It is easy to see $Holds$ as a specialization of $Demo$. Kowalski and Sadri [87] [88], discuss in depth how an Event Calculus program can be specified and assumptions on the nature of the domain accommodated, by manipulating the usual Vanilla meta-interpreter definition.

Since the first proposal, a number of improved formalization have steamed, in order to adapt the calculus to different tasks, such as abductive planning, diagnosis, temporal database and models of legislation. All extensions and applications cannot be accounted for here, but the reader may for instance refer to [89], [90], and [91].

6.4 Logical Frameworks

A logical framework [92] is a formal system that provides tools for experimenting with deductive systems. Within a logical framework, a user can invent a new deductive system by defining its syntax, inference rules and proof-theoretic semantics. This specification is executable, so as the user can make experiments with this new system. A logical framework however cannot reasonably provide tools for defining any possible deductive system, but will stay within a certain class.

Formalisms with powerful meta-level features and strong semantic foundations have the possibility of evolving towards becoming logical frameworks.

The Maude system for instance [93] is a particular implementation of the meta-theory of rewriting logic. It provides the predefined functional module META-LEVEL, where Maude terms can be reified and where: the process of reducing a term to a normal form is represented by a function *meta-reduce*; the default interpreter is represented by a function *meta-rewrite*; the application of a rule to a term by *meta-apply*.

Recently, a reflective version of Maude has been proposed [94], based on the formalization of computational reflection proposed in [95]. The META-LEVEL module has been made more flexible, so as to allow a user to define the syntax of her own logic language L by means of meta-rules. The new language must however consist in an addition/variation to the basic syntax of the Maude language. Reflection is the tool for integrating the user-defined syntax into the proof procedure of Maude. In particular, whenever a piece of user-defined syntax is found, a reflection act to the META-LEVEL module happens, so as to apply the corresponding syntactic meta-rules. Then, the rewriting system Maude has evolved into a logical framework for logic languages based on rewriting.

The RCL (Reflective Computational Logic) logical framework [33] is an evolution of the Reflective Prolog metalogic language. The implicit reflection of Reflective Prolog has a semantic counterpart [39] in adding to the given theory a set of new axioms called *reflection axioms*, according to axiom schemata called *reflection principles*. Reflection principles can specify not only the shift between levels, but also many other meta-reasoning principles. For instance, reflection principles can define forms of analogical reasoning [96], and synchronous communication among logical agents [97].

RCL has originated from the idea that, more generally, reflection principles may be used to express the inference rules of user-defined deductive systems. The deductive systems that can be specified in RCL are however evolutions of the Horn clause language, based on a predefined enhanced syntax. A basic version

of naming is provided in the enhanced Horn clause language, formalized through an equational theory.

The specification of a new deductive system DS in RCL is accomplished through the following four steps.

Step I Definition of the naming device (encoding) for DS. The user definition must extend the predefined one. RCL leaves significant freedom in the representation of names.

Step II After defining the naming convention, the user of RCL has to provide a corresponding unification algorithm (again by suitable additions to the predefined one).

Step III Representation of the axioms of DS, in the form of enhanced Horn clauses.

Step IV Definition of the inference rules of DS as reflection principles.

In particular, the user is required to express each inference rule R as a function $\mathcal{R}$, from clauses, which constitute the antecedent of the rule, to sets of clauses, which constitute the consequent.

Then, given a theory T of DS consisting of a set of axioms A and a reflection principle $\mathcal{R}$, a theory T' containing T is obtained as the deductive closure of $A \cup A'$, where A' is the set of additional axioms generated by $\mathcal{R}$. Consequently, the model-theoretic and fixed point semantics of T under $\mathcal{R}$ are obtained as the model-theoretic and fixed point semantics of T'. RCL does not actually generate T'. Rather, given a query for T, RCL dynamically generates the specific additional axioms usable to answer the query according to the reflection principle $\mathcal{R}$, i.e., according to the inference rule R of DS.

6.5 Logical Agents

In the area of intelligent software agents there are several issues that require the integration of some kind of meta-reasoning ability into the system. In fact, most existing formalisms, systems and frameworks for defining agents incorporate, in different forms, a meta-component.

An important challenge in this area is that of interconnecting several agents that are *heterogeneous* in the sense that they are not necessarily uniform in the implementation, in the knowledge they possess and in the behavior they exhibit. Any framework for developing multi-agent systems must provide a great deal of flexibility for integrating heterogeneous agents and assembling communities of independent service providers. Flexibility is required in structuring cooperative interactions among agents, and for creating more accessible and intuitive user interfaces.

Meta-reasoning is essential for obtaining such a degree of flexibility. Meta-reasoning can either be performed within the single agent, or special meta-agents can be designed, to act as meta-theories for sets of other agents. Meta-reasoning can help: (i) in the interaction among agents and with the user; (ii) in the implementation suitable strategies and plans for responding to requests. These

strategies can be either domain-independent, or rely on domain- and application-specific knowledge or reasoning (auxiliary inference rules, learning algorithms, planning, and so forth)

Meta-rules and meta-programming may be particularly useful for coping with some aspects of the ontology problem: meta-rules can switch between descriptions that are syntactically different though semantically equivalent, and can help fill the gap between descriptions that are not equivalent. Also, meta-reasoning can be used for managing incomplete descriptions or requests.

The following are relevant examples of approaches to developing agent systems that make use of some form of meta-reasoning.

In the Open Agent Architecture^{TM} [98], which is meant for integrating a community of heterogeneous software agents, there are specialized server agents, called *facilitators,* that perform reasoning (and, more or less explicitly, meta-reasoning) about the agent interactions necessary for handling a complex expression. There are also *meta–agents,* that perform more complex meta-reasoning so as to assist the facilitator agent in coordinating the activities of the other agents.

In the constraint logic programming language CaseLP, there are *logical agents,* which show capabilities of complex reasoning, and *interface agents,* which provide an interface with external modules. There are no meta-agents, but an agent has *meta–goals* that trigger meta-reasoning to guide the planning process.

There are applications where agents may have objectives and may need to reason about their own as well as other agents' beliefs and about the actions that agents may take. This is the perspective of the BDI formalization of multi-agent systems proposed in [99] and [100], where BDI stands for "Belief, Desire, Intentions".

The approach of Meta-Agents [101] allow agents to reason about other agents' state, beliefs, and potential actions by introducing powerful meta-reasoning capabilities. Meta-Agents are a specification tool, since for efficient implementation they are translated into ordinary agent programs, plus some integrity constraints.

In logic programming, research on multi-agent systems starts, to the best of our knowledge, from the work by Kim and Kowalski in [102], [103]. The amalgamation of language and meta-language and the *demo* predicate with theories named by constants are used for formalizing reasoning capabilities in multi-agent domains. In this approach, the *demo* predicate is interpreted as a belief predicate and thus agents can reason, like in the BDI approach, about beliefs.

In the effort of obtaining logical agents that are rational, but also reactive (i.e. logical reasoning agents capable of timely response to external events) a more general approach has been proposed in [82], by Kowalski, and in [104] and [105] by Kowalski and Sadri. A meta-logic program defines the "observe-think-act" cycle of an agent. Integrity constraints are used to generate actions in response to updates from the environment.

In the approach of [97], agents communicate via the two meta-level primitives *tell/told.* An agent is represented by a theory, i.e. by a set of clauses prefixed with the corresponding theory name. Communication between agents is formalized by the following reflection principle $\mathcal{R}_{com}$:

$$T : told(\text{“}S\text{”}, \text{“}A\text{”}) \Leftarrow_{\mathcal{R}_{com}} S : tell(\text{“}T\text{”}, \text{“}A\text{”}).$$

The intuitive meaning is that every time an atom of the form $tell(\text{“}T\text{”}, \text{“}A\text{”})$ can be derived from a theory S (which means that agent S wants to communicate proposition A to agent T), the atom $told(\text{“}S\text{”}, \text{“}A\text{”})$ is consequently derived in theory T (which means that proposition A becomes available to agent T).

The objective of this formalization is that each agent can specify, by means of clauses defining the predicate *tell*, the modalities of interaction with the other agents. These modalities can thus vary with respect to different agents or different conditions. For instance, let P be a program composed of three agents, a and b and c, defined as follows.

$a : tell(X, \text{“}ciao\text{”})$:- $friend(X)$.
$a : friend(\text{“}b\text{”})$.

$b : happy$:-$told(\text{“}a\text{”}, \text{“}ciao\text{”})$.

$c : happy$:-$told(\text{“}a\text{”}, \text{“}ciao\text{”})$.

Agent a says “*ciao*” to every other agent X that considers to be its friend. In the above definition, the only friend is b. Agents b and c are happy if a says “*ciao*” to them. The conclusion *happy* can be derived in agent b, while it cannot be derived in agent c. In fact, we get $a : tell(\text{“}b\text{”}, \text{“}ciao\text{”})$ from $a : friend(\text{“}b\text{”})$; instead, $a : tell(\text{“}c\text{”}, \text{“}ciao\text{”})$ is not a conclusion of agent a.

In [106], Dell'Acqua, Sadri and Toni propose an approach to logic-based agents as a combination of the above approaches, i.e. the approach to agents by Kowalski and Sadri [105] and the approach to meta-reasoning by Costantini et al. [65], [97]. Similarly to Kowalski and Sadri's agents, the agents in [106] are *hybrid* in that they exhibit both *rational* (or *deliberative*) and *reactive* behavior. The reasoning core of these agents is a proof procedure that combines forward and backward reasoning. Backward reasoning is used primarily for deliberative activities. Forward reasoning is used primarily for reactivity to the environment, possibly including other agents. The proof procedure is executed within an “observe-think-act” cycle that allows the agent to be alert to the environment and react to it, as well as think and devise plans. The proof procedure (IFF proof procedure proposed by Fung and Kowalski in [107]) treats both inputs from the environment and agents' actions as *abducibles* (hypotheses). Moreover, by adapting the techniques proposed in [97], the agents are capable of reasoning about their own beliefs and the beliefs of other agents.

In [108], the same authors extend the approach by providing agents with *proactive* communication capabilities. Proactive agents are able to communicate on their own initiative, not only in response to stimula. In the resulting framework reactive, rational or hybrid agents can reason about their own beliefs as well as the beliefs of other agents and can communicate proactively with each other. The agents' behavior can be regulated by condition-action rules. In this approach, there are two primitives for communication, *tell* and *ask*, treated as

abducibles within the "observe-think-act" cycle of the agent architecture. The predicate *told* is used to express both passive reception of messages from other agents and reception of information in response to an active request.

The following example is taken by [108] and is aimed at illustrating the basic features of the approach. Let Ag be represented by the abductive logic program $\langle P, \mathcal{A}, I \rangle$ with:

$$P = \left\{ \begin{array}{l} \texttt{told}(A, X) \leftarrow \texttt{ask}(A, X) \wedge \texttt{tell}(A, X) \\ \texttt{told}(A, X) \leftarrow \texttt{tell}(A, X) \\ \texttt{solve}(X) \leftarrow \texttt{told}(A, X) \\ \texttt{desire}(y) \leftarrow y = \texttt{car} \\ \texttt{good\_price}(p, x) \leftarrow p = 0 \end{array} \right\}$$

$$\mathcal{A} = \left\{ \texttt{tell}, \texttt{ask}, \texttt{offer} \right\}$$

$$I = \left\{ \begin{array}{l} \texttt{desire}(x) \wedge \texttt{told}(B, '\texttt{good\_price}('p, 'x)) \\ \quad \Rightarrow \texttt{tell}(B, '\texttt{offer}('p, 'x)) \end{array} \right\}.$$

The first two clauses in P state that Ag may be told something, say X, by another agent A either because A has been explicitly asked about X (first clause) or because A tells X proactively (second clause). The third clause in P says that Ag believes anything it is told. The fourth and fifth clauses in P say, respectively, that the agent desires a car and that anything that is free is at a good price. The integrity constraint says that, if the agent desires something and it is told (by some other agent B) of a good price for it, then it makes an offer to B, by telling it.

The logic programming language DALI [109], is indebted to all previously mentioned approaches to logical agents. DALI introduces explicit reactive and proactive rules at the object level. Thus, reactivity and proactivity are modeled in the basic logic language of the agent In fact, declarative semantics is very close to that of the standard Horn clause language. Procedural semantics relies on an extended resolution. The language incorporates *tell*/*told* primitives, integrity constraints and *solve* rules. An "observe-think-act" cycle can of course been implemented in a DALI agent, but it is no longer necessary for modeling reactivity and proactivity.

Below is a simplified fragment of a DALI agent representing the waiter of a pub, that tries to serve a customer that enters. The customer wants some X. This request is an *external event* (indicated with 'E') that arrives to the agent. The event triggers a reactive rule (indicated with ':>' instead of usual ':-'), and determines the body of the rule to be executed. This is very much like any other goal: only, computation is not initiated by a query, but starts on reception of the event.

During the execution of the body of the reactive rule, the waiter first checks whether X is one of the available drinks. If so, the waiter serves the drink: the predicate *serve_drink* is in fact an action (indicated with 'A'). Otherwise, the waiter checks whether the request is expressed in some foreign language, for which a translation is available (this is a simple example of coping with one

aspect of the ontology problem). If this is not the case, the waiter asks the customer for explanation about X: it expects to be *told* that X is actually an Y, in order to try to serve this Y.

Notice that the predicate *translate* is symmetric, where symmetry is managed by the *solve* rule. To understand the behavior, one can assume this rule to be an additional rule of a basic meta-interpreter that is not explicitly reported. A subgoal like $translate(beer, V)$ is automatically transformed into a call to the meta-interpreter, of the form $solve(\text{“}translate\text{”}(\text{“}beer\text{”}, \text{“}V\text{”}))$ (formally, this is implicit upward reflection). Then, since $symmetric(\text{“}translate\text{”})$ succeeds, $solve(\text{“}translate\text{”}(\text{“}beer\text{”}, \text{“}V\text{”}))$ is attempted, and automatically reflected at the object level (formally, this is implicit downward reflection). Finally, the unquoted subgoal $translate(beer, V)$ succeeds with V instantiated to *birra*.

$$
\begin{aligned}
&Waiter\\
&request(Customer, \text{“}X\text{”})_E :> serve(Customer,X).\\
&\\
&serve(C,X) \text{ :- } drink(X), serve\_drink(C,X)_A.\\
&serve(C,X) \text{ :- } translate(X,Y),\\
&\qquad\qquad\quad drink(Y),\\
&\qquad\qquad\quad serve\_drink(C,Y)_A.\\
&serve(C,X) \text{ :- } ask(C,X,Y), serve(C,Y).\\
&\\
&ask(C,X,Y) \text{ :- } ask\_for\_explanation(C, \text{“}X\text{”}), told(C, \text{“}Y\text{”}).\\
&\\
&drink(beer).\\
&drink(coke).\\
&translate(birra,beer).\\
&translate(cocacola,coke).\\
&symmetric(\text{“}translate\text{”}).\\
&\\
&solve(\text{“}P\text{”}(\text{“}X\text{”}, \text{“}Y\text{”})) \text{ :- } symmetric(\text{“}P\text{”}), solve(\text{“}P\text{”}(\text{“}Y\text{”}, \text{“}X\text{”})).
\end{aligned}
$$

Agents that interact with other agents and/or with an external environment, may expand and modify their knowledge base by incorporating new information. In a dynamic setting, the knowledge base of an agent can be seen as the set of *beliefs* of the agent, that may change over time. An agent may reach a stage where its beliefs have become inconsistent, and actions must be taken to regain consistency. The theory of belief revision aims at modeling how an agent updates its state of belief as a result of receiving new information [110], [111]. Belief revision is, in our opinion, another important issue related to intelligent agents where meta-reasoning can be usefully applied.

In [32] a model-based diagnosis system is presented, capable of revision of the description of the system to be diagnosed if inconsistencies arise from observations. Revision strategies are implemented by means of meta-programming and meta-reasoning methods.

In [112], a framework is proposed where rational, reactive agents can dynamically change their own knowledge bases as well as their own goals. In particular, an agent can make observations, learn new facts and new rules from the environment (even in contrast with its current knowledge) and then update its knowledge accordingly. To solve contradictions, techniques of contradiction removal and preferences among several sources can be adopted [113].

In [114] it is pointed out that most existing approaches to intelligent agents have difficulties to model the way agents revise their beliefs, because new information always come together certain meta-information: e.g., where the new information comes from? Is the source reliable? and so on. Then, the agent has to reason about this meta-information, in order to revise its beliefs. This leads to the proposal of a new approach, where this meta-information can be explicitly represented and reasoned about, and revision strategies can be defined in a declarative way.

7 Semantic Issues

In computational logic, meta-programming and meta-reasoning capabilities are mainly based on self-reference, i.e. on the possibility of describing language expressions in the language itself. In fact, in most of the relevant approaches the object language and the meta-language coincide.

The main tool for self-reference is a naming mechanism. An alternative form of self-reference has been proposed by McCarthy [115], who suggests that introducing function symbols denoting concepts (rather than quoted expressions) might be sufficient for most forms of meta-reasoning. But Perlis [40] observes:

> "The last word you just said" is an expression that although representable as a function still refers to a particular word, not to a concept. Thus quotation seems necessarily involved at some point if we are to have a self-describing language. It appears we must describe specific expressions as carriers of (the meaning of) concepts.

The issue of appropriate language facilities for naming is addressed by Hill and Lloyd in [35]. They point out the distinction between two possible representation schemes: the *non-ground* representation, in which an object-level variable is represented by a meta-level variable, and the *ground* representation, in which object-level expressions are represented by ground (i.e. variable free) terms at the meta-level. In the ground representation, an object level variable may be represented by a meta-level constant, or by any other ground term.

The problem with the non-ground representation is related to meta-level predicates such as the Prolog *var(X),* which is true if the variable X is not instantiated, and is false otherwise. As remarked in [35]:

To see the difficulty, consider the goals:

$$:- var(X) \wedge solve(p(X))$$

and

$$:- solve(p(X)) \wedge var(X)$$

> If the object program consists solely of the clause $p(a)$, then (using the "leftmost literal" computation rule) the first goal succeeds, while the second goal fails.

Hill and Lloyd propose a ground representation of expressions of a first-order language $\mathcal{L}$ in another first-order language $\mathcal{L}'$ with three types ω, μ and η.

Definition 1 (Hill and Lloyd ground representation). *Given a constant a in $\mathcal{L}$, there is a corresponding constant a' of type ω in $\mathcal{L}'$. Given a variable x in $\mathcal{L}$, there is a corresponding constant x' of type ω in $\mathcal{L}'$. Given an n-ary function symbol f in $\mathcal{L}$, there is a corresponding n-ary function symbol f' of type $\omega \times \ldots \omega \longrightarrow \omega$ in $\mathcal{L}'$. Given an n-ary predicate symbol p in $\mathcal{L}$, there is a corresponding n-ary function symbol f' of type $\omega \times \ldots \omega \longrightarrow \mu$ in $\mathcal{L}'$. The language $\mathcal{L}'$ has a constant empty of type μ. The mappings $a \longrightarrow a'$, $x \longrightarrow x'$, $f \longrightarrow f'$ and $p \longrightarrow p'$ are all injective.*

Moreover, $\mathcal{L}'$ contains some function and predicate symbols useful for declaratively redefining the "impure" features of Prolog and the Vanilla meta-interpreter. For instance we will have:

$constant(a'_1)$.
...
$constant(a'_n)$.
$\forall_\omega x\ nonvar(x) \leftarrow constant(x)$.
$\forall_\omega x\ var(x) \leftarrow \neg\, nonvar(x)$.

The above naming mechanism is used in [35] for providing a declarative semantics to a meta-interpreter that implements SLDNF resolution [116] for normal programs and goals. This approach has then evolved into the meta-logical facilities of the Gödel language [59]. Notice that, since names of predicate symbols are function symbols, properties of predicates (e.g. symmetry) cannot be explicitly stated. Since levels in Gödel are separated rather than amalgamated, this naming mechanism does not provide operators for referentiation/dereferentiation.

An important issue raised in [40] is the following:

> Now, it is essential to have also an un-naming device that would return a quoted sentence to its original (assertive) form, together with axioms stating that that is what naming and un-naming accomplish.

Along this line, the approach of [36], developed in detail in [117], proposes to name an atom of the form $\alpha_0(\alpha_1, \ldots, \alpha_n)$ as $[\beta_0, \beta_1, \ldots, \beta_n]$, where each β_i

is the name of α_i. The name of the name of $\alpha_0(\alpha_1, \ldots, \alpha_n)$ is the name term $[\gamma_0, \gamma_1, \ldots, \gamma_n]$, where each γ_i is the name of β_i, etc. Requiring names of compound expressions to be compositional allows one to use unification for constructing name terms and accessing their components.

In this approach, we are able to express properties of predicates by using their names. For instance, we can say that predicate p is binary and predicate q is symmetric, by asserting $\mathit{binary\_pred}(p^1)$ and $\mathit{symmetric}(q^1)$.

Given a term t and a name term s, the expression $\uparrow t$ indicates the result of quoting t and the expression $\downarrow s$ indicates the result of unquoting s. The following axioms for the operators $\uparrow$ and $\downarrow$ formalize the relationship between terms and the corresponding name terms. They form an equality theory, called NT and first defined in [118], for the basic compositional encoding outlined above. Enhanced encodings can be obtained by adding axioms to this theory. NT states that there exist names of names (each term can be referenced n times, for any $n \geq 0$) and that the name of a compound term is obtained from the names of its components.

Definition 2 (Basic encoding NT). *Let* **NT** *be the following equality theory.*

- *For every constant or meta-constant c^n, $n \geq 0$,*
  $\uparrow c^n = c^{n+1}$.
- *For every function or predicate symbol f of arity k,*
  $\forall x_1 \ldots \forall x_k \ \uparrow (f(x_1, \ldots, x_k)) = [f^1, \uparrow x_1, \ldots, \uparrow x_k]$.
- *For every compound name term $[x_0, x_1, \ldots, x_k]$*
  $\forall x_0 \ldots \forall x_k \ \uparrow [x_0, x_1, \ldots, x_k] = [\uparrow x_0, \uparrow x_1, \ldots, \uparrow x_k]$.
- *For every term t $\downarrow\uparrow t = t$.*

The above set of axioms admits an associated convergent rewrite system UN. Then, a corresponding extended unification algorithm (E-unification algorithm) $UA(UN)$ can be defined, that deals with name terms in addition to usual terms. In [118] it is shown that:

Proposition 1 (Unification Algorithm for NT). *The E-unification algorithm $UA(UN)$ is sound for NT, terminates and converges.*

The standard semantics of the Horn clause language can be adapted, so as to include the naming device. Precisely, the technique of quotient universes by Jaffar et al. [119] can be used to this purpose.

Definition 3 (Quotient Universe). *Let R be a congruence relation. The* quotient universe *of U with respect to R, indicated as U/R, is the set of the equivalence classes of U under R, i.e., the partition given by R in U.*

By taking R as the finest congruence relation corresponding to UN (that always exists) we get the standard semantics of the Horn clause language [116], modulo the naming relation. The naming relation can be extended according to the application domain at hand, by adding new axioms to NT and by correspondingly extending UN and $UA(UN)$, provided that their nice formal properties

are preserved. What is important is that, as advocated in [37], the approach to meta-programming and the approach to naming become independent.

It is important to observe that, as shown in [36], any (ground or non-ground) encoding providing names for variables shows in an amalgamated language the same kind of problems emphasized in [35]. In fact, let P be the following definite program, x an object-level variable and Y a meta-variable:

$$p(x) \text{ :- } Y =\uparrow x,\ q(Y)$$
$$q(a^1).$$

Goal :-$p(a)$ succeeds by first instantiating Y to a^1 and then proving $q(a^1)$. In contrast, the goal :-$p(x)$ fails, as Y is instantiated to the name of x, say x^1, and subgoal $q(x^1)$ fails, x^1 and a^1 being distinct. Therefore, if choosing naming mechanisms providing names for variables, on the one hand terms can be inspected with respect to variable instantiation, on the other hand however important properties are lost.

A ground naming mechanism is used in [49] for providing a declarative semantics to the (conservative) amalgamation of language and meta-language in logic programming.

A naming mechanism where each well-formed expression can act as a name of itself is provided by the ambivalent logic AL of Jiang [73]. It is based on the assumption that each expression can be interpreted as a formula, as a term, as a function and as a predicate, where predicates and functions have free arity.

Unification must be extended accordingly, with the following results:

Theorem 1 (Termination of AL Unification Algorithm). *The unification algorithm for ambivalent logic terminates.*

Theorem 2 (Correctness of AL Unification Algorithm). *If the unification algorithm for ambivalent logic terminates successfully, then it provides an ambivalent unifier. If the algorithm halts with failure, then no ambivalent unifier exists.*

The limitation is that ambivalent unifiers are less general than traditional unifiers.

Theorem 3 (Properties of Resolution for AL). *Resolution is a sound and complete inference method for AL.*

Ambivalent logic has been used in [75] for proving correctness of the Vanilla meta-interpreter, also with respect to the (conservative) amalgamation of object language and meta-language. Let P be the object program, $\mathcal{L}_P$ the language of P, V_P the Vanilla meta-interpreter and $\mathcal{L}_{V_P}$ the language of V_P. Let M_P be the least Herbrand model of P, M_{V_P} be the least Herbrand model of V_P, and $M_{V_P \cup P}$ be the least Herbrand model of $V_P \cup P$. We have:

Theorem 4 (Properties of Vanilla Meta-Interpreter under AL). *For all (ground) A in $\mathcal{L}_{V_P}$, $demo(A) \in M_{V_P}$ iff $demo(A) \in M_{V_P \cup P}$;*
for all (ground) A in $\mathcal{L}_P$, $demo(A) \in M_P$ iff $demo(A) \in M_{V_P \cup P}$

A similar result is obtained by Martens and De Schreye in [120] and [50] for the class of *language independent programs.* They use a non-ground representation with overloading of symbols, so as the name of an atom is a term, identical to the atom itself. Language independent programs can be characterized as follows:

Proposition 2 (Language Independence). *Let P be a definite program. Then P is language independent iff for any definite goal G, all (SLD) computed answers for $P \cup G$ are ground.*

Actually however, the real practical interest lies in enhanced meta-interpreters. Martens and De Schreye extend their results to meta-interpreters without additional clauses, but with additional arguments. An additional argument can be for instance an explicit theory argument, or an argument denoting the proof tree. The amalgamation is still conservative, but more expressivity is achieved.

The approach to proving correctness of the Vanilla meta-interpreter proposed by Levi and Ramundo in [48] uses the S-semantics introduced by Falaschi et al. in [121]. In order to fill the gap between the procedural and declarative interpretations of definite programs, the S-least Herbrand model M_P^S of a program P contains not only ground atoms, but all atoms $Q(T)$ such that $t = x\,\theta$, where θ is the computed answer substitution for $P \cup \{\leftarrow Q(x)\}$. The S-semantics is obtained as a variation of the standard semantics of the Horn clause language. Levi and Ramundo [48] and Martens and De Schreye prove (independently) that $demo(p(t)) \in M_{V_P}^S$ iff $p(t) \in M_P^S$.

In the approach of Reflective Prolog, axiom schemata are defined at the meta-level, by means of a distinguished predicate *solve* and of a naming facility. Deduction is performed at any level where there are applicable axioms. This means, conclusions drawn in the basic theory are available (by implicit reflection) at the meta-level, and vice versa. The following definition of RSLD-resolution [65] (SLD-resolution with reflection) is independent of the naming mechanism, provided that a suitable unification algorithm is supplied.

Definition 4 (RSLD-resolution). *Let G be a definite goal $\leftarrow A_1, \ldots, A_k$, let A_m be the selected atom in G and let C be a definite clause.*

The goal $(\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$ is derived from G and C using mgu θ iff one of the following conditions holds:

i. C *is* $A \leftarrow B_1, \ldots, B_q$
   θ *is a mgu of* A_m *and* A
ii. C *is* $solve(\alpha) \leftarrow B_1, \ldots, B_q$
   $A_m \neq solve(\delta)$
   $\uparrow A_m = \alpha'$
   θ *is a mgu of* α' *and* α
iii. A_m *is* $solve(\alpha)$
   C *is* $A \leftarrow B_1, \ldots, B_q$
   $\downarrow \alpha = A'$
   θ *is a mgu of* A' *and* A

If the selected atom A_m is an object-level atom (e.g $p(a, b)$), it can be resolved in two ways. First, by using as usual the clauses defining the corresponding predicate (case (i)); for instance, if A_m is $p(a, b)$, by using the clauses defining the predicate p. Second, by using the clauses defining the predicate *solve* (case (ii), *upward reflection*) if the name $\uparrow A_m$ of A_m and α unify with mgu θ; for instance, referring to the NT naming relation defined above, we have $\uparrow p(a, b) = [p^1, a^1, b^1]$ and then a clause with conclusion $solve([p^1, v, w])$ can be used, with $\theta = \{v/a^1, w/b^1\}$.

If the selected atom A_m is $solve(\alpha)$ (e.g $solve([q^1, c^1, d^1])$), again it can be resolved in two ways. First, by using the clauses defining the predicate *solve* itself, similarly to any other goal (case (i)). Second, by using the clauses defining the predicate corresponding to the atom denoted by the argument α of *solve* (case (iii), *downward reflection*); for instance, if α is $[q^1, c^1, d^1]$ and thus $\downarrow \alpha = q(c, d)$, by using the clauses defining the predicate q can be used.

In the declarative semantics of Reflective Prolog, upward and downward reflection are modeled by means of axiom schemata called *reflection principles.* The Least Reflective Herbrand Model RM_P of program P is the Least Herbrand Model of the program itself, augmented by all possible instances of the reflection principles. RM_P is the least fixed point of a suitably modified version of operator T_P.

Theorem 5 (Properties of RSLD-Resolution). *RSLD-resolution is correct and complete w.r.t.* RM_P

8 Conclusions

In this paper we have discussed the meta-level approach to knowledge representation and reasoning that has its roots in the work of logicians and has played a fundamental role in computer science. We believe in fact that meta-programming and meta-reasoning are essential ingredients for building any complex application and system.

We have tried to illustrate to a broad audience what are the main principles meta-reasoning is based upon and in which way these principles have been applied in a variety of languages and systems. We have illustrated how sentences can be arguments of other sentences, by means of naming devices. We have distinguished between amalgamated and separated approaches, depending on whether the meta-expressions are defined in (an extension of) a given language, or in a separate language. We have shown that the different levels of knowledge can interact by reflection.

In our opinion, the choice of logic programming as a basis for meta-programming and meta-reasoning has several theoretical and practical advantages. From the theoretical point of view, all fundamental issues (including reflection) can be coped with on a strong semantic basis. In fact, the usual framework of first-order logic can be suitably modified and extended, as demonstrated by the various existing meta-logic languages. From the practical point of view, in

logic programming the meta-level mechanisms are understandable and easy-to-use and this has given rise to several successful applications. We have in fact tried (although necessarily shortly) to revise some of the important applications of meta-programming and meta-reasoning.

At the end of this survey, I wish to explicitly acknowledge the fundamental, deep and wide contribution that Robert A. Kowalski has given to this field. Robert A. Kowalski initiated meta-programming in logic programming, as well as many of its successful applications, including meta-interpreters, event calculus, logical agents. With his enthusiasm he has given constant encouragement to research in this field, and to researchers as well, including myself.

9 Acknowledgements

I wish to express my gratitude to Gaetano Aurelio Lanzarone, who has been the mentor of my research work on meta-reasoning and reflection. I gratefully acknowledge Pierangelo Dell'Acqua for his participation to this research and for the important contribution to the study of naming mechanisms and reflective resolution. I also wish to mention Jonas Barklund, for the many interesting discussions and the fruitful cooperation on these topics.

Many thanks are due to Luigia Carlucci Aiello, for her careful review of the paper, constructive criticism and useful advice. Thanks to Alessandro Provetti for his help. Thanks also to the anonymous referees, for their useful comments and suggestions. Any remaining errors or misconceptions are of course my entire responsibility.

References

1. Hill, P.M., Gallagher, J.: Meta-programming in logic programming. In Gabbay, D., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5, Oxford University Press (1995)
2. Barklund, J.: Metaprogramming in logic. In Kent, A., Williams, J.G., eds.: Encyclopedia of Computer Science and Technology. Volume 33. M. Dekker, New York (1995) 205–227
3. Lanzarone, G.A.: Metalogic programming. In Sessa, M.I., ed.: 1985–1995 Ten Years of Logic Programming in Italy. Palladio (1995) 29–70
4. Abramson, H., Rogers, M.H., eds.: Meta-Programming in Logic Programming, Cambridge, Mass., THE MIT Press (1989)
5. Bruynooghe, M., ed.: Proc. of the Second Workshop on Meta-Programming in Logic, Leuven (Belgium), Dept. of Comp. Sci., Katholieke Univ. Leuven (1990)
6. Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992)
7. Fribourg, L., Turini, F., eds.: Logic Program Synthesis and Transformation – Meta-Programming in Logic. LNCS 883, Springer-Verlag (1994)
8. Barklund, J., Costantini, S., van Harmelen, F., eds.: Proc. Workshop on Meta Programming and Metareasonong in Logic, post-JICSLP96 workshop, Bonn (Germany), UPMAIL technical Report No. 127 (Sept. 2, 1996), Computing Science Dept., Uppsala Univ. (1996)

9. Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995)
10. Maes, P., Nardi, D., eds.: Meta-Level Architectures and Reflection, Amsterdam, North-Holland (1988)
11. Kiczales, G., ed.: Meta-Level Architectures and Reflection, Proc. Of the First Intnl. Conf. Reflection 96, Xerox PARC (1996)
12. Cointe, A., ed.: Meta-Level Architectures and Reflection, Proc. Of the Second Intnl. Conf. Reflection 99. LNCS 1616, Berlin, Springer-Verlag (1999)
13. Smorinski, C.: The incompleteness theorem. In Barwise, J., ed.: Handbook of Mathematical Logic. North-Holland (1977) 821–865
14. Smullyan, R.: Diagonalization and Self-Reference. Oxford University Press (1994)
15. Kripke, S.A.: Semantical considerations on modal logic. In: Acta Philosophica Fennica. Volume 16. (1963) 493–574
16. Carlucci Aiello, L., Cialdea, M., Nardi, D., Schaerf, M.: Modal and meta languages: Consistency and expressiveness. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 243–266
17. Aiello, M., Weyhrauch, L.W.: Checking proofs in the metamathematics of first order logic. In: Proc. Fourth Intl. Joint Conf. on Artificial Intelligence, Tbilisi, Georgia, Morgan Kaufman Publishers (1975) 1–8
18. Bundy, A., Welham, B.: Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. Artificial Intelligence **16** (1981) 189–212
19. Weyhrauch, R.W.: Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence (1980) 133–70
20. Carlucci Aiello, L., Cecchi, C., Sartini, D.: Representation and use of metaknowledge. Proc. of the IEEE **74** (1986) 1304–1321
21. Carlucci Aiello, L., Levi, G.: The uses of metaknowledge in AI systems. In: Proc. European Conf. on Artificial Intelligence. (1984) 705–717
22. Davis, R., Buchanan, B.: Meta-level knowledge: Overview and applications. In: Procs. Fifth Int. Joint Conf. On Artificial Intelligence, Los Altos, Calif., Morgan Kaufmann (1977) 920–927
23. Maes, P.: Computational Reflection. PhD thesis, Vrije Universiteit Brussel, Faculteit Wetenschappen, Dienst Artificiele Intelligentie, Brussel (1986)
24. Genesereth, M.R.: Metalevel reasoning. In: Logic-87-8, Logic Group, Stanford University (1987)
25. Carlucci Aiello, L., Levi, G.: The uses of metaknowledge in AI systems. In Maes, P., Nardi, D., eds.: Meta-Level Architectures and Reflection. North-Holland, Amsterdam (1988) 243–254
26. Carlucci Aiello, L., Nardi, D., Schaerf, M.: Yet Another Solution to the Three Wisemen Puzzle. In Ras, Z.W., Saitta, L., eds.: Methodologies for Intelligent Systems 3: ISMIS-88, Elsevier Science Publishing (1988) 398–407
27. Carlucci Aiello, L., Nardi, D., Schaerf, M.: Reasoning about Knowledge and Ignorance. In: Proceedings of the International Conference on Fifth Generation Computer Systems 1988: FGCS-88, ICOT Press (1988) 618–627
28. Genesereth, M.R., Nilsson, J.: Logical Foundations of Artificial Intelligence. Morgan Kaufmann, Los Altos, California (1987)
29. Russell, S.J., Wefald, E.: Do the right thing: studies in limited rationality (Chapter 2: Metareasoning Architectures). The MIT Press (1991)
30. Carlucci Aiello, L., Cialdea, M., Nardi, D.: A meta level abstract description of diagnosis in Intelligent Tutoring Systems. In: Proceedings of the Sixth International PEG Conference, PEG-91. (1991) 437–442

31. Carlucci Aiello, L., Cialdea, M., Nardi, D.: Reasoning about Student Knowledge and Reasoning. Journal of Artificial Intelligence and Education **4** (1993) 397–413
32. Damásio, C., Nejdl, W., Pereira, L.M., Schroeder, M.: Model-based diagnosis preferences and strategies representation with logic meta-programming. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 267–308
33. Barklund, J., Costantini, S., Dell'Acqua, P., Lanzarone, G.A.: Reflection Principles in Computational Logic. Journal of Logic and Computation **10** (2000)
34. Barklund, J.: What is a meta-variable in Prolog? In Abramson, H., Rogers, M.H., eds.: Meta-Programming in Logic Programming. The MIT Press, Cambridge, Mass. (1989) 383–98
35. Hill, P.M., Lloyd, J.W.: Analysis of metaprograms. In Abramson, H., Rogers, M.H., eds.: Meta-Programming in Logic Programming, Cambridge, Mass., THE MIT Press (1988) 23–51
36. Barklund, J., Costantini, S., Dell'Acqua, P., Lanzarone, G.A.: Semantical properties of encodings in logic programming. In Lloyd, J.W., ed.: Logic Programming – Proc. 1995 Intl. Symp., Cambridge, Mass., MIT Press (1995) 288–302
37. van Harmelen, F.: Definable naming relations in meta-level systems. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992) 89–104
38. Cervesato, I., Rossi, G.: Logic meta-programming facilities in ′Log. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992) 148–161
39. Costantini, S.: Semantics of a metalogic programming language. Intl. Journal of Foundation of Computer Science **1** (1990)
40. Perlis, D.: Languages with self-reference I: foundations (or: we can have everything in first-order logic!). Artificial Intelligence **25** (1985) 301–322
41. Perlis, D.: Languages with self-reference II. Artificial Intelligence **34** (1988) 179–212
42. Konolige, K.: Reasoning by introspection. In Maes, P., Nardi, D., eds.: Meta-Level Architectures and Reflection. North-Holland, Amsterdam (1988) 61–74
43. Genesereth, M.R.: Introspective fidelity. In Maes, P., Nardi, D., eds.: Meta-Level Architectures and Reflection. North-Holland, Amsterdam (1988) 75–86
44. van Harmelen, F., Wielinga, B., Bredeweg, B., Schreiber, G., Karbach, W., Reinders, M., Voss, A., Akkermans, H., Bartsch-Spörl, B., Vinkhuyzen, E.: Knowledge-level reflection. In: Enhancing the Knowledge Engineering Process – Contributions from ESPRIT. Elsevier Science, Amsterdam, The Netherlands (1992) 175–204
45. Carlucci Aiello, L., Weyhrauch, R.W.: Using Meta-theoretic Reasoning to do Algebra. Volume 87 of Lecture Notes in Computer Science., Springer Verlag (1980) 1–13
46. Bowen, K.A., Kowalski, R.A.: Amalgamating language and metalanguage in logic programming. In Clark, K.L., T̃ärnlund, S.Å., eds.: Logic Programming. Academic Press, London (1982) 153–172
47. McCarthy, J.e.a.: (The LISP 1.5 Programmer's Manual)
48. Levi, G., Ramundo, D.: A formalization of metaprogramming for real. In Warren, D.S., ed.: Logic Programming - Procs. of the Tenth International Conference, Cambridge, Mass., The MIT Press (1993) 354–373
49. Subrahmanian, V.S.: Foundations of metalogic programming. In Abramson, H., Rogers, M.H., eds.: Meta-Programming in Logic Programming, Cambridge, Mass., The MIT Press (1988) 1–14

50. Martens, B., De Schreye, D.: Why untyped nonground metaprogramming is not (much of) a problem. J. Logic Programming **22** (1995)
51. Sterling, L., Shapiro, E.Y., eds.: The Art of Prolog. The MIT Press, Cambridge, Mass. (1986)
52. Kowalski, R.A.: Meta matters. invited presentation at Second Workshop on Meta-Programming in Logic META90 (1990)
53. Kowalski, R.A.: Problems and promises of computational logic. In Lloyd, J.W., ed.: Computational Logic. Springer-Verlag, Berlin (1990) 1–36
54. Smith, B.C.: Reflection and semantics in Lisp. Technical report, Xerox Parc ISL-5, Palo Alto (CA) (1984)
55. Lemmens, I., Braspenning, P.: A formal analysis of smithinsonian computational reflection. (In Cointe, P., ed.: Proc. Reflection '99) 135–137
56. Casaschi, G., Costantini, S., Lanzarone, G.A.: Realizzazione di un interprete riflessivo per clausole di Horn. In Mello, P., ed.: Gulp89, Proc. 4th Italian National Symp. on Logic Programming, Bologna (1989 (in italian)) 227–241
57. Friedman, D.P., Sobel, J.M.: An introduction to reflection-oriented programming. In Kiczales, G., ed.: Meta-Level Architectures and Reflection, Proc. Of the First Intnl. Conf. Reflection 96, Xerox PARC (1996)
58. Attardi, G., Simi, M.: Meta–level reasoning across viewpoints. In O'Shea, T., ed.: Proc. European Conf. on Artificial Intelligence, Amsterdam, North-Holland (1984) 315–325
59. Hill, P.M., Lloyd, J.W.: The Gödel Programming Language. The MIT Press, Cambridge, Mass. (1994)
60. Bowers, A.F., Gurr, C.: Towards fast and declarative meta-programming. In Apt, K.R., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 137–166
61. Giunchiglia, F., Cimatti, A.: Introspective metatheoretic reasoning. In Fribourg, L., Turini, F., eds.: Logic Program Synthesis and Transformation – Meta-Programming in Logic. LNCS 883 (1994) 425–439
62. Giunchiglia, F., Traverso, A.: A metatheory of a mechanized object theory. Artificial Intelligence **80** (1996) 197–241
63. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: how we can do without modal logics. Artificial Intelligence **65** (1994) 29–70
64. Costantini, S., Lanzarone, G.A.: A metalogic programming language. In Levi, G., Martelli, M., eds.: Proc. 6th Intl. Conf. on Logic Programming, Cambridge, Mass., The MIT Press (1989) 218–233
65. Costantini, S., Lanzarone, G.A.: A metalogic programming approach: language, semantics and applications. Int. J. of Experimental and Theoretical Artificial Intelligence **6** (1994) 239–287
66. Konolige, K.: An autoepistemic analysis of metalevel reasoning in logic programming. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992)
67. Dell'Acqua, P.: Development of the interpreter for a metalogic programming language. Degree thesis, Univ. degli Studi di Milano, Milano (1989 (in italian))
68. Maes, P.: Concepts and experiments in computational reflection. In: Proc. Of OOPSLA'87. ACM SIGPLAN NOTICES (1987) 147–155
69. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The Art of Meta-Object Protocol. The MIT Press (1991)
70. Malenfant, J., Lapalme, G., Vaucher, G.: Objvprolog: Metaclasses in logic. In: Proc. Of ECOOP'89, Cambridge Univ. Press (1990) 257–269

71. Malenfant, J., Lapalme, G., Vaucher, G.: Metaclasses for metaprogramming in prolog. In Bruynooghe, M., ed.: Proc. of the Second Workshop on Meta-Programming in Logic, Dept. of Comp. Sci., Katholieke Univ. Leuven (1990) 272–83
72. Stroud, R., Welch, I.: the evolution of a reflective java extension. LNCS 1616, Berlin, Springer-Verlag (1999)
73. Jiang, Y.J.: Ambivalent logic as the semantic basis of metalogic programming: I. In Van Hentenryck, P., ed.: Proc. 11th Intl. Conf. on Logic Programming, Cambridge, Mass., THE MIT Press (1994) 387–401
74. Kalsbeek, M., Jiang, Y.: A vademecum of ambivalent logic. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 27–56
75. Kalsbeek, M.: Correctness of the vanilla meta-interpreter and ambivalent syntax. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 3–26
76. Christiansen, H.: A complete resolution principle for logical meta-programming languages. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992) 205–234
77. Christiansen, H.: Efficient and complete demo predicates for definite clause languages. Datalogiske Skrifter, Technical Report 51, Dept. of Computer Science, Roskilde University (1994)
78. Brogi, A., Mancarella, P., Pedreschi, D., Turini, F.: Composition operators for logic theories. In Lloyd, J.W., ed.: Computational Logic. Springer-Verlag, Berlin (1990) 117–134
79. Brogi, A., Contiero, S.: Composing logic programs by meta-programming in Gödel. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 167–194
80. Brogi, A., Turini, F.: Meta-logic for program composition: Semantic issues. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 83–110
81. Barklund, J., Boberg, K., Dell'Acqua, P.: A basis for a multilevel metalogic programming language. In Fribourg, L., Turini, F., eds.: Logic Program Synthesis and Transformation – Meta-Programming in Logic. LNCS 883, Berlin, Springer-Verlag (1994) 262–275
82. Barklund, J., Boberg, K., Dell'Acqua, P., Veanes, M.: Meta-programming with theory systems. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 195–224
83. Shoham, Y., McDermott, D.: Temporal reasoning. (In C., S.S., ed.: Encyclopedia of Artificial Intelligence)
84. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
85. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence **4** (1969) 463–502
86. Kowalski, R.A.: Database updates in the event calculus. J. Logic Programming (1992) 121–146
87. Kowalski, R.A., Sadri, F.: The situation calculus and event calculus compared. In: Proc. 1994 Intl. Logic Programming Symp. (1994) 539–553
88. Kowalski, R.A., Sadri, F.: Reconciling the event calculus with the situation calculus. J. Logic Programming **31** (1997) 39–58
89. Provetti, A.: Hypothetical reasoning: From situation calculus to event calculus. Computational Intelligence Journal **12** (1996) 478–498

90. Díaz, O., Paton, N.: Stimuli and business policies as modeling constructs: their definition and validation through the event calculus. In: Proc. of CAiSE'97. (1997) 33–46
91. Sripada, S.: Efficient implementation of the event calculus for temporal database applications. In Lloyd, J.W., ed.: Proc. 12th Intl. Conf. on Logic Programming, Cambridge, Mass., The MIT Press (1995) 99–113
92. Pfenning, F.: The practice of logical frameworks. In Kirchner, H., ed.: Trees in Algebra and Programming - CAAP '96. LNCS 1059, Linkoping, Sweden, Springer–Verlag (1996) 119–134
93. Clavel, M.G., Eker, S., Lincoln, P., Meseguer, J.: Principles of Maude. (In Meseguer, J., ed.: Proc. First Intl Workshop on Rewriting Logic, volume 4 of Electronic Notes in Th. Comp. Sc.)
94. Clavel, M.G., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J.: Maude as a metalanguage. (In: Proc. Second Intl. Workshop on Rewriting Logic, volume 15 of Electronic Notes in Th. Comp. Sc.)
95. Clavel, M.G., Meseguer, J.: Axiomatizing reflective logics and languages. In Kiczales, G., ed.: Proc. Reflection '96, Xerox PARC (1996) 263–288
96. Costantini, S., Lanzarone, G.A., Sbarbaro, L.: A formal definition and a sound implementation of analogical reasoning in logic programming. Annals of Mathematics and Artificial Intelligence **14** (1995) 17–36
97. Costantini, S., Dell'Acqua, P., Lanzarone, G.A.: Reflective agents in metalogic programming. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992) 135–147
98. Martin, D.L., Cheyer, A.J., Moran, D.B.: The open agent architecture: a framework for building distributed software systems. Applied Artificial Intelligence **13(1–2)** (1999) 91–128
99. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Fikes, R., Sandewall, E., eds.: Proceedings of Knowledge Representation and Reasoning (KR&R-91), Morgan Kaufmann Publishers: San Mateo, CA (1991) 473–484
100. Rao, A.S., Georgeff, M.: BDI Agents: from theory to practice. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA (1995) 312–319
101. J., D., Subrahmanian, V., Pick, G.: Meta-agent programs. J. Logic Programming **45** (2000)
102. Kim, J.S., Kowalski, R.A.: An application of amalgamated logic to multi-agent belief. In Bruynooghe, M., ed.: Proc. of the Second Workshop on Meta-Programming in Logic, Dept. of Comp. Sci., Katholieke Univ. Leuven (1990) 272–83
103. Kim, J.S., Kowalski, R.A.: A metalogic programming approach to multi-agent knowledge and belief. In Lifschitz, V., ed.: Artificial Intelligence and Mathematical Theory of Computation, Academic Press (1991)
104. Kowalski, R.A., Sadri, F.: Towards a unified agent architecture that combines rationality with reactivity. In: Proc. International Workshop on Logic in Databases. LNCS 1154, Berlin, Springer-Verlag (1996)
105. Kowalski, R.A., Sadri, F.: From logic programming to multi-agent systems. (In: Annals of Mathematics and Artificial Intelligence) (to appear).
106. Dell'Acqua, P., Sadri, F., Toni, F.: Combining introspection and communication with rationality and reactivity in agents. In Dix, J., Cerro, F.D., Furbach, U., eds.: Logics in Artificial Intelligence. LNCS 1489, Berlin, Springer-Verlag (1998)
107. Fung, T.H., R. A. Kowalski, R.A.: The IFF proof procedure for abductive logic programming. J. Logic Programming **33** (1997) 151–165

108. Dell'Acqua, P., Sadri, F., Toni, F.: Communicating agents. In: Proc. International Workshop on Multi-Agent Systems in Logic Programming, in conjunction with ICLP'99, Las Cruces, New Mexico (1999)
109. Costantini, S.: Towards active logic programming. In Brogi, A., Hill, P., eds.: Proc. of 2nd International Workshop on Component-based Software Development in Computational Logic (COCL'99). PLI'99, Paris, France, http://www.di.unipi.it/ brogi/ ResearchActivity/COCL99/ proceedings/index.html (1999)
110. Gärdenfors, P.: Belief revision: a vademecum. In Pettorossi, A., ed.: Meta-Programming in Logic. LNCS 649, Berlin, Springer-Verlag (1992) 135–147
111. Gärdenfors, P., Roth, H.: Belief revision. In Gabbay, D., Hogger, C., Robinson, J., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 4. Clarendon Press (1995) 36–119
112. Dell'Acqua, P., Pereira, L.M.: Updating agents. (1999)
113. Lamma, E., Riguzzi, F., Pereira, L.M.: Agents learning in a three-valued logical setting. In Panayiotopoulos, A., ed.: Workshop on Machine Learning and Intelligent Agents, in conjunction with Machine Learning and Applications, Advanced Course on Artificial Intelligence (ACAI'99), Chania (Greece) (1999) (Also available at http://centria.di.fct.unl.pt/∼lmp/).
114. Brewka, G.: Declarative representation of revision strategies. In Baral, C., Truszczynski, M., eds.: NMR'2000, Proc. Of the 8th Intl. Workshop on Non-Monotonic Reasoning. (2000)
115. McCarthy, J.: First order theories of individual concepts and propositions. Machine Intelligence **9** (1979) 129–147
116. Lloyd, J.W.: Foundations of Logic Programming, Second Edition. Springer-Verlag, Berlin (1987)
117. Dell'Acqua, P.: Reflection principles in computational logic. PhD Thesis, Uppsala University, Uppsala (1998)
118. Dell'Acqua, P.: SLD–Resolution with reflection. PhL Thesis, Uppsala University, Uppsala (1995)
119. Jaffar, J., Lassez, J.L., Maher, M.J.: A theory of complete logic programs with equality. J. Logic Programming **3** (1984) 211–223
120. Martens, B., De Schreye, D.: Two semantics for definite meta-programs, using the non-ground representation. In Apt, K., Turini, F., eds.: Meta-Logics and Logic Programming. The MIT Press, Cambridge, Mass. (1995) 57–82
121. Falaschi, M.and Levi, G., Martelli, M., Palamidessi, C.: A new declarative semantics for logic languages. In Kowalski, R. A.and Bowen, K.A., ed.: Proc. 5th Intl. Conf. Symp. on Logic Programming, Cambridge, Mass., MIT Press (1988) 993–1005