Nonmonotonic Reasoning in FLORA-2*

Michael Kifer

Department of Computer Science
State University of New Your at Stony Brook
Stony Brook, NY 11794, U.S.A.
kifer@cs.stonybrook.edu

Abstract. FLORA-2 is an advanced knowledge representation system that integrates F-logic, HiLog, and Transaction Logic. In this paper we give an overview of the theoretical foundations of the system and of some of the aspects of nonmonotonic reasoning in FLORA-2. These include scoped default negation, behavioral inheritance, and nonmonotonicity that stems from database dynamics.

1 Introduction

FLORA-2 is a knowledge base engine and a complete environment for developing knowledge-intensive applications. It integrates F-logic with other novel formalisms such as HiLog and Transaction Logic. FLORA-2 is freely available on the Internet[1] and is in use by a growing research community. Many of the features of FLORA-2 have been adopted by the recently proposed languages in the Semantic Web Services domain: WSML-Rule[2] and SWSL-Rules.[3]

One of the main foundational ingredients of FLORA-2, F-logic [20], extends classical predicate calculus with the concepts of objects, classes, and types, which are adapted from object-oriented programming. In this way, F-logic integrates the paradigms of logic programming and deductive databases with the object-oriented programming paradigm. Most of the applications of F-logic have been in intelligent information systems, but more recently it has been used to represent ontologies and other forms of Semantic Web reasoning [14, 12, 27, 1, 11, 2, 19].

HiLog [8] is an extension of the standard predicate calculus with higher-order syntax. Yet the semantics of HiLog remains first-order and tractable. In FLORA-2, HiLog is the basis for simple and natural querying of term structures and for reification (or objectification) of logical statements, which is an important requirement for a Semantic Web language. Transaction Logic [6] provides the basis for declarative programming of "procedural knowledge" that is often embedded in intelligent agents or Semantic Web services.

In this paper we first survey the main features of FLORA-2 and then discuss three forms of nonmonotonic reasoning provided by the system.

* This work was supported in part by NSF grant CCR-0311512 and by U.S. Army Medical Research Institute under a subcontract through Brookhaven National Lab.

[1] http://flora.sourceforge.net/

[2] http://www.w3.org/Submission/WSML/

[3] http://www.daml.org/services/swsl-rules/1.0/

2 Overview of F-logic

F-logic extends predicate calculus both syntactically and semantically. It has a monotonic logical entailment relationship, and its proof theory is sound and complete with respect to the semantics. F-logic comes in two flavors: the first-order flavor and the logic programming flavor. The first-order flavor of F-logic can be viewed as a syntactic variant of classical logic [20]. The logic programming flavor uses a subset of the syntax of F-logic, but gives it a different, non-first-order semantics by interpreting the negation operator as negation-as-failure.

The relationship between the first-order variant of F-logic and its logic programming variant is similar to the relationship between predicate calculus and standard logic programming [23]: object-oriented logic programming is built on the rule-based subset of F-logic by adding the appropriate non-monotonic extensions [32, 33, 24]. These extensions are intended to capture the semantics of negation-as-failure (like in standard logic programming [28]) and the semantics of multiple inheritance with overriding (which is not found in standard logic programming).

F-logic uses first-order variable-free terms to represent *object identity* (abbr., OID); for instance, `John` and `father(Mary)` are possible Ids of objects. Objects can have attributes. For instance,

```
Mary[spouse -> John, children -> {Alice,Nancy}].
Mary[children -> Jack].
```

Such formulas are called F-logic *molecules*. The first formula says that object `Mary` has an attribute `spouse` whose value is the OID `John`. It also says that the attribute `children` is set-valued and its value is a set that *contains* two OIDs: `Alice` and `Nancy`. We emphasize "contains" because sets do not need to be specified all at once. For instance, the second formula above says that `Mary` has an additional child, `Jack`.

In earlier versions of F-logic, set-valued attributes were denoted with `->>` instead of `->`. However, subsequently the syntax was modernized and simplified. Instead of using different arrows, cardinality constraints (to be explained shortly) were introduced to indicate that an attribute is single-valued.

While some attributes of an object are specified explicitly, as facts, other attributes can be defined using deductive rules. For instance, we can derive `John[children -> {Alice,Nancy,Jack}]` using the following deductive rule:

```
?X[children -> {?C}] :- ?Y[spouse -> ?X, children -> {?C}].
```

In the new and simplified syntax, alphanumeric symbols prefixed with the ?-sign denote variables and unprefixed alphanumeric symbols denote constants (i.e., OIDs). The earlier versions of FLORA-2 used Prolog conventions where variables were capitalized alphanumeric symbols.

F-logic objects can also have *methods*, which are functions that take arguments. For instance,

```
John[grade(cs305,fall2004) -> 100, courses(fall2004) -> {cs305,cs306}].
```

says that `John` has a method, `grade`, whose value on the arguments `cs305` (a course identifier) and `fall2004` (a semester designation) is `100`; it also has a set-valued method `courses`, whose value on the argument `fall2004` is a set of OIDs that contains course identifiers `cs305` and `cs306`. Like attributes, methods can be defined using deductive rules.

The F-logic syntax for *class membership* is `John:student` and for *subclass relationship* it is `student::person`. Classes are treated as objects and it is possible for the same object to play the role of a class in one formula and of an object in another. For instance, in the formula `student:class`, the symbol `student` plays the role of an object, while in `student::person` it appears in the role of a class.

F-logic also provides means for specifying schema information through *signature* formulas. For instance, `person[spouse {0:1}=> person, name {0:1}=> string, child=> person]` is a signature formula that says that class `person` has three attributes: single-valued attributes `spouse` and `name` (single-valuedness is indicated by the cardinality constraint `0:1`) and a set-valued attribute `child`. It further says that the first attribute returns objects of type `person`, the second of type `string`, and the last returns sets of objects such that each object in the set is of type `person`.

3 HiLog and Meta-Information

F-logic provides simple and natural means for exploring the structure of object data. Both the schema information associated with classes and the structure of individual objects can be queried by simply putting variables in the appropriate syntactic positions. For instance, to find the set-valued methods that are defined in the *schema* of class `student` and return objects of type `person`, one can ask the following query:

```
?- student[?M=>person].
```

The next query is about the type of the results of the attribute `name` in class `student`. This query also returns all the superclasses of class `student`.

```
?- student::?C and student[name=>?T].
```

The above are schema-level meta-queries: they involve the subclass relationship and the type information. One can also pose meta-queries that involve object data (rather than schema). The following queries return the methods that have a known value for the object `John`:

```
?- John[?Meth->?SomeValue].
```

However, the meta-query facilities of F-logic are not complete. For instance, there is no way in such queries to separate method names from their arguments. Thus, if we had a fact of the form

```
John[age(2005)-> 20].
```

then the first of the above queries will bind `?Meth` to `age(2005)`—we cannot separate `age` from `2005`.

This is where HiLog [8] comes into picture. In HiLog, second-order syntax is allowed and so variables can appear in positions of function and predicate symbols. For instance, queries such as

```
?- person[?M(?Arg) -> ?SomeValue].
?- person[?M(?Arg) => integer].
```

are allowed and `?M` would be bound to `age` and, possibly, to other values as well. The semantics for this second-order syntax is first order, however. Roughly it means that variables get bound not to the extensional values of the symbols (i.e., the actual relations that are used to interpret the function and predicate symbols), but to the symbols themselves. Details of these semantics can be found in [8].

HiLog does not stop at allowing variables over function and predicate symbols—it also permits them over atomic formulas. For instance, the following query is legal and will succeed with `?X` bound to `p(a)`.

```
p(a).
q(p(a)).
?- q(?X), ?X.
```

What happens here is that the proposition `p(a)` is *reified* (made into an object) and so it can be bound to a variable. HiLog's reification of atomic formulas can be extended to arbitrary quantifier-free formula of the rule-based subset of HiLog and F-logic, and this has been done in [31, 19]. For instance, one can say that John believes that Mary likes Sally as follows:

$$\texttt{John}[\texttt{believes} \texttt{->} \$\{\texttt{Mary}[\texttt{likes} \texttt{->} \texttt{Sally}]\}]. \tag{1}$$

Here ${...} is the syntax that FLORA-2 uses to denote reified statements. An example of a more complicated reified statement is this:

$$\texttt{John}[\texttt{believes} \texttt{->} \$\{\texttt{Bob}[\texttt{likes} \texttt{->} ?\texttt{X}]\ :-\ \texttt{Mary}[\texttt{likes} \texttt{->} ?\texttt{X}]\}]. \tag{2}$$

This sentence reifies a rule (not just a fact) and states that John also believes that Bob likes anybody who is liked by Mary. Combined with the previous statement that John believes that Mary likes Sally, one would expect that John would also believe that Bob likes Sally. However, we cannot conclude this just yet because we do not know that John is a rational being who applies modus ponens in his daily life. But this rational principle can be stated rather easily:

$$\begin{array}{l}\texttt{John}[\texttt{believes -> } ?\texttt{A}]\ :- \\ \qquad\qquad \texttt{John}[\texttt{believes -> } \{\$\{?\texttt{Head} :- ?\texttt{Body}\},\ ?\texttt{Body}\}].\end{array} \tag{3}$$

4 Transaction Logic

Knowledge intensive applications, such as those in semantic Web services and intelligent agents, often require primitives for modifying the underlying state of the system. Prolog provides the well-known `assert` and `retract` operators, which are non-logical and are therefore widely viewed as problematic. Various attempts to formalize updates in a logic programming language have had only a limited success (*e.g.*, [21, 26, 22]). A detailed discussion of this subject appears in [5, 6]. Some of the most serious problems with these approaches is that they impose special programming styles (which is a significant burden) and that they do not support subroutines — one of the most fundamental aspects of any programming language.

Transaction Logic [4–6] is a comprehensive solution to the problem of updates in logic programming. This solution has none of the above drawbacks and it fits nicely with the traditional theory of logic programming. The use of Transaction Logic has been illustrated on a vast variety of applications, ranging from databases to robot action planning to reasoning about actions to workflow analysis and Web services [5, 7, 10, 19].

An important aspect of the update semantics of Transaction Logic is that updates are *atomic*, which means that an update transaction executes in its entirety or not at all. In contrast, in Prolog, if a post-condition of a state-changing predicate is false, the execution "fails" but the changes made by `assert` and `retract` would stay and the knowledge base is left in a inconsistent state. This property is responsible for many complications in Prolog programming. This and related problems are rectified by Transaction Logic semantics.

FLORA-2 integrates F-logic and Transaction Logic along the lines of [18] with certain refinements that distinguish queries from transactions and thus enable a number of compile-time checks. In Transaction Logic, both actions (transactions) and queries are represented as predicates. In FLORA-2, transactions are expressed as object methods that are prefixed with the special symbol "%".

The following program is an implementation of a block-stacking robot in FLORA-2. Here, the action `stack` is defined as a Boolean method of the robot.

```
?R[%stack(0,?X)]   :- ?R:robot.
?R[%stack(?N,?X)]  :- ?R:robot, ?N > 0,
                      ?Y[%move(?X)], ?R[%stack(?N-1,?Y)].
?Y[%move(?X)]      :- ?Y:block, ?Y[clear], ?X[clear], ?X[widerThen(?Y)],
                      btdelete{?Y[on->?Z]}, btinsert{?Z[clear]},
                      btinsert{?Y[on->?X]}, btdelete{?X[clear]}.
```

The primitives `btdelete` and `btinsert` are FLORA-2's implementations of the insert and delete operators with the Transaction Logic semantics. Informally, the above rules say that to stack a pyramid of `N` blocks on top of block `?X`, the robot must find a block `?Y`, move it onto `?X`, and then stack `N-1` blocks on top of `?Y`. To move `?Y` onto `?X`, both blocks must be "clear" (*i.e.*, with no other block sitting on top of them), and `?X` must be wider than `?Y`. If these conditions are met, the database will be updated accordingly. If any of the conditions fails, it means that the current attempted execution is not a valid try and another attempt will be made. If no valid execution is found, the transaction fails and no changes will be made to the database.

A simple-minded translation of the above program into Prolog is incorrect, since such a program might leave the database in an inconsistent state. A *correct* version of the above FLORA-2 program in Prolog is more complicated and much less natural.

5 Scoped Default Negation

Closed world assumption [25] is an inference rule for negative information. It states that in the absence of a positive proof that a fact, `F`, is true one must conclude that `not F` is true. Negation that obeys such an inference rule is *not* classical and is often called *default negation*.[4]

Various forms of the closed-world assumption (CWA) have been successfully used in database and logic programing applications for over thirty years now and vast experience has been accumulated with the use of this paradigm in knowledge representation [9, 28, 16]. In contrast, classical logic is based on the open-world assumption (OWA), and this has been the *sine qua non* in, for example, the description logic community. Each community allowed the other to continue to believe in its respective heresy until the Semantic Web came along.

The advent of the Semantic Web caused heated discussions about the one and only kind of negation that is suitable for this emerging global knowledge base (see, e.g., `http://robustai.net/papers/Monotonic_Reasoning_on_the_Semantic_Web.html` for a compendium). The main argument against closed-world assumption goes like this. The Web is practically infinite and failure to derive some fact from the currently obtained information does not warrant the conclusion that this fact is false. Nevertheless, thirty years of experience in practical knowledge representation cannot be dismissed lightly and even the proponents of the open-world assumption are beginning to realize that. One idea that is beginning to take hold is that CWA is acceptable—even in the Web environment—as long as the *scope* of the closure is made explicit and concrete [17]. A simplified form of this idea was recently added to the N3 rule language [3] (whose author, Tim Berners-Lee, previously resisted the use of default negation).

Scoped default negation was introduced in FLORA-2 as part of its innovative architecture for knowledge base modules. It is related to (but is different from) the so called *local closed world assumption* [13]. In FLORA-2, a module is a container for a concrete knowledge base (or a part of it). Modules isolate the different parts of a knowledge base and provide a clean interface by which these parts can interact. Modules can be created dynamically, associated with

[4] Some researchers also sometimes call this type of negation *negation-as-failure*. We avoid this terminology because negation-as-failure was originally used to denote a specific proof strategy used in the Prolog language.

knowledge bases on the fly, and they support a very powerful form of encapsulation. For our discussion, the relevant aspect of the FLORA-2 modules is that they provide a simple and natural mechanism for scoped default negation.

Consider the statements (1), (2), and (3) about John's beliefs from the end of Section 3. To use these statements, one must insert them into a module, let us call it `johnmodule`, by, for instance, loading the file that contains these statements into the module. To query the information about John's beliefs one would now pose queries such as

```
?- John[believes -> ${Mary[likes->Sally]}]@johnmodule.
```

Similarly, to inquire whether John *does not* believe that Bob is Sally's husband one would ask the query

$$?-\,\texttt{not John}[\texttt{believes}\,\texttt{->}\$\{\texttt{Sally}[\texttt{spouse}\,\texttt{->Bob}]\}]@\texttt{johnmodule}. \qquad (4)$$

The scope of the above query is limited to the module `johnmodule` only. If it cannot be derived that `John[believes->${Sally[spouse->Bob]}]` is true from the knowledge base residing in the module then (and only then) the answer will be "Yes." The answer to (4) will remain the same even if some other module asserts that `John[believes->${Sally[spouse->Bob]}]` because the scope of the default negation in the query is limited to the module `johnmodule`. It is, however, possible to ask unrestricted negative queries by placing a variable in the module position:

```
?- not John[believes -> ${Sally[spouse->Bob]}]@?Mod.
```

This query returns "Yes" iff John is not known to believe that `Sally[spouse->Bob]` is true in every module (that is registered with the system).

The semantics of FLORA-2 modules is very simple. The attribute and method names of the formulas that are loaded into a module, such as `johnmodule`, are uniquified so that the same attribute name in the program will be given different and unique *real* names in different modules. For instance, a formula such as `John[believes->abc]` might become `John[believes#foo->abc]` in module `foo` and `John[believes#bar->abc]` in module `bar`. Due to this transformation, the query (4) turns into the following query in the actual knowledge base:

```
?- not John[believes#johnmodule-> ${Sally[spouse#johnmodule->Bob]}].
```

Since other modules cannot have facts or rules whose heads have the form `...[believes#johnmodule->...]`, the answer "Yes" or "No" depends only on the information stored in module `johnmodule`.

6 Nonmonotonic Inheritance

F-logic supports both *structural* and *behavioral* inheritance. The former refers to inheritance of method types from superclasses to their subclasses and the latter deals with inheritance of method definitions from superclasses to subclasses.

Structural inheritance is defined by very simple inference rules:

If `subcl::cl`, `cl[attr*=>type]` **then** `subcl[attr*=>type]`
If `obj:cl`, `cl[attr*=>type]` **then** `obj[attr=>type]`

The statement `cl[attr*=>type]` above says that `attr` is an *inheritable* attribute, which means that both its type and value are inheritable by the subclasses and members of class `cl`. Inheritability of the type of an attribute is indicated with the star attached to the arrow: `*=>`. In all of our previous examples we have been dealing with *non-inheritable* attributes, which were designated with star-less arrows. Note that when the type of an attribute is inherited to a subclass it remains inheritable. However, when it is inherited to a member of the class it is no longer inheritable.

Type inheritance, as defined by the above rules, is *monotonic* and thus is peripheral to the subject of this paper. Behavioral inheritance is more complex. To get a flavour of behavioral inheritance, consider the following knowledge base:

```
royalElephant::elephant.
clyde:royalElephant.
elephant[color*->grey].
royalElephant[color*->white].
```

As with type definitions, a star attached to the arrow `*->` indicates inheritability. For instance, `color` is an inheritable attribute in classes `elephant` and `royalElephant`. The inference rule that guides behavioral inheritance can informally be stated as follows. If `obj` is an object and `cl` is a class, then

`obj:cl, cl[attr*->value]` should imply `obj[attr->value]`

unless the inheritance is overwritten by a more specific class. The meaning of the exception here is that the knowledge base should *not* imply the formula `obj[attr->value]` if there is an intermediate class, `cl′`, which overrides the inheritance, i.e., if `obj : cl′`, `cl′ :: cl` are true and `cl′[attr*->value′]` (for some `value′` $\neq$ `value`) is defined explicitly.[5] A similar exception exists in case of multiple inheritance conflicts. Note that inheritable attributes become non-inheritable after they are inherited by class members. In the above case, inheritance of the color *grey* is overwritten by the color *white* and so `clyde[color->white]` is derived by the rule of inheritance.

This type of inheritance is clearly nonmonotonic. For instance, if in the above example we add the fact `clyde[color->yellow]` to the knowledge base then `clyde[color->white]` is no longer inferred by inheritance (the inference is said to be overwritten).

Model-theoretic semantics for nonmonotonic inference by inheritance is rather subtle and has eluded researchers for many years. Although the above informal rules for inference by inheritance seem natural, there are subtle problems when behavioral inheritance is used together with deductive rules. To understand the problem, consider the following example:

[5] The notion of an explicit definition seems obvious at first but, in fact, is quite subtle. Details can be found in [29].

```
cl[attr*=>v1].
subcl::cl.
obj:subcl.
subcl[attr*=>v2] :- obj[attr->v1].
```

If we apply the rule of inheritance to this knowledge base, then `obj[attr->v1]` should be inherited, since no overriding takes place. However, once `obj[attr->v1]` is derived by inheritance, `subcl[attr*=>v2]` can be derived by deduction—and now we have a chicken-and-egg problem. Since `subcl` is a more specific superclass of `obj`, the derivation of `subcl[attr*=>v2]` appears to override the earlier inheritance of `obj[attr->v1]`. But this, in turn, undermines the very reason for deriving `subcl[attr*=>v2]`. The above is only one of several suspicious derivation patterns that arise due to interaction of inheritance and deduction. The original solution reported in [20] was not model-theoretic and was problematic in several other respects as well. A satisfactory and completely model-theoretic solution was proposed in [29, 30].

7 Database Dynamics and Nonmonotonicity

In [5], a generalization of the perfect-model semantics was defined for Transaction Logic programs with negation in the rule body. This semantics was implemented in FLORA-2 only partially, with negation applicable only to non-transactional formulas in the rule body. For instance, the following transaction logs all unauthorised accesses to any given resource, and default negation, `not`, is applied only to a query (not an action that has a side effect):

$$\begin{array}{l} ?\mathtt{Rsrc}[\%\mathtt{recordUnauthAaccess}(?\mathtt{Agent})] :- \\ \qquad\qquad \mathtt{not}\ ?\mathtt{Rsrc}[\mathtt{eligible}\ \texttt{->}\ ?\mathtt{Agent}], \\ \qquad\qquad \mathtt{insert}\{\mathtt{unAuthLog}(?\mathtt{Agent}, ?\mathtt{Rsrc})\}. \end{array} \tag{5}$$

Nonmonotonicity comes into play here in a somewhat different sense than in standard logic programming (ignoring the non-logical `assert` and `retract`). In the standard case, nonmonotonicity means that certain formulas that were derivable in a database state, `s1`, will not be derivable in the state obtained from `s1` by adding more facts. In the above example, however, no inference that was enabled by rule (5) can become invalidated by adding more facts, since this rule is not a statement about the initial database state.

In our example, nonmonotonicity reveals itself in a different way: the transaction of the form `?- mySecrets[%recordUnauthAaccess(John)]` can be executable in the initial state (if `mySecrets[eligible->John]` is not derivable) and non-executable in the state obtained by adding `mySecrets[eligible->John]` to the original state.

This kind of non-monotonicity can be stated formally in the logic as: there are database states $\mathbf{D}$ and $\mathbf{D'}$, where $\mathbf{D} \subseteq \mathbf{D'}$, such that

$$\mathbf{D}\texttt{---}\ \models\ \mathtt{mySecrets}[\%\mathtt{recordUnauthAaccess}(\mathtt{John})]$$

but

$$\mathbf{D}' \models \texttt{not}\,\Diamond\,\texttt{mySecrets}[\%\texttt{recordUnauthAaccess}(\texttt{John})]$$

The first statement above is called *executional entailment*; it means that there is a sequence of states, beginning with the given state **D**, which represents an execution path of the transaction `mySecrets[%recordUnauthAaccess(John)]`. The second statement says that there is no execution path, which starts at state $\mathbf{D}'$, for the transaction `mySecrets[%recordUnauthAaccess(John)]`.

Another instance of nonmonotonic behavior that is different from the classical cases occurs when enlarging the initial state of transaction execution leads to a possible elimination of facts in the final state of transaction execution. To illustrate this, consider a slightly modified version of transaction (5):

```
?Rsrc[%recordUnauthAaccess(?Agent)] :-
      not ?Rsrc[eligible->?Agent], insert{unAuthLog(?Agent,?Rsrc)}.
?Rsrc[%recordUnauthAaccess(?Agent)] :- ?Rsrc[eligible->?Agent].
```

In this case, the transaction `?- mySecrets[%recordUnauthAaccess(John)]` can be executed regardless of whether John is eligible or not. If John is not eligible then `unAuthLog(John,mySecrets)` becomes true in the final state of the execution of this transaction. If John is already eligible then nothing changes. Now, if we add the fact that John *is* eligible to access `mySecrets` then the transaction executes without changing the state. Therefore, `unAuthLog(John,mySecrets)` is no longer derivable in the final state. Thus, enlarging the initial state of transaction execution does not necessarily lead to a monotonic enlargement of the final state.

8 Conclusion

This paper presents an overview of the formal foundations of the FLORA-2 system with a focus on various forms of nonmonotonic reasoning in the system. Three aspects have been considered: scoped default negation, behavioral inheritance, and nonmonotonicity that stems from database dynamics. Scoped negation is believed to be the right kind of negation for the Semantic Web. Behavioral inheritance is an important concept in object-oriented modeling; in FLORA-2 it has been extended to work correctly (from the semantic point of view) in a rule-based system. Finally, we discussed database dynamics in FLORA-2 and have shown how it can lead to of nonmonotonic behavior.

References

1. J. Angele and G. Lausen. Ontologies in F-logic. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*, pages 29–50. Springer Verlag, Berlin, Germany, 2004.
2. D. Berardi, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, J. Su, and S. Tabet. SWSL: Semantic Web Services Language. Technical report, Semantic Web Services Initiative, April 2005. http://www.daml.org/services/swsl/.

3. T. Berners-Lee. Primer: Getting into RDF & Semantic Web using N3, 2004. http://www.w3.org/2000/10/swap/Primer.html.
4. A. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Comput. Sci.*, 133:205–265, October 1994.
5. A. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.
6. A. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.
7. A. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In [15]. Springer-Verlag, 1998.
8. W. Chen, M. Kifer, and D. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
9. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322. Plenum Press, 1978.
10. H. Davulcu, M. Kifer, C. Ramakrishnan, and I. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.
11. J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, and D. Fensel. The WSML family of representation languages. Technical report, DERI, March 2005. http://www.wsmo.org/TR/d16/d16.1/.
12. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL'98 - The Query Languages Workshop*, December 1998.
13. O. Etzioni, K. Golden, and D. Weld. Sound and effcient closed-world reasoning for planning Artificial Intelligence. *Artificial Intelligence*, 89(1-2):113–148, 1997.
14. D. Fensel, M. Erdmann, and R. Studer. OntoBroker: How to make the WWW intelligent. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1998.
15. B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
16. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.
17. S. Hawke, S. Tabet, and C. de Sainte Marie. Rule Language Standardization: Report from the W3C Workshop on Rule Languages for Interoperability, May 2005. http://www.w3.org/2004/12/rules-ws/report/.
18. M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Int'l Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Int'l Conference on Deductive and Object-Oriented databases.
19. M. Kifer, R. Lara, A. Polleres, and C. Zhao. A logical framework for web service discovery. In *ISWC 2004 Semantic Web Services Workshop*. CEUR Workshop Proceedings, November 2004.
20. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
21. R. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.

22. G. Lausen and B. Ludäscher. Updates by reasoning about states. In *2-nd International East/West Database Workshop*, Klagenfurt, Austria, September 1994.
23. J. W. Lloyd. *Foundations of Logic Programming (Second, extended edition)*. Springer series in symbolic computation. Springer-Verlag, New York, 1987.
24. Ontoprise, GmbH. OntoBroker Manual. http://www.ontoprise.com/.
25. R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, 1978.
26. R. Reiter. Formalizing database evolution in the situation calculus. In *Conference on Fifth Generation Computer Systems*, 1992.
27. S. Staab and A. Maedche. Knowledge portals: Ontologies at work. *The AI Magazine*, 22(2):63–75, 2000.
28. A. Van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, July 1991.
29. G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE)*, October 2002.
30. G. Yang and M. Kifer. Inheritance and rules in object-oriented semantic Web languages. In *Rules and Rule Markup Languages for the Semantic Web (RuleML03)*, volume 2876 of *Lecture Notes in Computer Science*. Springer Verlag, November 2003.
31. G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the Semantic Web. *Journal on Data Semantics, LNCS 2800*, 1:69–98, September 2003.
32. G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)*, November 2003.
33. G. Yang, M. Kifer, and C. Zhao. FLORA-2: User's Manual. http://flora.sourceforge.net/documentation.php, March 2005.