5th International Conference on Deductive and Object-Oriented Databases (DOOD'97), December 8-12, 1997, Montreux, Switzerland.
Springer LNCS 1341, pp. 320–336.

Well-Founded Semantics for Deductive Object-Oriented Database Languages

Wolfgang May* Bertram Ludäscher Georg Lausen

Institut für Informatik, Universität Freiburg, Germany
{may,ludaesch,lausen}@informatik.uni-freiburg.de

Abstract. We present a well-founded semantics for deductive object-oriented database (*dood*) languages by applying the alternating-fixpoint characterization of the well-founded model to them. In order to compute the state sequence, states are explicitly integrated by making them first-class citizens of the underlying language. The concept is applied to FLORID, an implementation of F-Logic, previously supporting only inflationary negation. Using our approach, well-founded models of F-Logic programs can be computed.
The method is also applicable to arbitrary dood languages which provide a sufficiently flexible syntax and semantics. Given an implementation of the underlying database language, any program given in this language can be evaluated wrt. the well-founded semantics.

1 Introduction

The well-founded semantics (WFS) [VGRS88] is generally accepted as a sceptical "well-behaved"[2] semantics for logic programs with negation. It assigns a unique, three-valued model $\mathcal{W}(P)$ to every program P. The third truth-value *undefined* is assigned to atoms which depend negatively on themselves and for which no independent "well-founded" derivation exists. Although several relational database systems now support WFS, this is not the case for dood systems. Existing dood systems are limited to inflationary or stratified semantics and may benefit from a WFS for the following reasons:

- In relational languages the notion of stratification is based on explicit dependencies between relation symbols. For object-oriented (OO) frameworks, those dependencies are conceptually more involved due to value inheritance, a dynamic class hierarchy, and higher-order features like variables at method or class positions.
- Since stratified negation is less expressive than well-founded negation, certain concepts cannot be expressed in stratified semantics, most notably the notion of *deep equality* [AdB95] (cf. Section 5) in presence of set values, which is crucial for OO-systems. Another example are argumentation frameworks [Dun95] – which inherently require the WFS.

* Supported by grant no. GRK 184/1-97 of the Deutsche Forschungsgemeinschaft.
[2] Dix [Dix95] formally defines this notion using certain abstract properties of semantics.

In this paper, we show how WFS can be applied to dood languages, using the well-known alternating-fixpoint characterization (AFP) [VGRS88, VG93]. For this, analogous to reification in relational database languages, a notion of states is incorporated into the modeling, and the program is transformed accordingly. Evaluating the *transformed* program with the *original* semantics of the underlying framework yields the WFS of the *original* program.

The paper is structured as follows: the introduction is completed with an overview of related work and some notational conventions. Section 2 exhibits some problems of stratified semantics in the OO paradigm. In Section 3, the WFS and its alternating-fixpoint characterization are reviewed and a formalization of the alternating fixpoint characterization for the OO paradigm is given. In Section 4, the approach is instantiated for F-Logic. Section 5 illustrates the concept and its application by examples. Section 6 closes with some concluding remarks.

Related Work. To our knowledge, none of the existing dood languages supports WFS: For several early logics introducing OO-features like object id's, types/classes, or set-values, e.g. O-Logic [Mai86] and its relatives presented in [KW93] or [CW89] (C-Logic), or ILOG [HY90], the semantics of programs is reduced to the semantics of first-order logic programs via program transformations, but these approaches lack some typical OO-features. COL (complex object language) [AG91] extends Datalog by structured values and set constructors, without providing a class hierarchy or inheritance. Its semantics is given in terms of minimal models and stratification. IQL [AK92] provides oid's, set and tuple types corresponding to method applications, and *types* corresponding to classes where type inheritance corresponds to a subclass hierarchy. There is no value inheritance. IQL is evaluated bottom-up using an inflationary fixpoint operator. LOGRES [CCCR+90] additionally supports multisets, (multiple) value inheritance, and integrity constraints, coming with an inflationary or stratified semantics. ROL [Liu96] is based on the standard notions of objects, methods, and classes. Regarding classes, there is only structural inheritance, but no value inheritance. Thus, stratification is possible in ROL. Stratified programs are evaluated wrt. a minimal-model semantics. The semantics of ROLL [BPF+94] is defined by a mapping to an internal Datalog representation which is then input to an SLDNF resolution proof-procedure. Noodle [MR93] is based on HiLog and comes with modularly stratified semantics. In [KLW95], F-Logic is defined with a minimal-model and a perfect model semantics. The FLORID implementation provides an inflationary semantics with user-defined stratification.

Summarizing, although a number of dood languages have been developed, to our knowledge, none of them provides WFS for handling negation.

Notation. An Object-oriented model is represented by three types of atoms, i.e., method applications, class membership, and the subclass relation. In order to obtain a uniform notation, we will use F-Logic syntax (cf. Section 4) throughout this paper: o[m→v] denotes that application of method m to object o results in the value v; o:c denotes that o is a member of class c; and c::d denotes that c is a subclass of d. We will use capital letters for variables.

2 Stratified Semantics

In the relational context, stratification is defined using a dependency graph over relation symbols: a relation p depends positively/negatively on another relation q if there is a rule with p occurring in the head and q occurring positively/negatively in the body. The dependency graph is given as (V, E) s.t. V is the set of relation symbols and E contains a "positive" edge $y \rightarrow x$ if x depends positively on y, and a "negative" edge $y \stackrel{\neg}{\rightarrow} x$ if x depends negatively on y. A program is *stratified* if its dependency graph contains no cycle with a negative edge.

For dood languages, stratification can also be defined via a dependency graph: Given an atom o[m→v], the method position m is the *distinguished position*, for an atom o:c, c is the distinguished position, and for an atom c::d, d is the distinguished position. A symbol x *depends positively (negatively)* on a symbol y if there is a rule r s.t. x occurs at the distinguished position of the head of r and y occurs in a positive (negative) literal in the body.

In restricted OO frameworks – such as ROL [Liu96], which provides no value inheritance, only a static class hierarchy, and no variables at method name or class positions –, this is a practicable solution.

However, in presence of non-monotonic inheritance, the notion of stratification becomes more complicated: Let c[m•→v] denote an *inheritable scalar method*, defined for the class c. If an object o is a member of class c, the value of m applied to o should be v by default, i.e., unless another value has been defined for $o.m$. This is expressed by the rule

$$\mathsf{O[m{\rightarrow}Default]} \leftarrow \mathsf{c[m{\bullet\!\rightarrow}Default]},\ \mathsf{O{:}c},\ \mathsf{not}\ \exists\ \mathsf{W{:}}\ (\mathsf{O[m{\rightarrow}W]} \wedge \mathsf{W} \neq \mathsf{Default}).$$

Thus, every application of an inheritable method to an object depends negatively on itself. Consequently, such programs are not stratifiable. In a full-fledged dood framework, there are two additional aspects rendering already simple programs non-stratified:

- If variables are allowed at distinguished positions, i.e., method or class positions in o[M→v] resp. o:C, since the variables can potentially be replaced by every symbol, the graph becomes very "dense" and almost all non-trivial programs become non-stratified.
- If equalities can be defined in a framework where entities can simultaneously play the roles of objects, classes, and methods, the result of equating two entities x and y depends on all symbols on which one of them depends. Note that equating is not detectable by static analysis of a program.

In some contexts, a *user-defined stratification* can be applied. However, this approach is also problematic: overlooking certain dependencies can lead to unintended models. Even more important, independent of the chosen framework, there are several problems which are not expressible with stratified semantics. Two of them, deep equality, and the win-move-game, are sketched in Section 5.

3 Well-Founded Semantics

3.1 Well-Founded Semantics in the Relational Context

For Datalog, the WFS has become widely accepted for general programs. Here, we briefly review the WFS [VGRS88] in the form of its bottom-up alternating fixpoint characterization [VG93].

Example 1 (Win-Move Game) One of the classical examples for WFS is the *win-move-game*: A game is given by a set of positions and a set of moves between them. It is played by two players moving alternately; if a player cannot move, she loses. Thus, a position X is won, if there is a move to some position Y which is not won (since then the opponent has to move). The game is represented by the single non-stratified rule

win(X) ←move(X,Y), ¬ win(Y).

Consider e.g. a game where $move = \{(a,b),(b,a),(b,c),(c,d)\}$. Obviously, $win(d)$ is false, since there are no moves from d. Consequently, $win(c)$ is true, since it is possible to move from c to d. On the other hand, the positions a and b in the game are *drawn*: the player moving from b has no winning strategy (moving to c would leave the opponent in a won position), but she can enforce a game of infinite length by moving from b to a and thus avoid losing. Thus, $win(a)$ and $win(b)$ are undefined because there is no "well-founded" argument to make each of them either false or true. □

Alternating Fixpoint Characterization. The original definition of the WFS is given in [VGRS88]. In [VG93], its bottom-up alternating fixpoint characterization is given as follows: Given a fixed Herbrand interpretation $\mathcal{J}$ (i.e., a set of ground atoms), every logic program P (containing the database D as a set of facts) gives rise to an operator $T_P^{\mathcal{J}}$, mapping interpretations to interpretations:

$$T_P^{\mathcal{J}}(\mathcal{I}) := \{H \mid (H \leftarrow B_1,\ldots,B_n,\neg C_1,\ldots,\neg C_m) \in ground(P) \text{ and } B_i \in \mathcal{I} \text{ for all } i=1,\ldots,n \text{ and } C_j \notin \mathcal{J} \text{ for all } j=1,\ldots m\}$$

Since $\mathcal{J}$ is fixed, $T_P^{\mathcal{J}}$ is a monotone operator. Let $\Gamma_P(\mathcal{J}) := \mathrm{lfp}(T_P^{\mathcal{J}})$ be its least fixpoint. The operator Γ_P is antimonotone (observe how $\mathcal{J}$ is used in $T_P^{\mathcal{J}}$), i.e., $\mathcal{J}_1 \subseteq \mathcal{J}_2$ implies $\Gamma_P(\mathcal{J}_2) \subseteq \Gamma_P(\mathcal{J}_1)$. It follows that Γ_P^2 $(:= \Gamma_P \circ \Gamma_P)$ is a monotone operator. Thus, the even indices in the sequence $\Gamma_P^0 := \emptyset, \Gamma_P^1, \Gamma_P^2, \ldots$ form a monotonically growing sequence of underestimates of the true atoms, finally reaching the least fixpoint, $\mathrm{lfp}(\Gamma_P^2)$, whereas the odd indices form a monotonically decreasing sequence of overestimates, converging against the greatest fixpoint $\mathrm{gfp}(\Gamma_P^2)$.

Theorem 1 (AFP Characterization, [VG93]) *For every ground atom A, its truth value in the* well-founded model $\mathcal{W}(P)$ *of a given program P is*

$$\mathcal{W}(P)(A) = \begin{cases} \mathit{true} & \text{if } A \in \mathrm{lfp}(\Gamma_P^2), \\ \mathit{false} & \text{if } A \notin \mathrm{gfp}(\Gamma_P^2), \\ \mathit{undef} & \text{if } A \in \mathrm{gfp}(\Gamma_P^2) \backslash \mathrm{lfp}(\Gamma_P^2). \end{cases}$$

□

In Example 1, the sequence $\Gamma_P^0, \Gamma_P^1, \Gamma_P^2, \ldots$ yields the alternating sequence

$$\emptyset, \{a, b, c\}, \{c\}, \{a, b, c\}, \{c\}, \ldots$$

of values for the *win* relation. Since $win(c) \in \mathrm{lfp}(\Gamma_P^2)$, $win(d) \notin \mathrm{gfp}(\Gamma_P^2)$, and $win(a), win(b) \in \mathrm{gfp}(\Gamma_P^2) \backslash \mathrm{lfp}(\Gamma_P^2)$ the expected truth values from above are obtained.

Computing WFS via States. The sequence of applications of Γ_P can be computed by a logic program which is obtained from the original program by introducing an additional argument position for IDB-relations, representing the state sequence (this construction is a variant of [KRS95], see also [ZAO93], [LHL95]). Corresponding to the definition of $T_P^{\mathcal{J}}$, in each rule, this argument is set to $S+1$ for all positive literals (including the head literal), otherwise to S. The distinguished *state variable* S is restricted by the additional literal state(S). The rule from Example 1 translates into

win(S+1, X) ← move(X,Y), ¬ win(S, Y), state(S).
state(0).
state(S+1) ← state(S).

Note that negative dependencies in the translated program are only to the predecessor state and to EDB relations, hence there are no cyclic negative dependencies between state-ground atoms. Thus, using the rewritten program, the WFS can be computed also by systems which do not originally provide a WFS: For every fixed state s, a *positive* program is evaluated, since all negated atoms refer to the – completely evaluated – predecessor state and thus can be regarded as input. By successively instantiating S with $0, 1, 2, \ldots$, precisely the AFP computation is obtained. Given a finite database, the computation finally becomes stationary or 2-periodic, and the well-founded model can be determined from the fixpoints. In the sequel, we will exploit this technique in the context of an object-oriented data model. An optimization of AFP has been presented in [ZFB97].

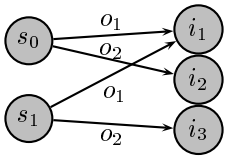
3.2 Translation into the OO-Paradigm

Although the definition in [VGRS88] is given in relational context, it does not depend on the fact that the atoms of the respective logic programs are relational atoms. Thus, the definition can be carried over to dood languages by generalizing from (implicitely) relational atoms to atoms of an arbitrary deductive language. Then, the handling of single-valued methods, transitivity of class hierarchy, and inheritance must be integrated accordingly. Also, the AFP characterization can be carried over.

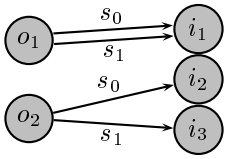
As presented in Section 3.1, in Datalog, the AFP characterization can be implemented via *reification*, i.e., every n-ary relation $p(X_1, \ldots, X_n)$ is replaced by an $n+1$-ary one, $p(S, X_1, \ldots, X_n)$, where the first argument holds the state. In the OO paradigm, the extension by states can be done analogously: Each atom has to be extended (in at least one position) with a state component.

States in an Object-Oriented Model. With the rich variety of concepts to cover different roles, i.e., objects, classes, and methods, there are several possibilities how to integrate the notion of states into a given program. The modeling of explicit states in the OO paradigm and F-Logic in context of process modeling is dealt with in [MSL97]; here, the use of states as an internal tool is demonstrated.

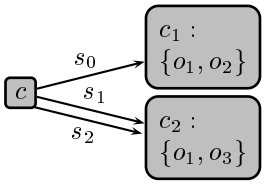
States as objects: If the focus is on the computation sequence represented by a program, it is preferable to view states s as objects. Objects o act on them as methods, addressing the instance i corresponding to object o in this state.



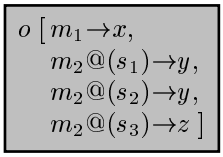
Dynamic objects: For an object o, a state s is a method, giving the instance of o corresponding to state s. In this case, the result of applying some method m to an object o in state s is derived as the result of the application of m to the corresponding instance.



Dynamic classes: For a class c, a state s is a method, giving the instance of the class c_s in this state. Dynamic classes are closely related with dynamic objects since classes and objects can be seen as two roles of the same entities.



Dynamic methods: For an object o, a state s is an additional argument of a method m, $o[m@(s) \rightarrow X]$, yielding the value of the method in this state. The concept of dynamic methods is in some sense complementary to dynamic objects.



Depending on the semantic and syntactic capabilities of the chosen framework, the choice between the above possibilities can be restricted. Especially, "states as objects", "dynamic objects", and "dynamic classes" require variables to appear at method positions: With "states as objects", the objects are methods to states, thus, variables at object positions become variables at method positions. With "dynamic objects" and "dynamic classes", states appear as methods, thus state variables appear as variables at method positions. Both approaches also require object creation, anonymous objects, and anonymous classes.

"Dynamic methods" corresponds directly to reification in relational frameworks, but it must be complemented by one of the other approaches to cover also a state-dependent class-membership and class hierarchy.

3.3 Alternating Fixpoint in the Object-Oriented Paradigm

Regarding the method application atoms $\mathsf{o[m{\rightarrow}v]}$, the state component must at least be associated with the object or with the method. Due to the fact that *is-a* atoms (i.e., $\mathsf{o{:}c}$ or $\mathsf{c{::}d}$) contain only objects and classes, dynamic methods would not be sufficient there. Thus, states are associated with objects and classes. This can be done equivalently by *states as objects* or by *dynamic objects* and *dynamic classes*. In both cases, let $\mathsf{a}[\![S]\!]$ denote the atom a extended by a state S.

For a given program P, assume that certain atoms are not subject to change, we call these *EDB atoms*. The other atoms are called *IDB atoms*.

In the same way as presented in Statelog [LHL95], a program P is transformed into a program P_{AFP} computing the WFS via the alternating-fixpoint characterization: for every rule $\mathsf{h} \leftarrow \mathsf{b}$,

- EDB literals (occurring only in the body) remain unchanged,
- every positive IDB literal l is replaced by $l[\![S{+}1]\!]$,
- every negative IDB literal $\neg l$ (which can occur only in the body) is replaced by $\neg l[\![S]\!]$, and
- the body is extended with the literal $\mathsf{S{+}1{:}state}$.

Additionally, there are rules $\mathsf{0{:}state}$ and $\mathsf{S{+}1{:}state} \leftarrow \mathsf{S{:}state}$.

Negative dependencies in P_{AFP} are only from atoms of one state to atoms of the preceding state and to EDB atoms without state association. Thus, the state sequence provides a *local* stratification. The program must now be evaluated accordingly, i.e., one state after another. Thus, the only control needed is to check if a deductive fixpoint is reached and then starting the next deductive fixpoint, and to check if the state sequence becomes stationary or 2-periodic, i.e., whether the least and greatest fixpoints are computed (which will eventually be the case for finite databases). Then, the computation can be stopped, yielding a finite structure $\mathcal{A}_P$. For arbitrary languages, the required control can be encoded using the inflationary semantics (cf. [AHV95, p. 400 ff] where it shown how WFS can be computed using *while*$^+$). In F-Logic, such control can be implemented much more directly using its trigger mechanism as will be described in Section 4.1. W.l.o.g., assume that the last state which has been computed has an even index s_0. For every s s.t. $\mathcal{A}_P \models (s : \mathsf{state})$, let

$$\mathcal{A}_P^{[s]} := \{a \mid \mathcal{A}_P \models a[\![s]\!],\ a \text{ an IDB atom}\} \cup \{a \mid \mathcal{A}_P \models a,\ a \text{ an EDB atom}\}$$

be the "snapshot" at state s.
Then, either the last underestimate $\mathcal{A}_P^{[s_0]}$ and the last overestimate $\mathcal{A}_P^{[s_0-1]}$ coincide, i.e., $\mathcal{A}_P^{[s_0]} \models a \ \Leftrightarrow\ \mathcal{A}_P^{[s_0-1]} \models a$ for all a, or some atoms a are false in the underestimate and true in the overestimate, i.e., $\mathcal{A}_P^{[s_0]} \models \neg a$ and $\mathcal{A}_P^{[s_0-1]} \models a$.

Theorem 2 *Analogous to Theorem 1, the well-founded model $\mathcal{W}_P$ is given as*

$$\mathcal{W}_P(a) = \begin{cases} \mathit{true} & \Leftrightarrow\ \mathcal{A}_P^{[s_0]} \models a \\ \mathit{undef} & \Leftrightarrow\ \mathcal{A}_P^{[s_0]} \models \neg a \text{ and } \mathcal{A}_P^{[s_0-1]} \models a \\ \mathit{false} & \Leftrightarrow\ \mathcal{A}_P^{[s_0-1]} \models \neg a\ . \end{cases}$$

□

Like in the relational case,

- If $\mathcal{A}_P^{[s_0]} = \mathcal{A}_P^{[s_0-1]}$, the well-founded model $\mathcal{W}_P$ of P is total and $\mathcal{W}_P = \mathcal{A}_P^{[s_0]}$.
- If the original program was locally stratified, $\mathcal{W}_P$ is total.

In contrast to the relational case, in the object-oriented paradigm there are some semantical intricacies due to the inherent semantics of functional methods and object identity:

- in the overestimates, i.e., odd s, for some method applications $o[m{\rightarrow}\_]$ which are intended to be single-valued, there can be $v_1 \neq v_2$ s.t. $\mathcal{A}_P \models [m{\rightarrow}v_1][\![s]\!]$ and $\mathcal{A}_P \models [m{\rightarrow}v_2][\![s]\!]$, thus, the functionality requirement can be (temporally) violated in the overestimates (i.e., for odd s). Since at this intermediate point of the computation, true method atoms represent *potential* results, functionality is not required.
- In the well-founded model, the truth value of a method application $o[m{\rightarrow}v]$ can be undefined (not to be confused with the application of m to o being undefined). Moreover, for some method applications $o[m{\rightarrow}\_]$, there can be several such v's, thus, undefined atoms in the result also do not fit the functionality requirement:

Example 2 (Mutual Exclusion) Consider the following program, representing a database in which it is only known that John is either married to Jane or to Mary:

$$P := \{\, \mathsf{john[spouse{\rightarrow}mary]} \leftarrow \mathsf{not\ john[spouse{\rightarrow}jane]}.$$
$$\mathsf{john[spouse{\rightarrow}jane]} \leftarrow \mathsf{not\ john[spouse{\rightarrow}mary]}.$$
$$\mathsf{O[married{\rightarrow}true]} \leftarrow \mathsf{O[spouse{\rightarrow}X]}\ .\ \}$$

Then, P_{AFP} consists of the following rules:

john[spouse→mary][$\![S{+}1]\!$] ← not john[spouse→jane][$\![S]\!$], S+1:state.
john[spouse→jane][$\![S{+}1]\!$] ← not john[spouse→mary][$\![S]\!$], S+1:state.
O[married→true][$\![S{+}1]\!$] ← O[spouse→X], S+1:state.
0:state.
S+1 : state ← S : state.

The following AFP computation is obtained:
$\mathcal{A}_P^{[0]} = \emptyset$, $\mathcal{A}_P^{[1]} = \{$john[spouse→{jane,mary}], john[married→true]$\}$ and $\mathcal{A}_P^{[2]} = \emptyset$, hence the sequence becomes periodic for $s_0 = 2$. Thus, in the well-founded model,
$\mathcal{W}$(john[spouse→jane]) = $\mathcal{W}$(john[spouse→mary]) = $\mathcal{W}$(john[married→true]) = *undef*. □

Especially, if functionality is maintained by the underlying system (e.g., by equating of objects, or signaling an error), this must be disabled during the AFP computation. In some frameworks – e.g. F-Logic as shown in the subsequent section –, this can be done by replacing single-valued methods by multivalued ones.

Similarly, in the overestimates, the subclass-relation can happen to be cyclic, and the final result may contain undefined class-membership atoms.

Summarizing, WFS can be applied to the dood context, but one has to take care about the proper use of the truth-value *undefined*.

4 Implementation in F-Logic

F-Logic [KLW95] is a dood language combining the advantages of deductive databases with the rich modeling capabilities (objects, methods, class hierarchy,

non-monotonic inheritance, signatures) of the OO data model. The syntax allows to use variables for oid's, method names, method arguments and results, and class names. The full syntax and semantics is given in [KLW95, FLU94]. F-Logic has been implemented in FLORID (F-LOgic Reasoning In Databases) [FHK+97][3]. Here, only the features which are relevant for applying the required program transformations are presented. In brief, the syntax and semantics can be described as follows:

- The alphabet of an F-Logic language consists of a set $\mathcal{F}$ of *object constructors*, playing the role of function symbols, a set $\mathcal{P}$ of predicate symbols, a set $\mathcal{V}$ of variables, several auxiliary symbols, containing), (,], [, $\rightarrow$, $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, :, and the usual first-order logic connectives. By convention, object constructors start with lowercase letters whereas variables start with uppercase ones.
- *id-terms* are composed from object constructors and variables. They are interpreted as elements of the universe.

In the sequel, let O, C, D, M, X_i, V, V_i, ScM, and MvM denote id-terms.

- A *method application* is an expression $M@(X_1, \ldots, X_k)$.
- if $M@(X_1, \ldots, X_k)$ is a method application and O is an id-term, the *path expression* $O.(M@(X_1, \ldots, X_k))$, denoting the object resulting from applying $M@(X_1, \ldots, X_k)$ to O, is an id-term. This results in an *anonymous object* which is created when some object atom $O.M@(X_1, \ldots, X_k)[\ldots]$ is defined.
- The following are *object atoms*:
  - $O[ScM@(X_1, \ldots, X_k) \rightarrow V]$: applying the *scalar* method ScM with arguments $X_1, \ldots, X_k$ to O – as an object – results in V,
  - $O[ScM@(X_1, \ldots, X_k) \bullet\!\!\rightarrow V]$: O – as a class – provides the *inheritable scalar* method ScM to its members, which, if called with arguments $X_1, \ldots, X_k$ results in V,
  - $O[MvM@(X_1, \ldots, X_k) \twoheadrightarrow \{V_1, \ldots, V_n\}]$: applying the *multivalued* method MvM with arguments $X_1, \ldots, X_k$ to O results in some V_i.
  - $O[MvM@(X_1, \ldots, X_k) \bullet\!\!\twoheadrightarrow \{V_1, \ldots, V_n\}]$, analogous for an *inheritable multivalued* method.
- An *is-a assertion* is an expression of the form $O : C$ (object O is a member of class C), or $C :: D$ (class C is a subclass of class D).
- A *predicate atom* is an expression of the form $p(X_1, \ldots, X_n)$ where $p \in \mathcal{P}$.
- *Formulas* are built from F-Logic's atoms, i.e., is-a assertions, object atoms, and predicate atoms by first-order logic connectives.
- An F-Logic *rule* is a logic rule $h \leftarrow b$ over F-Logic's atoms.
- An F-Logic *program* is a set of rules.

In FLORID, F-Logic programs are evaluated wrt. inflationary fixpoint semantics, additionally, user-defined stratification is supported. Non-monotonic inheritance is implemented via a trigger mechanism in a *deduction precedes inheritance* manner which is described in the next section to implement the state sequence. We exploit this mechanism to obtain a concise implementation of the state sequence.

[3] available at `http://www.informatik.uni-freiburg.de/~dbis/flogic-project.html`.

4.1 Programming the State Sequence in F-Logic.

In F-Logic, the state-by-state evaluation can be enforced using its trigger mechanism which allows insertion of atoms into the database after a deductive fixpoint has been reached. Originally, this mechanism is used to implement non-monotonic inheritance: Non-monotonic inheritance of a property from a class to an object takes place if a) it is inheritable, and b) no other property can be derived for the object. Thus, inheritance is done *after* pure deduction: fixpoint computation and inheriting one fact at a time alternate until an outer fixpoint is reached.

This mechanism can be utilized to define a sequence of deductive fixpoint computations by defining a set of inheritable methods which "trigger" the next computation: By defining a class state which provides an inheritable boolean method ready, the sequential computation of states can be controlled (see Table 1, recall that $\langle atom \rangle [\![S]\!]$ denotes the atom $\langle atom \rangle$ associated with state S). The alternating fixpoint computation is stopped when the underestimates become stationary by comparing the even states.

```
0:state.
0:even.
state[ready•→true].
state[running•→false].
S:state ← T[running→true], T.ready[ ], S = T + 1.
S:even ← S:state, S = T + 1, T:odd.
S:odd ← S:state, S = T + 1, T:even.

0[running→true].
S[running→true] ← S:odd.

S[running→true] ← ⟨atom⟩[[S]], not ⟨atom⟩[[T]], S = T + 2, T:even.
S:final ← S[running→false].
```

Table 1. Implementation of the State Sequence

The rule state[ready•→true] defines an inheritable method of the class state. For every state s, its IDB is computed via deduction when s becomes a member of state. Additionally, either s.[running→true] is derived (trivially for odd s, or due to new atoms in the underestimate if s is even), or the method running remains undefined for s. Since deduction precedes inheritance, when the computation of a state s is completed, s[ready→true] is *inherited*, and, if running is still undefined, also s[running→false] is inherited. Depending on s.running, either the computation is continued by making s+1 the next state and starting the computation of s+1, or the subsequent deduction step derives s:final.

Due to the higher-order syntax of F-Logic, the fixpoint check can be implemented in a very generic way, using variables at object, method, argument, and class-positions:

S[running→true] ← O.S[M→V], not O.T.M[ ], S = T + 2, T:even.
S[running→true] ← O.S[M↠V], not O.T[M↠V], S = T + 2, T:even.
S[running→true] ← O.S:C.S, not O.T:C.T, S = T + 2, T:even.
S[running→true] ← C.S::D.S, not C.T::D.T, S = T + 2, T:even.

4.2 AFP Transformation

Equality, Scalar Methods. For scalar methods, functionality is enforced in F-Logic, i.e., if two atoms are derived assigning different objects as results of a method application to an object, e.g., john[spouse→mary] and john[spouse→jane], those objects are equated. To get around unintended equating of objects due to different values of a method application to an object in overestimates, scalar methods are encoded as multivalued methods during the computation.

Definition 1 The operator Ψ which transforms scalar methods into multivalued methods is defined as follows, marking transformed methods by #:

Ψ is the identity on is-a atoms, is-subclass atoms, predicate atoms, and object atoms handling multivalued methods. For object atoms handling scalar methods,

$$\Psi(\mathsf{O[M@(X_1,\ldots,X_n)} \rightarrow \mathsf{V]}) := \mathsf{O[M\#@(X_1,\ldots,X_n)} \twoheadrightarrow \mathsf{V]} \quad \text{and}$$
$$\Psi(\mathsf{O[M@(X_1,\ldots,X_n)} \bullet\!\!\rightarrow \mathsf{V]}) := \mathsf{O[M\#@(X_1,\ldots,X_n)} \bullet\!\!\twoheadrightarrow \mathsf{V}] \ .$$ □

Translation of the Program. For the translation, states are associated to atoms following the ideas of *dynamic objects* and *dynamic classes*:

Definition 2 For F-Logic atoms, the state associating operator $[\![S]\!]$ is defined as follows:

$$\begin{aligned}
&\mathsf{O[M@(X_1,\ldots,X_n)} \rightsquigarrow \mathsf{V]}[\![S]\!] &&:= \mathsf{O.S[M@(X_1,\ldots,X_n)} \rightsquigarrow \mathsf{V]} \\
&&&\quad \text{for } \text{“}\rightsquigarrow\text{”} \in \{\rightarrow, \bullet\!\!\rightarrow, \twoheadrightarrow, \bullet\!\!\twoheadrightarrow\}, \\
&\mathsf{O:C}[\![S]\!] &&:= \mathsf{O.S:C.S} \ , \\
&\mathsf{O::C}[\![S]\!] &&:= \mathsf{O.S::C.S} \ , \quad \text{and} \\
&\mathsf{p(X_1,\ldots,X_n)}[\![S]\!] &&:= \mathsf{p(S,X_1,\ldots,X_n)} \ .
\end{aligned}$$ □

Here, path expressions are used to create and address anonymous objects.

Definition 3 (Transformed Program) For a given program P, the transformed program P_{AFP} is obtained as follows: For every rule $\mathsf{h} \leftarrow \mathsf{b}$,

- EDB literals (occurring only in the body) remain unchanged,
- every positive IDB literal l is replaced by $\Psi(l[\![S]\!])$,
- every negative IDB literal $\neg l$ (which can occur only in the body) is replaced by $\neg\Psi(l[\![T]\!])$, and
- the rule body is extended by the atom S:state if rule contains no negative IDB literals.
- the body is extended by the atoms S=T+1 and T.ready[ ] if the rule contains negative IDB literals.

Then, the rules shown in Table 1 are added. □

Evaluation of the Result. Due to the transformation of scalar methods into multivalued methods, in the well-founded model $\mathcal{W}_P$ of P_{AFP}, all scalar methods are replaced by the corresponding marked multivalued methods.

Example 3 Consider again the program of Example 2. P_{AFP} consists of the rules given in Table 1 and the following ones:

john.S[spouse#→↠mary] ← not john.T[spouse#→↠jane], S = T + 1, T.ready[ ].
john.S[spouse#→↠jane] ← not john.T[spouse#→↠mary], S = T + 1, T.ready[ ].
P.S[married#→↠true] ←P.S[spouse#→↠X], S:state.

The alternating fixpoint computation stops for $s_0 = 2$, i.e., $\mathcal{A} \models 2 : \mathsf{final}$ and

$\mathcal{A}^{[1]} \models \mathsf{john[spouse\#\twoheadrightarrow\{jane,mary\}]} \wedge \mathsf{john[married\rightarrow true]}$, and for $s \in \{0, 2\}$.
$\mathcal{A}^{[s]} \models \neg\mathsf{john[spouse\#\twoheadrightarrow jane]} \wedge \neg\mathsf{john[spouse\#\twoheadrightarrow mary]} \wedge \neg\mathsf{john[married\#\twoheadrightarrow true]}$.

Thus, in the well founded model,
$\mathcal{W}(\mathsf{john[spouse\#\twoheadrightarrow jane]}) = \mathcal{W}(\mathsf{john[spouse\#\twoheadrightarrow mary]}) = \mathit{undef}$ and
$\mathcal{W}(\mathsf{john[married\#\twoheadrightarrow true]}) = \mathit{undef}$. □

Depending on the application, there can be several ways how to retranslate the well-founded model to the original signature, dealing with the requirements of scalar methods and the interpretation of undefinedness. The straightforward case is, when a) the well-founded model is total, and b) the functionality of scalar methods is satisfied. Then, the following rules extract a total F-structure over the signature of the original program from the alternating-fixpoint computation:

O[M@(X$_1$,...,X$_n$) →V] ← O.S[M#@(X$_1$,...,X$_n$) →↠V], S:final.
O[M@(X$_1$,...,X$_n$) •→V] ← O.S[M#@(X$_1$,...,X$_n$) •↠V], S:final.
O[M@(X$_1$,...,X$_n$) →↠V] ← O.S[M@(X$_1$,...,X$_n$) →↠V], S:final.
O[M@(X$_1$,...,X$_n$) •↠V] ← O.S[M@(X$_1$,...,X$_n$) •↠V], S:final.
O:C ← O.S:C.S, S:final.
C::D ← C.S::D.S, S:final.
p(X$_1$,...,X$_n$) ← p(S,X$_1$,...,X$_n$), S:final.
error ← ⟨atom⟩[[T]], not ⟨atom⟩[[S]], S = T + 1, T:state, S:final.
error ← O.S[M#@(X$_1$,...,X$_n$) •↠V], O.S[M#@(X$_1$,...,X$_n$) •↠W], not V = W, S:final.

If the well-founded model is partial, the meaning of an undefined truth-value has to be defined wrt. the application semantics. The examples in Section 5 illustrate that there are several possibilities, depending on the application:

(i) In some cases (cf. win-move game, Example 5.1), undefined atoms are an intended, reasonable result.
(ii) In some cases (cf. deep equality, Example 5.2), undefined atoms can be interpreted as true or as false ones, depending on the intended application semantics.
(iii) otherwise, they indicate an error in the program resp. in the specification, e.g. insufficient knowledge or inconsistencies. (For instance, in Example 2, where $\mathcal{W}(\mathsf{john[spouse\rightarrow mary]}) = \mathit{undef}$ and $\mathcal{W}(\mathsf{john[spouse\rightarrow jane]}) = \mathit{undef}$).

5 Applications and Examples

In this section, we illustrate the approach with two typical examples[4].

5.1 Win-Move Games

An object-oriented formulation of the win-move game (cf. Example 1) where the move relation is assumed to be given as EDB is

$P :=$ {game[win→↠X] ← move(X,Y), not game[win→↠Y].}

The translated P_{AFP} program comprises the rules given in Table 1 and additionally

S[running→true] ← game.S[win→↠X], not game.T[win→↠X], S = T + 2, T:even.
game.S[win→↠X] ← move(X,Y), not game.T[win→↠Y], S = T + 1, T.ready[ ].

The model is evaluated by

game[win→↠X] ← game.S[win→↠X], S:final.
game[undef→↠X] ← game.T[win→↠X], not game.S[win→↠X], S:final, T:state, S = T + 1.
game[lost→↠X] ← X:dom, not game.T[win→↠X], S:final, T:state, S = T + 1.

An important application of win-move games is the area of argumentation frameworks [Dun95].

5.2 Deep Equality

In an object-oriented framework, objects are called *deep-equal*, if they cannot be distinguished by looking at their values, possibly dereferencing the oid's appearing therein and doing this recursively, also called "pointer-chasing" (cf. [AdB95]). Deep equality is the coarsest equivalence relation among objects satisfying the requirement that two objects are equivalent if their values are.

In [AdB95] it is shown that, provided that there are no set values, deep-equality is expressible with stratified negation, for instance with the rules shown in Table 2; the stratification is given by EDB $\prec$ not_deep_eq $\prec$ deep_eq. (The rules are presented in the state-extended form for further usage.)

Two objects are not deep equal, if

- they are either different basic values, or
- one is a member (or a subclass) of a class where the other is no member (resp. subclass), or
- there is a method application which is defined for only one of the objects under consideration, or
- if there is a method application which results in different objects which are not deep-equal.

Then, two objects are deep equal, if with the above characteristics, it cannot be proven that they are not deep-equal.

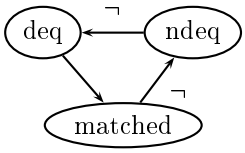
[4] The examples are available at `http://www.informatik.uni-freiburg.de/~dbis/flsys/morexamples.html`.

```
deep_eq(S,X,Y) ← X:dom, Y:dom, not not_deep_eq(T,X,Y), S = T + 1, T.ready[ ].
deep_eq(S,X,X) ← X:dom, S:state.

not_deep_eq(S,X,Y) ← X:basic_value, Y:basic_value, not X = Y, S:state.
not_deep_eq(S,X,Y) ← not_deep_eq(S,Y,X), S:state.

not_deep_eq(S,X,Y) ← X:dom, Y:dom, X:C, not Y:C, S:state.
not_deep_eq(S,X,Y) ← X::C, not X = C, Y:dom, C:dom, not Y::C, S:state.

not_deep_eq(S,X,Y) ← X:dom, Y:dom, X[M→V], not Y.M[ ], S:state.
not_deep_eq(S,X,Y) ← X:dom, Y:dom, X[M→V], Y[M→W],
                     not_deep_eq(S,V,W), S:state.
```

Table 2. Deep Equality without Set Values

In contrast, in presence of set values – which is the case in F-Logic due to the existence of multivalued methods –, deep-equality is *not* expressible with stratified negation:

For two objects x and y under consideration, an object v which results from applying a multivalued method m to x is *matched* if there is an object w resulting from applying m to y and v and w are deep-equal. Here, not-deep-equality depends negatively on matching which itself depends positively on deep-equality, thus negatively on not-deep-equality, making up a negative cycle.




```
not_deep_eq(S,X,Y) ← X:dom, Y:dom, X[M↠V], not Y.T[matched@(M)↠V],
                     S = T + 1, T.ready[ ].
Y.S[matched@(M)↠V] ← Y:dom, Y[M↠W], deep_eq(S,V,W), S:state.
```

Table 3. Deep Equality with Set Values

The rules of Tables 1, 2, and 3 together compute deep-equality in presence of multivalued methods with the WFS.

In case a signature contains no multivalued methods, the well-founded model is total, and coincides with the stratified model obtained by the rules given in Table 2.

Otherwise, if a signature contains multivalued methods, the above rules induce cyclic dependencies of the form "a is not deep-equal to b if c is not deep-equal to d", and "c is not deep-equal to d if a is not deep-equal to b". Then, there can be no well-founded argumentation that the respective objects are deep-equal. If there is also no well-founded argumentation that these objects are not deep-equal, the respective deep-equalities are undefined in the well-founded model. Note, that if two objects are not deep-equal, there is a well-founded derivation for this fact. Thus, if $\mathcal{W}(\mathsf{deep\_eq(x,y)}) = \mathit{undef}$, x and y are actually deep-equal. The well-founded model is evaluated as follows (note that since with s:final, s is even, thus $t = s–1$ is odd, representing an overestimate):

deep_eq(X,Y) ← deep_eq(T,X,Y), S:final, S = T + 1.

Deductive Equality. Furthermore, from the deductive point of view, two objects can also be distinguished by looking at their occurrences as *results* of method applications, corresponding to following references in the inverse direction. A finer equivalence relation, *deductive equality* is defined as shown in Table 4.

```
ded_eq(S,X,Y) ← X:dom, Y:dom, not not_ded_eq(T,X,Y), S = T + 1, T.ready[ ].
ded_eq(S,X,X) ← X:dom, S:state.

not_ded_eq(S,X,Y) ← not_deep_eq(S,X,Y).
not_ded_eq(S,X,Y) ← not_ded_eq(S,Y,X), S:state.
not_ded_eq(S,X,Y) ← X:dom, Y:dom, X[M→V], Y[M→W], not_ded_eq(S,V,W), S:state.
not_ded_eq(S,X,Y) ← X:dom, Y:dom, X[M↠V], not Y.T[mvMatched@(M)↠V],
                    S = T + 1, T.ready[ ].
O.S[mvMatched@(M)↠V] ← O:dom, O[M↠W], ded_eq(S,V,W), S:state.
not_ded_eq(S,X,Y) ← X:dom, Y:dom, V[M→X], not Y.T[invScMatched@(M)↠V],
                    S = T + 1, T.ready[ ].
O.S[invScMatched@(M)↠V] ← O:dom, W:dom, W[M→O], ded_eq(S,V,W), S:state.
not_ded_eq(S,X,Y) ← X:dom, Y:dom, V[M↠X], not Y.T[invMvMatched@(M)↠V],
                    S = T + 1, T.ready[ ].
O.S[invMvMatched@(M)↠V] ← O:dom, W:dom, W[M↠O], ded_eq(S,V,W), S:state.
```

Table 4. Additional Rules for Deductive Equality

Here again, undefined atoms can be interpreted as true:

ded_eq(X,Y) ← ded_eq(T,X,Y), S:final, S = T + 1.

Theorem 3 *Two objects are deductive equivalent iff they cannot be distinguished by any program which does not refer to object id's.* □

Example 4 Consider the following database:

```
a[value→1;next→b].  b[value→2;next→d].      c[value→2;next→e].
d[value→3].         e[value→3].
```

Then, b and c resp. d and e are deep-equal, although they are not deductive-equal (since a[next→b], but c is not the result of next applied to any object which is deductive-equal to a). If a is removed, $\mathcal{W}$(deep_eq(b,c)) = $\mathcal{W}$(deep_eq(d,e)) = *true*, and $\mathcal{W}$(ded_eq(b,c)) = $\mathcal{W}$(ded_eq(d,e)) = *undef* due to the cyclic dependency. Here, the above-mentioned policy to interpret undefined atoms as true becomes important. □

6 Conclusion

The work shows that the translation of well-founded semantics and its alternating-fixpoint characterization to deductive object-oriented database languages is possible and yields a reasonable semantics. There are several problems due to object-oriented features, such as scalar methods and object identity, or undefinedness in

class-membership. We suspect that this is the reason that until now, no deductive object-oriented framework came up with a well-founded semantics. We have shown that those problems can be described and solved in a model-theoretic, generic way. Thus, given a problem and a specification how to deal with different assignments to scalar methods and how to interpret undefined atoms, a semantics wrt. those parameters based on the well-founded model is uniquely determined and effectively computable. The approach can be applied to arbitrary dood languages.

Since there are several problems which are not expressible using only stratified negation, our approach makes this class of problems amenable to deductive object-oriented database languages.

References

[AdB95] S. Abiteboul and J. V. den Bussche. Deep Equality Revisited. In Ling et al. [LMV95].

[AG91] S. Abiteboul and S. Grumbach. A rule based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[AK92] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System – The Story of O_2*, chapter 5, pages 98–127. Morgan Kaufmann, 1992.

[BJZ94] J. B. Bocca, M. Jarke, and C. Zaniolo, editors. *Proc. Intl. Conference on Very Large Data Bases*, Santiago de Chile, 1994.

[BPF+94] M. L. Barja, N. W. Paton, A. A. A. Fernandes, M. H. Williams, and A. Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In Bocca et al. [BJZ94], pages 463–474.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In H. Garcia-Molina and H. V. Jagadish, editors, *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 225–236, 1990.

[CTT93] S. Ceri, K. Tanaka, and S. Tsur, editors. *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 760 in LNCS. Springer, 1993.

[CW89] W. Chen and D. S. Warren. C-Logic for complex objects. In *Proc. ACM Symposium on Principles of Database Systems*, pages 369 – 378, 1989.

[Dix95] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. In A. Fuhrmann and H. Rott, editors, *Logic, Action and Information.* de Gruyter, 1995.

[Dun95] P. M. Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-Person Games. *Artificial Intelligence*, 77:312–357, 1995.

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F–Logic. In *Proc. Intl. Conference on Data Engineering*, 1997.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In Bocca et al. [BJZ94].

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *Proc. Intl. Conference on Very Large Data Bases*, pages 455 – 468, Brisbane, 1990.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[KRS95] D. B. Kemp, K. Ramamohanarao, and P. J. Stuckey. ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS. In Ling et al. [LMV95].

[KW93] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77 – 120, August 1993.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A Logical Framework for Active Rules. In *Proc. 7th Intl. Conf. on Management of Data (COMAD)*, Pune, India, December 1995. Tata McGraw-Hill.

[Liu96] M. Liu. ROL: A Deductive Object Base Language. *Information Systems*, 21(5):431–457, 1996.

[LMV95] T. W. Ling, A. O. Mendelzon, and L. Vieille, editors. *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1013 in LNCS, Singapore, 1995. Springer.

[Mai86] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6 – 26, 1986.

[MR93] I. S. Mumick and K. A. Ross. Noodle: A Language for Declarative Querying in an Object-Oriented Database. In Ceri et al. [CTT93].

[MSL97] W. May, C. Schlepphorst, and G. Lausen. Integrating Dynamic Aspects into Deductive Object-Oriented Databases. In A. Geppert and M. Berndtsson, editors, *Proc. of the 3nd Intl. Workshop on Rules in Database Systems (RIDS)*, LNCS, Skövde, Sweden, 1997.

[VG93] A. Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VGRS88] A. Van Gelder, K. Ross, and J. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proc. ACM Symposium on Principles of Database Systems*, pages 221–230, 1988.

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach. In Ceri et al. [CTT93].

[ZFB97] U. Zukowski, B. Freitag, and S. Brass. Improving the Alternating Fixpoint: The Transformation Approach. In *4th Intl.Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, LNAI, Berlin, 1997. Springer.