**3rd International Workshop on Rules in Database Systems (RIDS'97), June 26-28, 1997, Skövde, Sweden.
Springer LNCS 1312, pp. 20-34.**

Integrating Dynamic Aspects into Deductive Object-Oriented Databases

Wolfgang May[⋆] Christian Schlepphorst[⋆⋆] Georg Lausen

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

Abstract. We show how the dynamics of database systems can be modeled by making states first-class citizens in an object-oriented deductive database language. With states at the same time acting as objects, methods, or classes, several concepts of dynamic entities can be implemented, allowing an intuitive, declarative modeling of the application domain. Exploiting the natural stratification induced by the state sequence, the approach also provides an implementable operational semantics.
The method is applicable to arbitrary object-oriented deductive database languages which provide a sufficiently flexible syntax and semantics. Provided an implementation of the underlying database language, any specification in the presented framework is directly executable, thus unifying specification, implementation, and metalanguage for proving properties of a system.
The concept is applied to F-Logic. Besides the declarative semantics given by the rules of a State-F-Logic program, the use of F-Logic's inheritance semantics for modeling states provides an effective operational semantics exploiting the naturally given state-stratification. State-F-Logic programs can be executed using the Florid implementation.

1 Introduction

Rules in database systems appear twofold: *Deductive* rules are used to express knowledge within states, and, orthogonally, *active* rules derive and express actions to be performed in transitions between states. In general, for modeling a temporally changing application domain, a more or less explicit notion of state is needed. Especially in deductive frameworks, integrating states explicitly into a database language provides additional flexibility and clarity in modeling, also supplying a model-theoretic base for reasoning about the database behavior. From the theory defined by the program specifying and implementing the deductive *and* dynamic behavior, correctness and liveness properties can be stated and verified using standard formal methods, such as temporal logics. Thus, for example, workflow systems can be defined, implemented, and validated from the same given specification/implementation.

In this paper, we present an abstract concept for modeling dynamic behavior by integrating explicit states into deductive, object-oriented frameworks. State

[⋆] Supported by grant no. GRK 184/1-97 of the Deutsche Forschungsgemeinschaft.
[⋆⋆] Supported by the Deutsche Forschungsgemeinschaft, La 598/3-2.

changes can be reflected by dynamic objects, dynamic methods, or dynamic classes, allowing an intuitive modeling of the application domain.

The concept is applied to F-Logic [KLW95], which by providing the required semantic and syntactic flexibility allows for a comprehensive treatment of state-changes and updates in databases. Providing as well a model-theoretic, declarative semantics as an operational semantics which is implemented by the FLORID system, State-F-Logic acts at the same time as specification language, implementation language, and metalanguage for proving properties of a system.

The paper is structured as follows: the introduction is completed with a review of related work. In Section 2, the roles of states in an object-oriented model are investigated. In Section 3, semantical aspects of state changes are analyzed, leading to a classification of rules wrt. their temporal scope and a class of programs suitable for specification and implementation of database systems. 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. The temporal, dynamic aspect of databases can be regarded as orthogonal to the static, data-oriented aspect: A single-state framework can be transferred into a multi-state framework by *versioning* (e.g. [TCG+93]), i.e. attaching an additional dimension by duplicating and indexing the single-state framework. Versioning can be employed with different granularity, e.g. the whole database, relations, objects, etc. In relational database languages, explicit states are introduced via *reification*, i.e., by adding an additional argument to each relation, corresponding to versioning of relations. Following this way, in [BCW93] (*Datalog*$_{1S}$) and [Zan93] (*XY-Datalog*), *Datalog* has been extended to explicit states. *Templog* [AM89] is another extension of Datalog, using temporal logic operators. Datalog$_{1S}$, XY-Datalog, and Templog have been proven to be equivalent. A similar concept with explicit states in Datalog, *Statelog*, has been presented in [LML96]. There, every atom $R(\bar{x})$ is augmented by a state term S to $[S]R(\bar{x})$. Thus, Statelog amounts to versioning the whole database. Since complex state terms are allowed, Statelog is not bound to linear time, but also allows branching or hierarchical state spaces. Versioning in object-oriented databases is dealt with in [CJ90]. There, the granularity of versioning is by objects, each database version consists of a version of each object stored in the system. Updates and versioning of objects in F-Logic has been presented in [KLS92]. There, updates are restricted to the form *ins*, *mod*, and *del* of method applications. Transaction Logic [BK94] is a deductive language focussing on the dynamic acspects of processes, supporting an abstract notion of states as theories. In [FWP97], an active rule language is incorporated into an object-oriented deductive database concept by introducing explicit states into the sublanguages concerned with events, conditions, and actions. Summarizing, in these approaches, the temporal aspect is not actually *integrated* into the modeling.

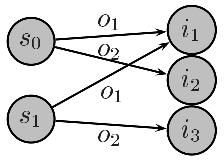
Notation. Object-oriented models can be represented by three types of atoms, i.e. method applications, class membership, and the subclass relation. In order to obtain a uniform notation, we will use F-Logic syntax (cf. Section 4) throughout this paper: o[m→v] denotes that application of method m to object o results in the value v. Parameterized methods are written as o[m@(x_1,...,x_n)→v]. Furthermore, 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 The Roles of States in an Object-Oriented Model

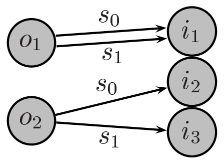
As mentioned in the previous paragraph, every object-oriented structure can be encoded into a relational schema, using atoms meth_appl(o,m,v), isa(o,c), and subcl(c,d). In this modeling, states can be introduced in the same way as in relational systems, via reification/versioning, i.e. augmenting every relation by an additional argument, denoting the state. Beyond the fact that the relational encoding impairs the intuitive modeling capabilities provided by the object-oriented paradigm, with this approach, states are not really integrated as first-order citizens into the modeling.

In an object-oriented modeling, providing a rich variety of concepts to cover different roles, such as objects, class hierarchy, and methods, there are several possibilities how states can interfere with entities of the application domain. Moreover, for every entity, it can be chosen individually how to model this interference. An important point when modeling large systems is that in every state transition, only some objects, classes, and methods will be affected. To take care of this, *abstract objects*, the "objects" of the application domain (e.g. the persons x, y), are distinguished from *object instances* which represent x and y at certain time point(s). Thus, if in state s, x is married to y, e.g. in x[married@(s)→y], x and y refer to the abstract objects, whereas, detailed state-dependent information about x in state s is provided by the *instance* of x in state s. The same applies to classes.

States as objects: If the focus is on the computation sequence represented by a specification, it is preferable to regard states s as objects. Abstract objects o act on them as methods, addressing the instance i corresponding to object o in this state. Then, state changing is simply modeled by changing the interpretation of methods from state to state.



Dynamic objects: Objects changing their behavior only from time to time can be modeled by the concept of dynamic objects, i.e., for an abstract 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: Dynamic classes are closely related with dynamic objects since classes and objects can be seen as two roles of the same entities (cf. F-Logic): For an abstract class c, a state s is a method, giving the instance of the class c_s in this state. Dynamic classes are suitable if over a computation, classes change their extension or some (default) properties inherited to all their members: If an object instance i is a member of class c in state s, $i : c_s$ s.t. $c[s \rightarrow c_s]$, it inherits some properties from the instance of the class in this state (in general, these are dynamic methods; cf. Example 5).

$$\begin{tikzcd}
 & c_1 : \{o_1, o_2\} \\
c \arrow[ur, "s_0"] \arrow[dr, "s_1"] \arrow[dr, "s_2"'] & \\
 & c_2 : \{o_1, o_3\}
\end{tikzcd}$$

Dynamic methods: If for some object, only parts of its behavior are changing, those can be modeled by dynamic methods (cf. Example 6). For an object o, a state s is an additional argument of a method m, $o[m@(s) \rightarrow X]$, giving the value of the method in this state. The concept of dynamic methods is in some sense complementary to dynamic objects.

$$\boxed{\begin{array}{l} o\,[\,m_1 \rightarrow x, \\ \quad m_2@(s_1) \rightarrow y, \\ \quad m_2@(s_2) \rightarrow y, \\ \quad m_2@(s_3) \rightarrow z\ ] \end{array}}$$

States as classes: A state can be regarded as a class, being able to have other states as subclasses and/or members, $s' :: s$ resp. $s' : s$ and to inherit properties to them.
A dynamic *EDB* entity is an instance whose behavior in some state is derived from its behavior in the predecessor state, i.e., by frame rules, whereas the behavior of a dynamic *IDB* entity is derived from the behavior of other entities in the same state.

For providing this semantical flexibility when choosing an optimal way for modeling changing properties, the object-oriented facet of the framework to be used must allow entities to act simultaneously as objects, classes, and methods. Additionally, the deductive facet should also support this flexibility by allowing variables to occur at arbitrary positions of rules, standing for arbitrary entities.

Especially, "states as objects", "dynamic objects", and "dynamic classes" require variables to appear at method positions: In "states as objects", the objects are methods to states, thus, variables at object positions become variables at method positions. In "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 must be complemented by one of the other approaches to cover also a state-dependent class-membership and class hierarchy.

3 States and Rules

For the abstract treatment, assume a deductive, object-oriented framework, called $\mathfrak{X}$, providing the facilities mentioned in the previous section. $\mathfrak{X}$ defines the syntactic notions of terms, atoms, literals, rules, and programs (recall that a logical rule is of the form $\mathsf{h} \leftarrow \mathsf{b}$ where h is an $\mathfrak{X}$-atom, and b is a conjunction of $\mathfrak{X}$-literals), and the semantic notions of an $\mathfrak{X}$-structure and a truth relation $\models$ between $\mathfrak{X}$-structures and $\mathfrak{X}$-formulas. As usual, for an $\mathfrak{X}$-structure $\mathfrak{I}$ and a ground instance of an $\mathfrak{X}$-rule $r := \mathsf{h} \leftarrow \mathsf{b}$, $\mathfrak{I} \models r$ iff $\mathfrak{I} \models \mathsf{b} \rightarrow \mathsf{h}$, and $\mathfrak{I}$ is a model of an $\mathfrak{X}$-program P iff $\mathfrak{I} \models r$ for all ground instances of rules r of P. Let $\mathfrak{D}(P)$ denote $\mathfrak{X}$'s notion of declarative semantics, assigning an $\mathfrak{X}$-structure to every $\mathfrak{X}$-program P (for instance, if $\mathfrak{X}$ is first-order logic, $\mathfrak{D}_{FO}(P)$ could be the well-founded model of P).

Definition 1 (State-$\mathfrak{X}$-Structure) A *State-$\mathfrak{X}$-structure* $\mathfrak{I}$ is an $\mathfrak{X}$-structure where the universe $\mathfrak{U} = \mathfrak{U}' \,\dot{\cup}\, \mathfrak{S}$ consists of a classical universe $\mathfrak{U}'$ and a distinguished universe $\mathfrak{S}$, the state space. $\square$

For modeling database behavior, some acyclic ordering on the state space $\mathfrak{S}$ is required. In this paper, we assume $\mathfrak{S} = (\mathbb{N}, <)$. In general, arbitrary state spaces can be modeled, for instance branching models, hierarchical models (as presented for Statelog in [LML96]), or even a possible-worlds semantics can be specified. For states, the notions of "next" state(s), "earlier", and "later", expressed by atoms $S > T$ or $S = T{+}n$ (occurring in the bodies of rules) play an important role.

Definition 2 For a linear state space $\mathfrak{S}$, a state $s \in \mathfrak{S}$, and a State-$\mathfrak{X}$-structure $\mathfrak{I}$ with a universe $\mathfrak{U}' \cup \mathfrak{S}$, the part which is *known in state s*, denoted by $\mathfrak{I}|_{\leq s}$ is obtained by restricting $\mathfrak{I}$ to the universe $\mathfrak{U}' \cup \{s' \in \mathfrak{S} \mid s' \leq s\}$. $\square$

Given an $\mathfrak{X}$-program P, the database evolution is determined by an initial database D and a sequence $E_0, E_1, \ldots$, where each E_i is the set of events occurring in state i, leading to the transition to state $i{+}1$. For simplicity, assume a mapping which maps every set E_i to a E_i' of ground $\mathfrak{X}$-atoms, representing the events in state i.

Example 1 For instance, an event *move x to y* occurring in state s is encoded as $\mathsf{x[moveTo@(s)\rightarrow y]}$. $\square$

Definition 3 A State-$\mathfrak{X}$-structure is a model of P, D, and $E_0, E_1, \ldots, E_n$ (as above) if $\mathfrak{X} \models P \cup D \cup E_0' \cup \ldots \cup E_n'$. The *declarative semantics* of a State-$\mathfrak{X}$-program P wrt. D and $E_0, \ldots, E_n$ as above is defined as $\mathfrak{D}(P \cup D \cup E_0' \cup \ldots \cup E_n')$. $\square$

When describing database *evolution* by a State-$\mathfrak{X}$-program, a model $\mathfrak{I}$ is generated by successively computing its restrictions $\mathfrak{I}|_{\leq 0}$, $\mathfrak{I}|_{\leq 1}$, ... To ensure a proper sequence, if some state is reached, *no* atoms must be derived which contribute to the interpretation relevant to a previous state. Also, as long as there are no events in a state s, no facts about state $s{+}1$ are derived.

Definition 4 A State-$\mathfrak{X}$-program P is *incremental* if for every D, $E_0, \ldots, E_n$ as above, with $\mathfrak{I} := \mathfrak{D}(P \cup D \cup E_0' \cup \ldots \cup E_n')$, for every $s \in \mathbb{N}$, the following holds:

$$\mathfrak{I}_{\leq s+1} = \mathfrak{D}(P \cup \mathfrak{I}_{\leq s} \cup E_s') \ .$$ $\square$

Obviously, incremental programs do not only give a declarative specification, but provided an implementation of $\mathfrak{D}$, can also serve as an implementation of the database system. In the sequel, a sufficient *syntactical* criterion for an $\mathfrak{X}$-program to be incremental is developed based on considering *state terms*: Presume that in every rule, every term s denoting a state also occurs in an atom s:state in the body[3]. Then, s is called a state term.

Definition 5 A *state-ground* instance of an $\mathfrak{X}$-rule is obtained by replacing all state variables of the rule by some elements of $\mathfrak{S}$ (e.g. natural numbers). A state ground instance can be given as an assignment $\beta := \{s_1/n_1, \ldots, s_k/n_k\}$ of elements of $\mathfrak{S}$ to state terms.
A *state-ground model* of an $\mathfrak{X}$-rule is a state-ground instance such that all elements of $\mathfrak{S}$ which are replaced for state terms satisfy the requirements imposed by the rule for states/natural numbers. □

Example 2 For a rule $\mathsf{h(t)} \leftarrow \ldots$, $\mathsf{s{:}state}$, $\mathsf{t{:}\ state}$, $\mathsf{t{>}s}$, $\ldots$, every β: $(s,t) \to \mathbb{N}^2$ is a state-ground instance, but only those β: $(s,t) \to \{(n,m) \in \mathbb{N}^2 \mid m > n\}$ are state-ground models. □

State-$\mathfrak{X}$-rules can be classified wrt. their temporal scope (cf. [LML96]):

Definition 6 (Rule Types for Linear State Spaces)
A State-$\mathfrak{X}$ rule $\mathsf{r} = \mathsf{h} \leftarrow \mathsf{b}$ is

- *global* if there occurs no state term in it.
- *local* if there is at least one state term S occurring in h, and for every state-ground model β of $h \wedge b$, for all state terms S_i, S_j occurring in r, $\beta(S_i) = \beta(S_j)$.
- *progressive* if for every state-ground model β of $h \wedge b$, there is a state term S occurring in h s.t. $\beta(S) \geq \beta(T_i)$ for all other state terms T_i occurring in r.
- *definite progressive* if there is a state term S occurring in h such that for all other state terms T_i occurring in r, there is a $k_i \in \mathbb{N}$ s.t. for every state-ground model β of $h \wedge b$, $\beta(S) = \beta(T_i) + k_i$.
  (*strictly definite progressive*, if each $k_i > 0$ and S does not occur in the body except in an atoms comparing S to other state terms).
  For a definite progressive rule, its *temporal scope* is defined to be the maximum of the above k_i.
- *1-progressive* if it is definite progressive with $k_i = 1$ for all i (analogous *strong 1-progressive*).
- *collective* if h contains no state term, but b contains one or more state terms.
- *backwards* a state-ground model β s.t. there is a state term S occurring in b such that for every state term T occuring in h, $\beta(T) < \beta(S)$.

Note, that local rules are also definite progressive rules with $k_i = 0$ for all i. □

Since the above criteria refer only to elements of S, the above properties can be decided without regarding any object-oriented features, solely by reasoning about the set S and its partial ordering.

[3] or s:x for some subclass x of state

Example 3 The rules

```
O.T[M→X] ← S:state, T:state, O.S[M→X], T= S + 1, not O.change@(S,M)[ ].
O.T[M→Q] ← S:state, T:state, T= S + 1, O[change@(S,M)→Q].
```

are 1-progressive: in every state-ground model, due to the literal T= S + 1, $\beta(T) = \beta(S)+1$. These rules act as frame rules for methods of dynamic objects, i.e., objects o which have an individual instance $o.s$ for every state s.
The rule

```
P[hasTalkedTo→→X] ← P[talksWith@(S)→X], S:state.
```

is collective: at the end of a workflow, for every person P, the method hasTalkedTo gives all persons, P has talked with. □

Obviously, for *specifying* database behavior reasonably, only global and progressive rules make sense (a past database state cannot be changed). Especially, the EDB is computed by 1-progressive rules, and the IDB is computed by local rules. Progressive rules with a scope > 1 are used mainly for transaction definitions, coupling modes etc.

Theorem 1 *Every program P containing only progressive rules and not deriving any facts about a state s+1 if there are no events in state s is incremental.* □

Clearly, backward rules impair the temporal stratification (and also the intuitive understanding of a running database system), they do not fit into the presented approach.

Note, that collective rules are problematic: For P[hasTalkedTo→→X], the answer set is different in every state s, although not directly visible from the rule. On the other hand, for reasoning *about* database behavior, collective and global rules can be quite useful. When modeling and reasoning about database behavior a distinguished set $\mathfrak{M}$ of ground atoms represents the knowledge *about* a process. Here, for instance, for every x, y of the active domain, the atom y[hasTalkedTo→→x] is in $\mathfrak{M}$ (for example, if a workflow should guarantee, that at the end, every person has talked to every other person).

Definition 7 A State-$\mathfrak{X}$-program P is *incremental modulo a set $\mathfrak{M}$ of ground atoms* if for every D, $E_0, \ldots, E_n$ as above, for $\mathfrak{I} := \mathfrak{D}(P \cup D \cup E'_0 \cup \ldots \cup E'_n)$, for every $s \in \mathbb{N}$, the following holds:

$$\begin{aligned}\mathfrak{I}_{\leq s+1} \backslash \mathfrak{M} &= (\mathfrak{D}(P \cup \mathfrak{I}_{\leq s} \cup E'_s)) \backslash \mathfrak{M} \\ &= (\mathfrak{D}(P \cup \mathfrak{I}_{\leq s} \backslash \mathfrak{M} \cup E'_s)) \backslash \mathfrak{M}\ .\end{aligned}$$ □

Theorem 2 *Let P be a State-$\mathfrak{X}$-program containing only global, progressive, and collective rules and $\mathfrak{M}$ a set of ground atoms. Then, P is incremental modulo $\mathfrak{M}$ if $\mathfrak{M}$ contains all ground atoms unifying with heads of collective rules and no atom from $\mathfrak{M}$ is used to derive any state-dependent information.* □

This corresponds to the intuitive understanding: collective rules are not used to derive data of the application domain, but to derive information *about* the running process.

Definition 8 For a State-$\mathfrak{X}$-program P which is incremental modulo a set $\mathfrak{M}$ of ground atoms representing knowledge, a database D, sets $E_0, E_2, \ldots$ of events, and $\mathfrak{I} := \mathfrak{D}(P \cup D \cup E_0 \cup E_1 \cup \ldots)$, the *operational semantics* is defined as the sequence $\mathfrak{I}_{\leq 0} \backslash \mathfrak{M}$, $\mathfrak{I}_{\leq 1} \backslash \mathfrak{M}, \ldots$. □

Theorem 3 *For a State-$\mathfrak{X}$-program P which is incremental modulo a set $\mathfrak{M}$ of ground atoms, the database in state $s{+}1$ can be computed from the database D_s and a set of events E_s as* $D_{s+1} = \mathfrak{D}(P \cup D_s \cup E_s)$. □

Now, after identifying a class of programs suitable for specifying *and* implementing dynamic systems, the details of modeling an evolving system can be considered.

In general, each state consists of several stages $stage_1$, $stage_2, \ldots, stage_n$; for instance, computing the EDB, then computing the IDB, and then deriving the actions to be performed in the transition to the successor state. With this, the rules have to be associated to stages:

Definition 9 For every rule r, a state term S occurring in r is a *governing* state term if for every state-ground model of r, $\beta(S)$ is maximal among the set $\{\beta(T) \mid T \text{ is a state term in } r\}$. □

Note that for local or definite progressive rules, the head contains at least one governing state term. Assume that for every local or progressive rule, at least one governing state term S of the rule is associated to one of the stages via S:stage_i. The partitioning of rules into stages imposes a kind of application-semantic stratification along the temporal axis which corresponds to the execution of a database system: each state represents one (or several successive) fixpoint(s). Since in general, local stratification is undecidable [CB94], for providing an evaluation and implementation for state-$\mathfrak{X}$-programs according to the above ideas, some mechanism is needed which controls the application of rules dependent on the instantiation of their state variables, their association to stages, and the existence of events.

4 Applying the Concept to F-Logic

F-Logic [KLW95] is a deductive, object-oriented database language, combining the advantages of deductive databases with the rich modeling capabilities (objects, methods, class hierarchy, non-monotonic inheritance, signatures) of the object-oriented data model. The syntax and semantics satisfies the requirements stated in Section 2 for exploiting the conceptual flexibility of states in an object-oriented framework. For the full syntax and semantics in all details, the reader is referred to [KLW95, FHKS96]. Here, only the features which are relevant for handling explicite states are presented. The modeling directly exploits F-Logic's inheritance mechanism and dynamic class-membership; the other features – both the rich built-in semantical concepts and the syntactical opportunities – make an intuitive modeling of the application domain possible, which will show up in the examples. F-Logic has been implemented in FLORID (F-LOgic Reasoning In Databases) [FHK$^+$97][4].

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

For a short glance, 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 $\mathcal{V}$ of variables, several auxiliary symbols, containing), (,], [, $\rightarrow$, $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, :, and the usual first-order logic connectives. For convention, object constructors start with lowercase letters whereas variables start with uppercase ones.
- *id-terms* are composed from object constructors and variables. Id-terms are interpreted as elements of the universe.

In the sequel, let O, C, D, M, Q_i, S, S_i, ScM, and MvM stand for id-terms.

- A *method application* is an expression $M@(Q_1, \ldots, Q_k)$.
- if $M@(Q_1, \ldots, Q_k)$ is a method application and O an id-term, the *path expression* $O.(M@(Q_1, \ldots, Q_k))$, denoting the object resulting from applying $M@(Q_1, \ldots, Q_k)$ to O, can occur instead of an id-term.
- 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).
- The following are *object atoms*:
  - $O[ScM@(Q_1, \ldots, Q_k) \rightarrow S]$: applying the *scalar* method ScM with arguments $Q_1, \ldots, Q_k$ to O – as an object – results in S,
  - $O[ScM@(Q_1, \ldots, Q_k) \bullet\!\!\rightarrow S]$: O – as a class – provides the *inheritable scalar* method ScM to its members, which, if called with arguments $Q_1, \ldots, Q_k$ results in S,
  - $O[MvM@(Q_1, \ldots, Q_k) \twoheadrightarrow \{S_1, \ldots, S_n\}]$: applying the *multivalued* method MvM with arguments $Q_1, \ldots, Q_k$ to O results in some S_i.
  - $O[MvM@(Q_1, \ldots, Q_k) \bullet\!\!\twoheadrightarrow \{S_1, \ldots, S_n\}]$, analogous for an *inheritable multivalued* method.
- *Formulas* are built from is-a assertions and object atoms by first-order logic connectives and quantifiers.
- An F-Logic *rule* is a logic rule $\mathsf{h} \leftarrow \mathsf{b}$ over F-Logic's atoms, i.e. is-a assertions and object atoms.
- An F-Logic *program* is a set of rules.

The syntax shows that in F-Logic, entities, described via *id-terms*, act at the same time as classes, objects, and methods. Also, variables can occur at arbitrary positions of formulas. Thus, states can be integrated into F-Logic as first-class citizens like all other entities, and they can be replaced by state variables in all positions.

4.1 Programming Explicit States 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: Every (abstract) state passes through several stages until it is computed completely. This is implemented using a distinguished class state, having subclasses stage$_1$... stage$_n$. Every stage corresponds to a fixpoint computation. When a stage is computed completely, a trigger inserts the facts which create the next stage resp. state. This is implemented via inheritable methods, defining suitable triggers.

The schema in Table 1 gives the rules for handling a four-stage state concept for an active database system, consisting of generating the EDB, calculating the IDB, receiving users' requests, and finally computing the changes to be executed in the transition to the next state:

(A) inheritable methods:	(B) the stage sequence:
stage1::state[ready_edb•→true].	S:stage2 ← S.ready_edb[ ].
stage2::state[ready_idb•→true].	S:stage3 ← S.ready_idb[ ].
stage3::state.	S:stage4 ← S:stage3, S.ready_user[ ].
stage4::state[ready_changes•→true].	T:stage1 ← S.ready_changes[ ], T= S + 1.
0:stage1. % the initialization	

Table 1. Basic Schema for Implementing States

Every fact in (A) – for instance stage1::state[ready_edb•→true] – defines an inheritable method of the subclass stage$_i$, e.g. every member s of class stage1 can inherit the property s[ready_edb→true]. Since deduction precedes inheritance, only when the computation of associated with stage 1 is completed, s[ready_edb→true] is inherited which enables the rule S:stage2 ← S.ready_edb[ ] (B.1), deriving that s also becomes a member of stage2, and the next deductive fixpoint is computed by the rules associated with stage 2. After stage 2 which computes the IDB, stage2::state[ready_idb•→true] (A.2) and S:stage3 ← S.ready_idb[ ] (B.2) define the transition to stage 3. Then, the user interaction takes place, finished by s.ready_user[ ]. This leads to stage 4, where the changes to be executed in the transition to the next state are derived. Finally, s[ready_changes→true] is inherited by (A.3), and the next state $t = s{+}1$ is founded by (B.4).

Example 4 Together with the rules given in Table 1, the following program maintains the invariant that in state s, the method r gives exactly the value s if $s < 10$, then, in state 10, r returns no value, and the program stops. Here, states are objects, providing an EDB-method r and IDB-methods p, del_r and ins_r, representing the requested changes.

```
T[r↠X] ← S[r↠X], not S[del_r↠X], T:stage1, T=S+1.   % frame rules for r.
T[r↠X] ← S[ins_r↠X], T:stage1, T=S+1.

S[q↠X] ← not S[r↠X], 0≤X<10, S:stage2.               % q is {1,...,9}\ r:

S[del_r↠X] ← S[r↠X, q↠Y], Y=X-1, S:stage3.           % derive changes.
S[ins_r↠X] ← S[q↠X, r↠Y], Y=X-1, S:stage3.

S[ready_user→true] ← 0≤S<10, S:stage3.               % run for 10 states.

0[r↠0].                                              % initialization.
```

□

5 Applications and Examples

In this section, we show how different situations can be modeled by different concepts of change.[5] Also, *generic* frame rules are given which model different kinds of dynamic entities as introduced in Section 2.

5.1 Simple Updates to a Database

This example shows a scenario where State-F-Logic's ability of modeling state change by *dynamic classes* provides an elegant and intuitive specification. It reveals only a very simple active behavior by translating user requests into the internal representation and creating an object.

Example 5 Imagine a tram net, consisting of stations and sections. For repairs, some sections have to be closed for some time. For each day, the possible connections are computed. Additionally, for each section, it has to be determined whether it runs hourly or two-hourly at some day (as a default, at weekend days, trams go only two-hourly). But, for single sections (e.g. between the stadium and the railway station on saturdays) it should be possible to deviate from this default.

Here, *dynamic classes* are well-suited for modeling: each section is a static object, the set of open sections is a dynamic EDB class. By implementing the running frequency as an *inheritable* dynamic IDB method, the desired properties can easily be modeled. For dynamic EDB classes, the *generic* frame rules read as follows (insert(E,C,S) means to insert an object E in state S into a dynamic class C; analogous for delete(E,C,S)):

```
% Frame rules for dynamic classes:
E:(C.T) ← T:stage1, T = S + 1, C:edbclass, E:(C.S), not delete(E,C,S).
E:(C.T) ← T:stage1, T = S + 1, C:edbclass, insert(E,C,S).
```

The problem-specific part includes the specification of weekdays, weekend days (using multivalued methods), and the frequency of running the sections for every day. The frequency is implemented as an inheritable method of every class *sections.s*, which is overwritten in case of the section $(stadium, railwStat)$ on saturdays. The computation of the reflexive transitive closure is implemented by local rules.

```
% Problem specific rules:
sections:edbclass.
days[isWeekday→→{0,1,2,3,4}].
days[isWeekend→→{5,6}].
S[weekno→N] ← N = S / 7, S:stage2.          % Here, a state acts as an object.
S[weekday→D] ← D = S - N * 7, S[weekno→N], S:stage2.
sections.S[frequency@(S)•→hourly] ← days[isWeekday→→S.weekday], S:stage2.
sections.S[frequency@(S)•→twohourly] ← days[isWeekend→→S.weekday], S:stage2.
E[frequency@(S)→hourly] ←
        E:(sections.S)[start→stadium; end→railwStat], S[weekday→5], S:stage2.
```

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

```
% Compute reflexive transitive closure
(sections.S)::(connections.S) ← S:stage2.
p(X,Y):(connections.S)[start→X; end→Y] ←
    E:(sections.S)[start→Y; end→X], S:stage2.
p(X,Z):(connections.S)[start→X; end→Z] ←
    E:(sections.S)[start→X; end→Y], P:(connections.S)[start→Y; end→Z], S:stage2.

% Actions
delete(E,sections,S) ← remove(X,Y,S), E:(sections.S)[start→X;end→Y], S:stage4.
insert(e(X,Y),sections,S), e(X,Y):sections[start→X;end→Y] ← add(X,Y,S), S:stage4.
```

The program uses the rules given in Table 1 for handling states which have to be inserted here.
The following interactive requests construct a small database:

```
state 0:    add(cathedral,zoo,0).          add(castle,stadium,0).
            add(stadium,railwStat,0).      0.ready_user[ ].
state 1:    add(airport,railwStat,1).      1.ready_user[ ].
state 2:    remove(castle,stadium,2).      2.ready_user[ ].
```

The following query outputs for every state all open sections with start and end point, and their frequency:

```
?- E:(sections.S), E[start→A, end→B, frequency@(S)→F].
```

As long as T is a state and its EDB is not yet computed, the frame rules are active, deriving T's EDB. When a fixpoint is reached, a trigger fires, setting $T.ready\_edb$ to true. Then, the IDB is computed, giving the set of connections and the frequencies. After this, the users give their update requests via add and remove. When the user has completed his requests, modeled by $T.ready\_user$, the change requests are processed, entering the next state. □

5.2 Active Databases and Integrity Maintenance

Active database behavior, which is often utilized e.g. for integrity maintenance, can also be modeled in State-F-Logic: As in the first example, the user specifies update requests interactively, but now, from these updates and the current database state, the database system derives additional updates. Then, from both the user-requested and the internally derived updates, the next database state is computed.

Example 6 The scenario is as follows, modeling a part of a production planning system: An enterprise produces several types of items, from small screws up to automobiles. There are many compound products, consisting of several parts. The user can change the composition of compound products, say, removing the 145/75 wheels from the parts needed to built some car, instead adding 155/75 wheels. On the other hand, the production of parts can be stopped or started. Obviously, if the production of e.g. a 3"-screw is stopped, the production of all compound products needing 3"-screws also stops.

Here, dynamic multivalued methods are the best way of modeling. Both the production palette and the needs-part relation are represented by EDB-multivalued-methods.

```
% Frame rules for scalar methods:
O[M@(T)→Q] ← T:stage1, T = S + 1, apply(O,M):edbscalar,
                    O[M@(S)→Q], not O.change@(S,M)[ ].
O[M@(T)→Q] ← T:stage1, T = S + 1, apply(O,M):edbscalar, O[change@(S,M)→Q].
% Frame-rules for multi-valued methods:
O[M@(T)→→Q] ← T:stage1, T = S + 1, apply(O,M):edbmultivalued,
                    O[M@(S)→→Q], not delete(O,M,Q,S).
O[M@(T)→→Q] ← T:stage1, T = S + 1, apply(O,M):edbmultivalued, insert(O,M,Q,S).
```

The example shows the flexibility of our approach to deal with different kinds of changes: Since the set of products is assumed to change frequently, is it implemented as a multivalued method which changes with every state, thus it is propagated by a frame rule, considering current updates. On the other side, as the configuration of a certain product changes from time to time, a configuration is modeled as an object, addressed by the dynamic scalar method hasConfig of the product. Thus, for a sequence of states where the configuration does not change, only hasConfig has to be copied to the next state. If the configuration is changed, a new configuration object is introduced, and hasConfig is set to point to it.

```
% Problem Specific rules:
% Semantic Types:
apply(pps,produces):edbmultivalued.
apply(P,hasConfig):edbscalar ← pps[produces@(S)→→P], S:state.
% start or stop production of some part:
insert(pps,produces,P,S) ← start(P,S), S:stage4.
delete(pps,produces,P,S) ← stop(P,S), S:stage4.
% addTo and removeFrom: change Configurations:
change(O,S,hasConfig) ← addTo(O,P,S), S:stage4.
change(O,S,hasConfig) ← removeFrom(O,P,S), S:stage4.
O[change@(S,hasConfig)→newConfig(O,T)] ←
        T= S + 1, change(O,S,hasConfig), S:stage4.
% active behavior:
% if configuration changes, create new configuration object.
newConfig(O,T)[needsPart→→P] ← T= S + 1, change(O,S,hasConfig),
        O.hasConfig@(S)[needsPart→→P], not removeFrom(O,P,S), S:stage4.
newConfig(O,T)[needsPart→→P] ← T= S + 1, change(O,S,hasConfig),
        addTo(O,P,S), S:stage4.
% stop all products which need stopped parts.
stop(P,S) ← P.hasConfig@(S)[needsPart→→Q], stop(Q,S), S:stage4.
```

An example database and an example action sequence could be the following:

```
pps[produces@(0)→→{golf,passat,motor14,motor18,wheel145,screw}].
golf[hasConfig@(0)→newConfig(golf,0)].
passat[hasConfig@(0)→newConfig(passat,0)].
motor14[hasConfig@(0)→newConfig(motor14,0)].
newConfig(golf,0)[needsPart→→{motor14,wheel145}].
newConfig(passat,0)[needsPart→→{motor14,wheel145}].
newConfig(motor14,0)[needsPart→→{screw}].
```

```
removeFrom(passat,motor14,0).  addTo(passat,motor18,0).
  start(wheel155,0).           0.ready_user[ ].
stop(screw,1).                 removeFrom(golf,wheel145,1).
addTo(golf,wheel155,1).        1.ready_user[ ].
```

With the following queries, for every state, all items which are currently produced and which parts they need are given:

```
?- pps[produces@(S)->>P].
?- P.hasConfig@(S)[needsPart->>Q].
```
□

5.3 Other Applications

In continuation of the above examples, the presented concept can be used for process modeling as a specification, implementation, and verification language. For instance, the Alternating-Bit-Protocol has been formulated as a transition system by State-F-Logic rules.

In the above examples, changes are only determined from the current database state. By using progressive rules with scope > 1, it is possible to specify and enforce transactions and dynamic constraints.

Additionally, the proposed extension by states can be employed for evaluating single-state programs wrt. complex logical semantics: Similar to relational databases and Datalog semantics, there is a hierarchy of differently expressive semantics for deductive object-oriented programs, including a well-founded style semantics. Analogous to well-founded Datalog semantics, it can be effectively computed as an alternating fixpoint by using explicite states.

An interesting aspect is the combination with Transaction Logic [BK94], a language dealing with transitions and transactions in a logic programming style. Transaction Logic makes no commitment which formalism to use for describing the interpretation of a state: Any kind of theory can be chosen. Then, the *transition oracle* must be instantiated accordingly. Here, for an arbitrary framework $\mathfrak{X}$ chosen as a state representation language, the transition oracle can be specified and implemented in State-$\mathfrak{X}$. The resulting language provides a powerful language for specification, implementation, and verification of databases and workflow-systems.

6 Conclusion

With its conceptional flexibility, i.e. allowing dynamic objects, dynamic classes, and dynamic methods, the presented approach allows a straightforward modeling of the application domain, thus relieving the user from the burden of encoding into some restrictive formalism. As shown in the examples, the frame rules can be given generically for each concept of object-oriented modeling. Thus, the user can concentrate on the application semantical aspects. With the given implementation scheme for a linear state space, provided an implementation of the underlying single-state framework $\mathfrak{X}$, State-$\mathfrak{X}$ can be used as an implementation language for an object-oriented interactive database system. Thus, a specification also provides an implementation, allowing rapid prototyping and testing. Due to the fact that the state sequence is isomorphic to the natural numbers, *temporal* properties can also be specified and verified by rules. Thus, meta-reasoning about the implemented specification can be done in the same language. Summarizing, the concept – and its instance State-F-Logic – provides an integrated framework for specification, implementation, validation, verification, and runtime checks in a single language.

Acknowledgements.
The authors thank Bertram Ludäscher and Rainer Himmeröder for many fruitful discussions.

References

[AM89] M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8(3), September 1989.

[BCW93] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [TCG$^+$93].

[BK94] A. J. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[CB94] P. Cholak and H. A. Blair. The Complexity of Local Stratification. *Fundamenta Informaticae*, 21(4), 1994.

[CJ90] W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. Intl. Conference on Very Large Data Bases*, pages 432–441, 1990.

[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.

[FHKS96] J. Frohn, R. Himmeröder, P.-T. Kandzia, and C. Schlepphorst. How to Write F-Logic Programs in FLORID, 1996. Available from ftp://ftp.informatik.uni-freiburg.de/pub/florid/tutorial.ps.gz.

[FWP97] A. A. A. Fernandes, M. H. Williams, and N. W. Paton. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205–244, 1997.

[KLS92] M. Kramer, G. Lausen, and G. Saake. Updates in a Rule-Based Language for Objects. In *Proc. Intl. Conference on Very Large Data Bases*, Vancouver, 1992.

[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.

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested Transactions in a Logical Language for Active Rules. In D. Pedreschi and C. Zaniolo, editors, *Proc. Intl. Workshop on Logic in Databases (LID)*, number 1154 in LNCS, pages 196–222, San Miniato, Italy, 1996. Springer.

[TCG$^+$93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases.* Benjamin/Cummings, 1993.

[Zan93] C. Zaniolo. A Unified Semantics for Active and Deductive Databases. In N. W. Paton and M. H. Williams, editors, *Proc. of the 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Workshops in Computing, Edinburgh, Scotland, 1993. Springer.