DYNAMICS'98

Proceedings of the Post-Conference Workshop on

Transactions and Change in Logic Databases

Joint International Conference and Symposium
on Logic Programming JICSLP'98
Manchester, UK
June 20, 1998

Anthony Bonner, Burkhard Freitag, Laura Giordano (Eds.)

Universität Passau
Fakultät für Mathematik und Informatik
D-94030 Passau

Technical Report MIP-9808
June 1998

Preface

The declarative specification of transactions and change is becoming increasingly important in a wide range of applications, including workflow systems, active databases, distributed information systems, cooperative systems, agent-based systems, to name just a few.
Although research on dynamic behaviour (i.e., the evolution of databases or even entire information systems with time) is far from complete, a number of interesting and solid approaches have begun to emerge. However, there is no general picture of the problem space, there are no widely accepted solutions to the central problems, and it is unclear how the various approaches relate to each other.
The need is not only for complex rule bases, but also for standard database functionality, such as concurrent access, transaction isolation and atomicity, large amounts of data, data distribution, recovery from system failures, etc. In addition, many current applications require an active, event-based approach as well as the invocation of external, non-database actions. The problems to be solved therefore span all of logic programming and databases, from theory to implementation.

The informal workshop proceedings are available at

`http://www.fmi.uni-passau.de/~freitag/dynamics_98/`

June 1998

Anthony Bonner
Burkhard Freitag
Laura Giordano

Organizers

Anthony Bonner
University of Toronto, Dept. of Computer Science
Toronto, Ontario M5S 3H5, Canada
bonner@db.toronto.edu

Burkhard Freitag*
University of Passau, Dept. of Computer Science
D-94030 Passau, Germany
freitag@fmi.uni-passau.de

Laura Giordano
University of Torino, Dept. of Computer Science
Corso Svizzera 185, I-10149 Torino, Italy
laura@di.unito.it

(* Workshop Coordinator)

Program Committee

Chitta Baral	Univ. of Texas at El Paso, USA	chitta@cs.utep.edu
Anthony Bonner	Univ. of Toronto, Canada	bonner@db.toronto.edu
Jan Chomicki	Monmouth College, USA	chomicki@moncol.monmouth.edu
Giorgio Delzanno	Max Planck Institut für Informatik, Saarbrcken, Germany	delzanno@mpi-sb.mpg.de
Burkhard Freitag	Univ. Passau, Germany	freitag@fmi.uni-passau.de
Laura Giordano	Univ. Torino, Italy	laura@di.unito.it
Donald Kossmann	Univ. Passau, Germany	kossmann@fmi.uni-passau.de
Georg Lausen	Univ. Freiburg, Germany	lausen@informatik.uni-freiburg.de
Jorge Lobo	Univ. of Illinois at Chicago, USA	jorge@eecs.uic.edu
Alberto Mendelzon	Univ. of Toronto, Canada	mendel@db.toronto.edu
Danilo Montesi	Universitá di Milano, Italy	montesi@dsi.unimi.it
Peter Revesz	Univ. of Nebraska-Lincoln, USA	revesz@tamana.unl.edu
David Toman	Univ. of Toronto, Canada	david@db.toronto.edu

Contents

A Modal Programming Language for Representing Complex Actions

Matteo Baldoni, Laura Giordano, Alberto Martelli, and Viviana Patti
Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (ITALY)
Tel. + 39 11 74 29 111 — Fax. + 39 11 75 16 03
E-mail: {baldoni,laura,mrt,patti}@di.unito.it
URL: http://www.di.unito.it/~argo

Abstract

In this paper we propose a modal approach for reasoning about dynamic domains in a logic programming setting. In particular we define a language, called DyLOG, in which actions are naturally represented by modal operators. DyLOG is a language for reasoning about actions which allows to deal with ramifications and to define *procedures* to build *complex actions* from elementary ones. Procedure definitions can be easily specified in the modal language by introducing suitable axioms.

In the language the frame problem is given a *non-monotonic* solution by making use of persistency assumptions in the context of an abductive characterization. Moreover, a *goal directed proof procedure* is defined, which allows to compute a query from a given dynamic domain description.

1 Introduction

Reasoning about the effects of actions in a dynamically changing world is one of the main problems which must be faced by intelligent agents. Most of the approaches which have been developed to model actions and change allow to reason about the effects of a single action or of a sequence of primitive actions. An interesting extension consists in providing more general ways of composing actions, by defining conditional or iterative actions. This requires to develop a "programming language" for actions, where primitive actions play the role of assignments to variables in conventional programming languages, i.e. their execution causes a state change. Among the most significant achievements in this area, we mention GOLOG [22] and transaction logic [5, 6].

The effects and preconditions of actions are usually described using logic. Thus logic programming is a natural candidate to define a language for reasoning about actions, since it allows to combine reasoning capabilities with control aspects. However pure logic programming lacks an essential aspect: a way of dealing with state changes.

Starting from Gelfond and Lifschitz' seminal work [14], several proposals have been put forward for representing actions in logic programming, which provides simple and well studied nonmonotonic mechanisms. Gelfond and Lifschitz have defined a high-level action language $\mathcal{A}$ and they have given its translation into logic programming with negation as failure. Furthermore, [10] and [11] have proposed translation of (extensions of) the language $\mathcal{A}$ into

abductive logic programming, while [24] has defined an extension of the language $\mathcal{A}$ to deal with concurrent actions, together with a sound and complete translation to abductive normal logic program. Other extensions of the language $\mathcal{A}$ have been proposed in the literature. For instance, the $\mathcal{AR}_0$ language of Kartha and Lifschitz [16] deals with the ramification problem, and for such language a translation is given into a formalism based on circumscription, rather than into logic programming. Furthermore, the language $\mathcal{A}_\mathcal{K}$, presented in [26], is an extension of $\mathcal{A}$ which deals with sensing actions.

Apart from [26] the above mentioned extensions of $\mathcal{A}$ do not cope with the problem of defining complex actions. This issue has been tackled in GOLOG [20, 22], a high level robot programming language which is based on a theory of actions in the situation calculus. The formalization of complex actions in GOLOG draws considerably from dynamic logic [17]. In particular, action operators like sequence, nondeterministic choice and iteration are provided. The definition of GOLOG is rather complex as it is based on second order logic. More precisely, second order logic is needed in defining complex actions, in particular, for iteration and for procedure definitions. While the second order formalization allows properties of GOLOG programs to be proved (through the induction principle), on the other hand it determines a gap between the definition of the language, which is quite general, and its Prolog implementation.

In this paper we present a modal action language DyLOG which allows to reason about complex actions. Its logical characterization is rather simple and very close to the procedural one, which is given by introducing a goal directed proof procedure. Since it allows complex actions, it is strongly related to GOLOG and to the $\mathcal{A}_\mathcal{K}$ language, which also incorporates complex actions. As a difference, rather than referring to an Algol-like paradigm for describing complex actions, DyLOG refers to a Prolog-like paradigm: complex actions are defined through (possibly recursive) definition, given by means of Prolog-like clauses.

The approach that we follow in defining our action theory is also different, as it is based on the adoption of a modal language. Starting from the similarities of GOLOG complex actions with dynamic logics, we argue that a natural definition of complex actions can be provided in a modal setting. Indeed, the adoption of Dynamic Logic or a modal logic to deal with the problem of reasoning about actions and change is common to many proposals, as for instance [9, 28, 8, 30, 15], and it is motivated by the fact that modal logic allows very naturally to represent actions as state transition, through the accessibility relation of Kripke structures.

In particular, in [3] we have presented a modal logic programming language for reasoning about actions, where actions are represented by modalities. The language extends the language $\mathcal{A}$ and it allows to deal with ramifications, by means of "causal rules" among fluents, with incomplete initial states, and with nondeterministic actions. The language relies on an abductive semantics, to provide a nonmonotonic solution to the frame problem, and, when there are no ramifications, it has been proved to be equivalent to the language $\mathcal{A}$. In fact, the semantics of the language $\mathcal{A}$, which is defined in terms of a transition function among states, appears to be quite near to a canonical Kripke structure for our modal language.

The language DyLOG extends the previous language in various ways. First of all we introduce rules to specify action preconditions making use of "existential" modal operators. The main contribution of the paper concerns the extension of the language with "procedures", which allow to define complex actions. We show that this can be easily achieved in modal logics by defining a suitable set of *axioms* of the form $\langle p_1\rangle\langle p_2\rangle \ldots \langle p_n\rangle\varphi \supset \langle p_0\rangle\varphi$.

If p_0 is a procedure name, and the $p_i (i = 1, \ldots, n)$ are either procedure names, or primitive actions or test actions, the above axiom can be interpreted as a procedure definition, which

can then be executed in a goal directed way, similarly to standard logic programs.

Though we do not make use of dynamic logic in our language, and, in particular, we do not introduce iteration, sequence and nondeterministic choice operators, we can model action iteration by recursive procedure definition, action sequences by action composition, and nondeterministic choice among actions by alternative clause definitions. Furthermore, we are able to give a name to complex actions by means of procedure definitions, which are not allowed in Dynamic Logic.

Summarizing, we present a language for reasoning about actions which combines the simplicity of the language $\mathcal{A}$, of which it can be regarded as an extension, with the capability to express complex actions as in GOLOG, building on the standard Kripke semantics of modal logics.

In the next section we present the language DyLOG. To make the comparison with other formalisms easier, we use a syntax close to that of language $\mathcal{A}$. In Section 3 we give a goal directed proof procedure for the language, and in Section 4 we present a logical characterization of DyLOG based on multimodal logics. We conclude by comparing our language with related work.

As it will emerge from Section 4 the adoption of a syntax close to that of language $\mathcal{A}$ is just a matter of notational convenience, and modal formulas could be directly used instead (as done for instance in [3]). Hence DyLOG can actually be regarded as a modal logic programming language.

2 The language DyLOG

The syntax of DyLOG is inspired from the language $\mathcal{A}$ and its extensions in [14, 16]. In the following we use atomic propositions for *fluent names* and we denote by F a *fluent expression*, consisting of a fluent name f or its negation $\neg f$ (note that $\neg\neg f$ is equal to f). Let *true* be a distinguished proposition, then we denote by Fs a *fluent conjunction* defined as follows:

$$Fs \quad ::= \quad true \mid F \mid Fs_1 \wedge Fs_2$$

A *dynamic domain description* in DyLOG consist of three parts: a set of *simple action clauses*, a set of *procedures*, and a set of *observations* on the initial state.

The *simple action clauses* are rules that allow to describe direct and indirect effects of primitive actions on the world. In particular, simple action clauses consist of *action laws*, *precondition laws*, and *causal laws*.

Action laws define direct effects of primitive actions on a fluent and allow to represent actions with conditional effects. They have the form "a **causes** F **if** Fs", where a is a primitive action name, F is a fluent, and Fs is a fluent conjunction, meaning that action a has effect on F, when executed in a state where the *fluent preconditions* Fs hold.

Precondition laws allow to specify *action preconditions*, i.e. those conditions which make an action executable in a state. Precondition laws are of the form "a **possible if** Fs" meaning that when the fluent conjunction Fs holds in a state, execution of the action a is possible in that state.

Causal laws are used to express causal dependencies among fluents and, then, to describe *indirect* effects of primitive actions. They are of the form: "F **if** Fs" meaning that the fluent F holds if the fluent conjunction Fs holds too.

Intuitively, for describing a dynamic domain, we need a set of action and precondition laws for each primitive action together with a (possibly empty) set of causal laws.

Procedures define the behavior of *complex actions*. Complex actions are defined on the basis of primitive actions, and *test* actions. Test actions are needed for testing if some fluent holds in the current state and for expressing conditional complex actions. They are written as "$(Fs)?$", where Fs is a fluent conjunction. A *procedure* in DyLOG is defined as a collection of *procedure clauses* of the form

$$p_0 \ \textbf{is} \ p_1, \ldots, p_n \ \ (n \geq 0)$$

where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either a primitive action, or a test action, or a procedure name (i.e. a procedure call). The sequence of p_i's is the body of the clause. If $n = 0$ we simply write the above procedure clause as p_0. Of course it is possible to define recursive procedures. Moreover, procedure can be *non-deterministic*: there can be more than one procedure clause for a certain procedure name p_0.

A set of *observations* describes what fluents are true in the initial state. They have the form " **initially** F" meaning that the fluent F holds in the initial state (i.e. before any action execution). For simplicity, we require the initial state to be complete, i.e. for each fluent name f either f or $\neg f$ holds in the initial state.

In the following, we use (Π, Obs) to denote a dynamic domain description, where Π is a set of simple action clauses Π_a and procedure clauses Π_p, while Obs is a set of observations on the initial state.

A *goal* in a dynamic domain description in DyLOG has the form:

$$Fs \ \textbf{after} \ p_1, p_2, \ldots, p_n \ \ (n \geq 0) \tag{1}$$

where p_i, $i = 1, \ldots, n$, is either a primitive action, or a procedure name, or a test and Fs is a fluent conjunction. Note that, if $n = 0$ we simply write the above goal as Fs.

Intuitively, the goal (1) corresponds to ask if it is possible to execute the sequence of test, primitive, and complex actions $p_1, p_2, \ldots, p_n$ in such a way that it terminates in a state where Fs is satisfied. Thus, Fs represents a *condition* on the terminating state. A goal succeeds if there is a terminating execution leading to a new state in which Fs holds. As usual in logic programming, execution of a goal returns as a side-effect an answer which is an *execution trace* "$a_1, a_2, \ldots, a_m$", i.e. a primitive action sequence from the *initial state* to the new one.

Though we will make use of variables in the next example, in this paper we develop the proof theory and the modal theory of our language for the propositional case only. An extension to the first order case is shortly addressed in Section 5.

Example 2.1 This example is taken from [5] and simulates the movements of a robot arm in a world of toy blocks. Possible states are represented by means of the fluents $on(x, y)$, $clear(x)$, and $wider(x, y)$ that say that block x is on top of block y, that nothing is on top of x, and that x is wider that y, respectively. We define two complex actions. The first one, $stack(n, x)$, specifies how to stack n arbitrary blocks on top of block x while the second one, $move(x, y)$, defines how to move block x on top of block y.

(1) $stack(N, X)$ **is** $(N > 0)?, move(Y, X), stack(N - 1, Y)$.
(2) $stack(0, X)$.
(3) $move(X, Y)$ **is** $pickup(X), putdown(X, Y)$.

The definition of complex action $stack(N, Y)$ is recursive. The complex action $move(X, Y)$ is defined in terms of the execution of the primitive actions $pickup(X)$ and $putdown(X, Y)$, which are ruled by the following laws:

(4) $pickup(X)$ **possible if** $clear(X)$.
(5) $pickup(X)$ **causes** $clear(Y)$ **if** $on(X, Y)$.
(6) $putdown(X, Y)$ **possible if** $X \neq Y \wedge wider(Y, X) \wedge clear(Y)$.
(7) $putdown(X, Y)$ **causes** $on(X, Y)$ **if** $true$.

The following two causal laws express some indirect effects of the primitive actions *pickup* and *putdown* by representing a fluent dependency between $on(X, Y)$ and $clear(X)$.

(8) $\neg on(X, Y)$ **if** $clear(Y)$.
(9) $\neg clear(Y)$ **if** $on(X, Y)$.

Suppose that we have a world with the five blocks a, b, c, d, and e such that a is wider than b, c, and d, while e is narrower than b, c, and d. Assume that all blocks are alone and, then, clear. We want to stack two blocks on a such that b remains clear. Let Π be the set of clauses (1)-(9) and let Obs be the set of observations that describes the initial situation above. Then the goal $clear(b)$ **after** $stack(2, a)$, succeeds with two solutions; the first one with answer "$pickup(c), putdown(c, a), pickup(e), putdown(e, c)$" and the second one with answer "$pickup(d), putdown(d, a), pickup(e), putdown(e, d)$". Note that the sequence "$pickup(b), putdown(b, a), pickup(e), putdown(e, b)$" is not an answer because it makes $clear(b)$ false after stacking. □

To model persistency we say that if a fluent expression F holds in a state obtained after performing a sequence of primitive actions $a_1, \ldots, a_{m-1}$, and its complement $\neg F$ is not made true by the next primitive action a_m, then the fluent expression F holds after performing the actions $a_1, \ldots, a_{m-1}, a_m$. From a procedural point of view (see Section 3), our non-monotonic way of dealing with the frame problem consists in using *negation as failure* (NAF) in order to verify that the complement of the fluent F is not made true in the state $a_1, \ldots, a_{m-1}, a_m$. In the modal theory we will adopt an abductive semantics to capture persistency.

As an example, let us consider the case of executing the action $pickup(c)$ in a state in which c is on top of d and d is on top of e. Assume that before executing the action the following fluent expressions hold (among others): $clear(c)$, $\neg clear(d)$, $\neg clear(e)$, $on(c, d)$, and $on(d, e)$. Executing the action $pickup(c)$ has the immediate effect of making $clear(d)$ true, and the indirect effect of making $\neg on(c, d)$ true. All the other fluents are not affected by the action execution and, by applying persistency, they will keep their previous values. For instance, since $\neg on(d, e)$ cannot be proved after executing action $pickup(c)$, $on(d, e)$ will persist. On the other hand, the persistency of $\neg clear(d)$ and $on(c, d)$ is blocked.

As the action language in [3], DyLOG allows to deal with action ramifications. The ramification problem is concerned with the additional effects produced by an action besides its immediate effects [27, 25]. In our language ramifications can be expressed by means of causal laws which allow to formalize one-way causal relationships among fluents.

To represent causal laws correctly, we want to avoid their contrapositive use. In the logic programming setting it is quite natural to represent causal rules by making use of explicit negation [13]. When using explicit negation, a negative fluent $\neg f$ is regarded as a new positive atom. Hence, fluent expressions can be regarded as atomic formulae and causal laws can be seen as (modalized) Horn clauses, and thus they are directional implications.

3 Proof Procedure

In this section we introduce a *proof procedure* which allows to compute a query from a given dynamic domain description.

In [3] we have defined an abductive proof procedure in which alternative abductive solutions allow to model non-determinism w.r.t. the effects of primitive actions. The main focus of this paper is on the treatment of complex actions. For this reason and for sake of simplicity we will neither deal with primitive actions with non deterministic effects nor with incomplete initial state specification. To cope with such aspects an abductive proof procedure would be needed similar to the one presented in [3]. On the contrary, in this paper, we propose a proof procedure based on NAF.

The proof procedure reduces the procedure call in the query to a sequence of primitive actions and test actions, and verifies if execution of the primitive actions is possible and if the test actions are successful. To do this, it needs to reason about the execution of a sequence of primitive actions from the initial state and to compute the values of fluents at different states.

The first part of the proof procedure, denoted by " $\vdash_{ps}$ ", deals with execution of procedures (complex actions), primitive actions and test actions. To execute a procedure p we non-deterministically replace it with the body of a procedure clause for it. To execute a primitive action a, first we need to verify if that action is possible by using precondition laws. If so, then we can move to a new state, in which the action has been performed. Finally, to execute a test action $(Fs)?$, the value of Fs is checked in the current state. If Fs holds in the current state, the state action is simply eliminated, otherwise the computation fails.

During a computation, a *state* is represented by a sequence of primitive actions $a_1, a_2, \ldots, a_m$. The value of fluents at a state is not explicitly recorded but it is computed when needed in the computation. The second part of the procedure, denoted by " $\vdash_{fs}$ ", allows to determine the values of fluents in a state.

A goal of the form (1) succeeds if it is possible to execute $p_1, p_2, \ldots, p_n$ (in the order) starting from the current state, in such a way that Fs holds at the resulting state. In general, we will need to establish if a goal of the form (1) holds at a given state $a_1, \ldots, a_m$. Hence, we will write

$$a_1, \ldots, a_m \vdash_{ps} Fs \ \ \textbf{after} \ \ p_1, p_2, \ldots, p_n \text{ with answer } \sigma$$

to mean that the goal Fs **after** $p_1, p_2, \ldots, p_n$ can be proved from the dynamic domain description (Π, Obs) at the state $a_1, \ldots, a_m$ with answer σ, where σ is an action sequence $a_1, \ldots, a_m, \ldots a_{m+k}$ which represents the state resulting by executing $p_1, \ldots, p_n$ in the current state $a_1, \ldots, a_m$. In the following we denote by ε the initial state.

The first three rules define, respectively, how to execute procedure calls, test actions and primitive actions:

1) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p, p_2, \ldots, p_n$ with answer σ if there is a procedure clause $(p \ \textbf{is} \ p'_1, \ldots, p'_{n'}) \in \Pi$ and $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p'_1, \ldots, p'_{n'}, p_2, \ldots, p_n$ with answer σ;

2) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $(Fs')?, p_2, \ldots, p_n$ with answer σ if $a_1, \ldots, a_m \vdash_{fs} Fs'$ and $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $p_2, \ldots, p_n$ with answer σ;

3) $a_1, \ldots, a_m \vdash_{ps} Fs$ **after** $a, p_2, \ldots, p_n$ with answer σ if
there is a precondition law $(a$ **possible if** $Fs') \in \Pi$ such that
$a_1, \ldots, a_m \vdash_{fs} Fs'$ and $a_1, \ldots, a_m, a \vdash_{ps} Fs$ **after** $p_2, \ldots, p_n$ with answer σ;

4) $a_1, \ldots, a_m \vdash_{ps} Fs$ with answer $\sigma = a_1, \ldots, a_m$ if $a_1, \ldots, a_m \vdash_{fs} Fs$.

Rule 4) deals with the case when there are no more actions to be executed. The sequence of primitive actions to be executed $a_1, a_2, \ldots, a_m$ has been already determined and, to check if Fs is true after $a_1, a_2, \ldots, a_m$, proof rules 5)-8) below are used.

The second part of the procedure determines the derivability of a fluent conjunction Fs at a state $a_1, a_2, \ldots, a_m$, denoted by $a_1, a_2, \ldots, a_m \vdash_{fs} Fs$, and it is defined inductively on the structure of Fs:

5) $a_1, \ldots, a_m \vdash_{fs} true$;

6) $a_1, \ldots, a_m \vdash_{fs} F$ if

   a) $m > 0$ and there exists an action law $(a_m$ **causes** F **if** $Fs) \in \Pi$ such that $a_1, \ldots, a_{m-1} \vdash_{fs} Fs$,

   b) there exists a causal law $(F$ **if** $Fs) \in \Pi$ such that $a_1, \ldots, a_m \vdash_{fs} Fs$,

   c) $m > 0$ and $a_1, \ldots, a_{m-1} \vdash_{fs} F$ and **not** $a_1, \ldots, a_m \vdash_{fs} \neg F$;

7) $a_1, \ldots, a_m \vdash_{fs} Fs_1 \wedge Fs_2$ if $a_1, \ldots, a_m \vdash_{fs} Fs_1$ and $a_1, \ldots, a_m \vdash_{fs} Fs_2$;

8) $\varepsilon \vdash_{fs} F$ if (**initially** $F) \in Obs$.

A fluent expression F holds at state $a_1, a_2, \ldots, a_m$ if: either F is an immediate effect of action a_m, whose preconditions hold in the previous state (rule 6(a)); or F can be derived by applying a causal law (rule 6(b)); or F holds in the previous state $a_1, a_2, \ldots, a_{m-1}$ and it persists after executing a_m (rule 6(c)). This last case allows to deal with the *frame problem*: F persists from a state $a_1, a_2, \ldots, a_{m-1}$ to the next state $a_1, a_2, \ldots, a_m$ unless a_m makes $\neg F$ true, i.e. it persists if $\neg F$ fails from $a_1, a_2, \ldots, a_m$. In rule 6(c) **not** represents *negation as failure*.

We say that a goal Fs **after** $p_1, p_2, \ldots, p_n$ *succeeds* from a dynamic domain description (Π, Obs) if it is operationally derivable from (Π, Obs) in the initial state ε by making use of the above proof rules with the *execution trace* σ as answer (i.e., $\varepsilon \vdash_{ps} Fs$ **after** $p_1, p_2, \ldots, p_n$ with answer σ).

4 The logical semantics

In this section we present a logical characterization of DyLOG in two steps. First, we introduce a multimodal logic interpretation of a dynamic domain description which describes the monotonic part of the language. Then, we provide an *abductive semantics* to account for non-monotonic behaviour of the language.

Following the proposal in [3], a dynamic domain description (Π, Obs) in DyLOG is interpreted in a multimodal logic. The basic idea is to use a finite number of modal operators to represent primitive actions, procedure names, and test actions. Moreover, we use a universal

modal operator $[always]$ of type $S4$ to represent any sequence of primitive actions. Differently from [3], each dynamic domain description is interpreted in a particular multimodal logic characterized by a set of axiom schemas determined by its procedure clauses together with a theory based on the set of simple action clauses and the observation.

More precisely, given a dynamic domain description (Π, Obs), let us call $\mathcal{L}_{(\Pi,Obs)}$ the propositional modal logic on which (Π, Obs) is based. $\mathcal{L}_{(\Pi,Obs)}$ contains a finite number of modal operators $[t]$ and $\langle t\rangle$ (universal and existential modal operators, respectively) for each primitive action, procedure name, and test action t that appears in (Π, Obs). All these modalities are normal, that is, they are ruled at least by axiom $K(t) : [t](\psi \supset \varphi) \supset ([t]\psi \supset [t]\varphi)$.

Let us denote by Π_a the set of simple action clauses which are in Π and by Π_p the set of procedure clauses in Π (i.e. $\Pi = \Pi_a \cup \Pi_p$). The set Π_p is interpreted as a set of axiom schemas; more formally, the axiom system for $\mathcal{L}_{(\Pi,Obs)}$ contains an axiom schema of the form $\langle p_1\rangle\langle p_2\rangle \ldots \langle p_n\rangle\varphi \supset \langle p_0\rangle\varphi$, for each procedure clause p_0 **is** $p_1, p_2, \ldots, p_n$ in Π_p, where φ stands for an arbitrary formula, p_0 for a procedure name, and the p_i's for any primitive action, procedure name, or test action. We call $\mathcal{A}_{\Pi_p}$ the set of *axioms* determined by Π_p.

While the axioms determined by Π_p characterize a logic, the action clauses in Π_a and the observations in Obs define a *theory* fragment of the multimodal logic language $\mathcal{L}_{(\Pi,Obs)}$. In particular, we define the following mapping from action laws, precondition laws, causal laws, and observations which belong to Π_a and Obs to modal formulae (we call $\Sigma_{(\Pi,Obs)}$ the theory determined by Π_a and Obs):

$$
\begin{array}{lcl}
a \textbf{ causes } F \textbf{ if } Fs & \rightsquigarrow & [always](Fs \supset [a]F) \\
a \textbf{ possible if } Fs & \rightsquigarrow & [always](Fs \supset \langle a\rangle true) \\
F \textbf{ if } Fs & \rightsquigarrow & [always](Fs \supset F) \\
\textbf{initially } F & \rightsquigarrow & F
\end{array}
$$

where $[always]$ is a modal operator of type $S4$ that represents any sequence of primitive actions, and it is used to denote information which holds in any state. Hence, $[always]$ occurs in front of all simple action clauses that have to hold in any state, while it does not occur in front of observations that have to hold only in the initial state. The axiom system for $\mathcal{L}_{(\Pi,Obs)}$ contains the following *interaction axiom* schema $I(always, a_i) : [always]\varphi \supset [a_i]\varphi$, one for each primitive action a_i in (Π, Obs), which rules the interaction between the modal operator $[always]$ and the modalities $[a_i]$.

The language $\mathcal{L}_{(\Pi,Obs)}$ contains test modalities. Like in dynamic logic [18], if ψ is a proposition then $\psi?$ can be used as a label for a modal operator. As usual, besides the axiom K, the modalities built by the operator "?" are axiomatized by the following: $\langle\psi?\rangle\varphi \iff \psi \wedge \varphi$. Summarizing, a domain description (Π, Obs) is interpreted in a multimodal logic $\mathcal{L}_{(\Pi,Obs)}$ and the simple action clauses in Π_a and the observations in Obs determine a theory $\Sigma_{(\Pi,Obs)}$ in such a logic.

The meaning of a formula in $\mathcal{L}_{(\Pi,Obs)}$ is given by means of a standard Kripke semantics and more details can be find in [1].

The monotonic part of the language does not account for persistency. In order to deal with the frame problem, we introduce a non-monotonic semantics for our language by making use of an abductive construction: abductive assumptions will be used to model persistency from one state to the following one, when a primitive action is performed. In particular, we will assume that a fluent expression F persists through an action unless it is inconsistent to assume so, i.e. unless $\neg F$ holds after the action.

The semantics we define is an extension of the abductive semantics proposed in [3] to deal with complex actions definitions. As a difference with [3] here we have adopted a two valued semantics, instead of a three-valued one. Moreover, here we do not allow for incomplete initial states.

In defining our abductive semantics, we adopt (in a modal setting) the style of Eshghi and Kowalski's abductive semantics for negation as failure [12]. We define a new set of atomic propositions of the form $\mathbf{M}[a_1][a_2]\dots[a_m]F$ and we take them as being *abducibles.*[1] Their meaning is that the fluent expression F can be assumed to hold in the state obtained by executing primitive actions $a_1, a_2, \dots, a_m$. Each abducible can be assumed to hold, provided it is consistent with the domain description (Π, Obs) and with other assumed abducibles. More precisely, in order to deal with the frame problem, we add to the axiom system of $\mathcal{L}_{(\Pi,Obs)}$ the *persistency axiom schema*

$$[a_1][a_2]\dots[a_{m-1}]F \wedge \mathbf{M}[a_1][a_2]\dots[a_{m-1}][a_m]F \supset [a_1][a_2]\dots[a_{m-1}][a_m]F \qquad (2)$$

where $a_1, a_2, \dots, a_m$ $(m > 0)$ are primitive actions, and F is a fluent expression. Its meaning is that, if F holds after action sequence $a_1, a_2, \dots, a_{m-1}$, and F can be assumed to persist after action a_m (i.e., it is consistent to assume $\mathbf{M}[a_1][a_2]\dots[a_m]F$), then we can conclude that F holds after performing the sequence of actions $a_1, a_2, \dots, a_m$.

Given a domain description (Π, Obs), let $\models$ be the satisfiability relation in the monotonic modal logic $\mathcal{L}_{(\Pi,Obs)}$ defined in the previous section (including axiom schema (2)). In the following, $\neg\neg p$ is regarded as being equal to p.

Definition 4.1 *A set of* abducibles Δ *is an* abductive solution *for* (Π, Obs) *if,*

a) $\forall \mathbf{M}[a_1][a_2]\dots[a_m]F \in \Delta$, $\Sigma_{(\Pi,Obs)} \cup \Delta \not\models [a_1][a_2]\dots[a_m]\neg F$

b) $\forall \mathbf{M}[a_1][a_2]\dots[a_m]F \notin \Delta$, $\Sigma_{(\Pi,Obs)} \cup \Delta \models [a_1][a_2]\dots[a_m]\neg F$.

Condition a) is a *consistency* condition, which guarantees that each assumption cannot be assumed if its "complementary" formula holds. Condition b) is a *maximality* condition which forces an abducible to be assumed, unless its "complement" is proved. When an action is applied in a certain state, persistency of those fluents which are not modified by the direct or indirect effects of the action, is obtained by maximizing persistency assumptions.

Since p and $\neg p$ are taken as two different propositions, it might occur that both of them hold in the same state, in an abductive solution. To avoid this, we introduce a consistency condition to accept only those solutions without inconsistent states. We say that an abductive solution Δ is *acceptable* if, for every sequence of actions $a_1, a_2, \dots, a_m$, $(m \geq 0)$, and fluent name p: $\Sigma_{(\Pi,Obs)} \cup \Delta \not\models [a_1][a_2]\dots[a_m]p \wedge [a_1][a_2]\dots[a_m]\neg p$.

Definition 4.2 *Given a domain description* (Π, Obs) *and a goal* Fs **after** $p_1, p_2, \dots, p_n$, *a* solution for the goal in (Π, Obs) *is defined to be an acceptable abductive solution for* (Π, Obs) *such that* $\Sigma_{(\Pi,Obs)} \cup \Delta \models \langle p_1\rangle\langle p_2\rangle\dots\langle p_n\rangle Fs$.

[1] Notice that $\mathbf{M}$ has not to be regarded as a modality. Rather, $\mathbf{M}\alpha$ is the notation used to denote a new atomic proposition associated with α. This notation has been adopted in analogy to default logic, where a justification $\mathbf{M}\alpha$ intuitively means "α is consistent".

According to the definition above, a domain description may have more than one abductive solution. Indeed the semantics above also allows to model primitive actions with non-deterministic effects: executing an action may lead to alternative states, each one modeled by a different abductive solution. Though in our language primitive actions cannot be non-deterministic with respect to immediate effects, they can have non-deterministic effects through ramifications (causal laws). Since this paper is principally devoted to explore how to deal with complex actions definitions, as mentioned above, the proof procedure presented in Section 3 does not account for non-deterministic primitive actions. While we refer to [3] for an example on non-deterministic actions, we just want to observe that the case of multiple solutions occurs when there are negative dependencies in the set of causal rules (as, for instance, a **if** $\neg b \wedge c$ and b **if** $\neg a \wedge c$). These cases cannot be dealt with our proof procedure, since they cause the procedure to enter an infinite loop.

There are also cases when a set of action laws and causal rules Π may have no abductive solution. It may happen, for instance, when Π contains causal rules with negative dependencies, as the rule F **if** $\neg F$. In such a case, it might be impossible to find a set of abducibles satisfying both conditions (a) and (b). The fact that some domain description may have no abductive solution is quite similar to the problem of nonexistence of stable models in logic programs with negation as failure.

The proof procedure computes just one solution, while the abductive semantics may give no solutions or multiple solutions for a given domain description. Hence, we can say that the relation of this logical characterization with the proof procedure above is similar to the relation of stable model semantics with SLDNF. Since existence of abductive solutions is not guaranteed, what we can show is that our procedure gives sound results in the case an abductive solution exists. We can prove the following.

Theorem 4.1 (Soundness) *Let* (Π, Obs) *be a dynamic domain description and let* Fs **after** $p_1, p_2, \ldots, p_m$ *be a goal in* DyLOG. *Then, for all abductive solutions* Δ *for* (Π, Obs), *if* Fs **after** $p_1, p_2, \ldots, p_m$ *succeeds from* (Π, Obs) *then* $\Sigma_{(\Pi, Obs)} \cup \Delta \models \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_m \rangle Fs$.

The proof is omitted for sake of brevity. It can be done by induction on the rank of the derivation of the goal, and it makes use of a soundness and completeness result for the monotonic part of the proof procedure presented in Section 3 with respect to the monotonic part of the semantics.

As in the case of logic programs with negation as failure, syntactic conditions can be defined on domain descriptions which guarantee the existence of abductive solutions. For instance, it is quite natural to define a notion of *stratified* domain description, by imposing that causal laws are such that a fluent cannot depend negatively on itself. Such a restriction would allow to avoid negative loops and thus to assure existence of abductive solutions.

This restriction to stratified rule bases has also the effect to avoid multiple solutions and hence to force primitive actions to be deterministic. In this way the nondeterministic part of the language is confined at the level of procedure definitions.

Notice that the proof procedure in Section 3 does not perform any consistency check on the computed abductive solution. There are cases when a set of action laws and causal laws Π has abductive solutions, but it does not have acceptable ones. This may occur when Π contains alternative action laws for an action a, which may be applicable in the same state, and have mutually inconsistent (immediate or non immediate) effects. For instance, if there

are two action laws for a as follows:

$$a \textbf{ causes } f \textbf{ if } p \qquad a \textbf{ causes } \neg f \textbf{ if } q,$$

when a is executed in a state containing both p and q, the successor state will contain both f and $\neg f$, and hence it will be inconsistent.

Another possible cause for the nonexistence of acceptable solutions comes from inconsistencies in the set of observations Obs.

Conditions can be defined on domain descriptions to guarantee the acceptability (consistency) of abductive solutions (if any). Such conditions generalize the notion of *e-consistency* [10], which has been defined for the language $\mathcal{A}$. In our context, since causal laws are present in a domain description, essentially we have to require that, for any set of action laws (for a given action) which may be applicable in the same state, the set of their immediate and indirect effects (obtainable through the causal laws) is consistent.

The requirements of stratification and e-consistency together ensure that a domain description has a unique acceptable solution (if Obs is consistent). Under these conditions we argue that completeness of the proof procedure in Section 3 can be proved. In fact, the conditions above prevent the proof procedure from entering an infinite loop.

Rather then imposing syntactic restrictions on domain descriptions, an alternative way to solve the problem of non existence of abductive solutions is that of moving to a three-valued semantics, by weakening conditions (a) and (b) above. Such a solution was adopted in [3], for the language with only primitive actions.

5 Conclusions

In this paper we have defined a logic programming language for reasoning about actions, which includes complex actions introduced by means of procedure definitions. In defining the proof theory and the model theory of our language we have only focused on the propositional part of the language. However, both the proof procedure of the language and its modal characterization can be extended to the first order case. In the first order case, variable may occur both in fluent names and in action names (a first order monotonic modal programming language of this kind was studied in [2, 1]).

The problems which arise when addressing the first order case are similar to those which occur in first order logic programs in presence of negation as failure (NAF). More precisely, the *floundering* problem has to be addressed. In our language NAF is used by the proof procedure at step 6c), the one which deals with persistency of fluents. In order to avoid floundering we have to ensure that, during the computation, all the actions in the sequence $a_1, \ldots, a_m$ are groundly instantiated, and, at each state, only ground fluent expressions can be derived. To enforce this, an *allowedness* condition (see [19]) can be put on action laws, preconditions laws and causal laws. Essentially, all the variables occurring in the head of action laws, preconditions laws and causal laws have also to occur in their bodies. Moreover, the set Obs of observations on the initial state must be ground.

The major formalisms introduced for reasoning about dynamic domains and for defining and executing complex actions are Transaction Logic and GOLOG.

Transaction Logic ($\mathcal{TR}$) [5, 6, 7] is a formalism designed to deal with a wide range of update related problems in logic programming, databases and AI. It provides a natural way to define composite transactions as named procedures, by giving them a declarative specification in a

logical first order framework. In $\mathcal{TR}$ complex transactions can be constructed from elementary actions by sequential and concurrent composition. Moreover, non-deterministic transactions and constraints on transactions can be expressed, and both hypothetical and committing actions are allowed.

In our language only the sequential part of $\mathcal{TR}$ can be represented. Essentially our procedures can be defined by sequential composition, and both non-deterministic and recursive procedures are allowed. As a difference with $\mathcal{TR}$, in our language constraints on the execution of transactions cannot be defined, except on the initial and final state of the transactions. This is inherent in the semantic structure of our language, which is not based on "path structures" as the semantic theory of $\mathcal{TR}$.

A major difference with $\mathcal{TR}$ which makes our language much more similar GOLOG than to $\mathcal{TR}$ concerns elementary actions. While $\mathcal{TR}$ is parametric with respect to elementary transitions, which are defined through a "transition base" and whose behaviour is not modelled in the language, in our language effects of elementary actions have to be defined by introducing action laws and causal laws. Moreover, our domain description contains precondition laws which are used to establish if elementary actions are executable in a given state. Simple action clauses provide a very compact definition of actions. Aspects of action theory like action precondition, ramification, persistency have not to be addressed in $\mathcal{TR}$, since the "transition base" provides an extensional description of the behaviour of elementary actions (by describing the effects of actions on all possible states).

As concerns the specification of elementary actions, our language is much more similar to GOLOG [22], though, from the technical point of view, it is based on a different approach. While our language makes use of modal logic, GOLOG is based on classical logic and, more precisely, on the situation calculus. We make use of abduction to deal with persistency, while in GOLOG is given a monotonic solution of the frame problem by introducing successor state axioms. In our case, procedures are defined as axioms of our modal logic, while in GOLOG they are defined by macro expansion into formulae of the situation calculus. Moreover, in DyLOG it is very natural to express, within the goal, conditions on the final state of a procedure execution (e.g. Example 2.1).

As mentioned in the introduction, GOLOG definition is very general and it makes use of second order logic to define iteration and procedure definition. Hence there is a certain gap between the general theory on which GOLOG is based and its implementation in Prolog. In contrast, in this paper we have tried to keep the definition of the semantics of the language and of its proof procedure as close as possible.

In this work we do not deal with sensing actions, that is, actions whose effect is that of changing state of knowledge [29]. Sensing actions are foundamental to reason about conditional plans in presence of incomplete information [21, 4, 26]. In particular, in [26], the authors extend Gelfond and Lifschitz' language $\mathcal{A}$ to reason about complex plans, including conditional and iterations, in presence of sensing. In this case, plans can depend on the outcome of the sensing actions and the language allows to model a form of hypothetical reasoning on complex plans. Instead, in our language, as in [22], we can describe and execute complex plans, that are defined by means of a DyLOG logic program, though plans can only depend on information about the initial state. Nevertheless, in DyLOG we can provide names for complex plans and, moreover, we can express action preconditions and causal relationships among fluents to deal with ramifications. Extending DyLOG to reason about sensing actions is a direction for future work.

In defining our language we have not made use of dynamic logic, and, in particular, we

have not introduced in our language program operators as "sequential composition", "non-deterministic choice"and "iteration". On the other hand, we have the test operator and we define our procedures as sequences of actions. The capability of defining (recursive) procedures makes a further difference with dynamic logic.

An approach to reasoning about action based on dynamic logic has been presented in [9]. There a monotonic solution to the frame problem is adopted. The language in [9] does not deal with procedure definitions but it allows to deal with concurrent actions.

The semantics of our language is defined by following an abductive approach. Hence, our work has strong connections with [10, 11, 24, 23]. All these works address the problem of reasoning about actions in a logic programming abductive setting, by defining translations of the language $\mathcal{A}$ and its extensions to abductive logic programming. These languages do not deal specifically with procedure definitions and we refer to [3] for a more detailed comparison with these approaches. We just want to mention that in [24] a language with concurrent actions is developed. Parallel composition is an operator to construct complex actions that we have not considered in our language and this is a direction in which DyLOG can be extended.

References

[1] M. Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 1998. Available at `http://www.di.unito.it/~baldoni/`.

[2] M. Baldoni, L. Giordano, and A. Martelli. A Framework for Modal Logic Programming. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'96*, pages 52–66, Bonn, 1996. The MIT Press.

[3] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. An Abductive Proof Procedure for Reasoning about Actions in Modal Logic Programming. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Proc. of the 2nd International Workshop on Non-Monotonic Extensions of Logic Programming, NMELP'96*, volume 1216 of *LNAI*, pages 132–150. Springer-Verlag, 1997.

[4] C. Baral and T. C. Son. Approximate Reasoning about Actions in Presence of Sensing and Incomplete Information. In J. Małuszyński, editor, *Proc. of the International Symposium on Logic Programming, ILPS'97*, pages 387–404, Cambridge, 1997. MIT Press.

[5] A. J. Bonner and M. Kifer. Transaction Logic Programming. In *Proceedings of International Conference on Logic Programming, ICLP'93*, pages 257–279, Budapest, Hungary, 1993. The MIT Press.

[6] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, 1994.

[7] A. J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Proc. of Joint International Conference and Symposium on Logic Programming, JICSLP '96*, pages 142–156, Bonn, 1996. The MIT Press.

[8] M. Castilho, O. Gasquet, and A. Herzig. Modal tableaux for reasoning about actions and plans. In S. Steel, editor, *Proc. of European Conference on Planning (ECP'97)*, LNAI, pages 119–130. Springer-Verlag, 1997.

[9] G. De Giacomo and M. Lenzerini. PDL-based framework for reasoning about actions. In *Topics of Artificial Intelligence, AI*IA '95*, volume 992 of *LNAI*, pages 103–114. Springer-Verlag, 1995.

[10] M. Denecker and D. De Schreye. Representing Incomplete Knowledge in Abduction Logic Programming. In *Proc. of 1993 International Logic Programming Symposium, ILPS '93*, Vancouver, 1993. The MIT Press.

[11] P. M. Dung. Representing Actions in Logic Programming and its Applications to Database Updates. In *Proc. of the ICLP'93*, Budapest, 1993.

[12] K. Eshghi and R. Kowalski. Abduction compared with Negation by Failure. In *Proc. of 1989 International Conference on Logic Programming, ICLP '89*, Lisbon, 1989. The MIT Press.

[13] M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In *Proc. of 1990 International Conference on Logic Programming, ICLP '90*, Jerusalem, 1990. The MIT Press.

[14] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[15] L. Giordano, A. Martelli, and C. Schwind. Dealing with concurrent actions in modal action logic. In *Proc. of ECAI'98*, 1998. To appear.

[16] E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing actions: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[17] D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. D. Reidel Publishing Company, 1984.

[18] D. Kozen and J. Tiuryn. Logics of Programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 788–840. Elsevier Science Publishers, 1990.

[19] K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7, 1989.

[20] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. A logical approach to high-level robot programming — a progress report. In B. Kuipers, editor, *Proc. of 1994 AAAI Fall Symposium: Control of the Physical World by Intelligent Systems*, 1994.

[21] H. J. Levesque. What is planning in the presence of sensing? In *Proc. of the AAAI-96*, pages 1139–1146, 1996.

[22] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31, 1997.

[23] R. Li and L. M. Pereira. Representing and Reasoning about Concurrent Actions with Abductive Logic Programs. *Annals of Mathematics and AI*, 1996. Special Issue for Gelfondfest.

[24] R. Li and L. M. Pereira. Temporal Reasoning with Abductive Logic Programming. In *Proc. of the ECAI'96*, 1996.

[25] F. Lin. Embracing Causality in specifying the Indirect Effects of Actions. In *Proc. of International Joint Conference on Artificial Intelligence, IJCAI '95*, Montréal, Canada, 1995. Morgan Kaufmann.

[26] J. Lobo, G. Mendez, and S. R. Taylor. Adding Knowledge to the Action Description Language $\neg$. In *Proc. of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference, AAAI'97/IAAI'97*, pages 454–459, Menlo Park, 1997. AAAI Press.

[27] N. McCain and H. Turner. A Causal Theory of Ramifications and Qualifications. In *Proc. of International Joint Conference on Artificial Intelligence, IJCAI '95*, Montréal, Canada, 1995. Morgan Kaufmann.

[28] H. Prendinger and G. Schurz. Reasoning about action and change. a dynamic logic approach. *Journal of Logic, Language, and Information*, 5(2):209–245, 1996.

[29] R. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proc. of the AAAI-93*, pages 689–695, Washington, DC, 1993.

[30] C. B. Schwind. A logic based framework for action theories. In J. Ginzburg, Z. Khasidashvili, C. Vogel, J.-J. Lévy, and E. Vallduví, editors, *Language, Logic and Computation*, pages 275–291, Stanford, USA, 1997. CSLI publication.

Database Evolution under Non-deterministic and Non-chronological Updates

Yannis Dimopoulos* Antonis Kakas† Suryanarayana M. Sripada‡

Abstract

Data base applications that model aspects of the real-world, should keep track of the changes that are happening in it, and reflect these changes as accurately as possible in the database. If the information about changes, in the form of *updates*, is not supplied to the database in an ideal way, the database needs to have mechanisms that assimilate correctly, whenever possible, this new information.

We propose a model called FlexUp that addresses two cases of non-ideal information supply. The first is the well-known, in the context of deductive databases, problem of *non-deterministic updates* of intensional predicates. The second is the problem of *non-chronological updates*, i.e. updates that come to the system in an order that does necessarily reflect the order in which the corresponding changes happen in the real-world.
The main features of FlexUp are (a) it is conservative in the sense that in case of ambiguity it avoids realizing updates in ad hoc ways (b) it satisfies ACID-like properties and therefore it is in accordance with traditional *transaction systems*.

1 Introduction

Databases in general and deductive databases in particular try to model the real-world and keep track of the changes that are happening in it. The information that the database holds forms the basis for taking decisions by the application system, and therefore it is important that the state of the database reflects the state of the real-world as correctly as possible. When the changes that happen in the world are provided to the database in an ideal way, it is easy to satisfy this criterion of correctly reflecting the state of the world. However, there are cases where this information about the changes in the world is not always ideally provided. The information may be incomplete or it may arrive at the database in an order different from that of the changes in the world. This problem is particularly acute when we have intensional (view) relations in our database which can be updated in several different ways. Let us examine the following illustrative example.

Example 1.1 *Consider the database in an organisation, comprising of the extensional relations Employee(EmployeeName, Department) and Manager(ManagerName, Department) and the intensional relation Manages(ManagerName, EmployeeName) defined as follows:*

*Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany.

†Department of Computer Science, University of Cyprus, Nicosia, Cyprus.
Corresponding author. Email: antonis@turing.cs.ucy.ac.cy

‡Department of Computer Science, University of Aachen, Aachen, Germany.

$Manages(MName, EName) \leftarrow Manager(MName, Dept), Employee(EName, Dept)$
Assume the following data in the extensional relations to start with:
Manager(Bob, Audio), Manager(Mary, Video),
Employee(Tom, Audio), Employee(Peter, Accounts)
Now consider the update request U_1: Add Manages(Mary, Tom)

The intensional relation *Manages* can be updated by any combination of (basic) updates (add and delete) on the extensional relations (*Manager*, *Employee*) that would bring the database in a state that makes the body of the rule hold. Therefore, the specific update request U_1 can be realised in at least 3 ways:

• Tom moved to the Video department: { delete Employee(Tom, Audio), add Employee(Tom, Video) }.

• Mary moved to the Audio department: { delete Manager(Bob, Audio), add Manager(Mary, Audio) }.

• Both Tom and Mary moved to a new department, say Marketing: { delete Manager(Mary, Video), delete Employee(Tom, Audio), add Manager(Mary, Marketing), add Employee(Tom, Marketing)}.

Such an update request U_1 is *non-deterministic* since it is not clear which of the alternatives above reflects the actual change that has occurred in the world and therefore should be used to realize the update request.
The problem of non-deterministic view updates is well-studied in the context of deductive databases (see e.g. [11, 2, 7, 20, 12, 1]). In these works the primary concern is that of realizing a single update working under the implicit assumption that one of the many different realizations can be choosen at the time when the update is presented to the database. Several important criteria for tackling the problem of non-deterministic updates such as preferring minimal updates have been considered e.g. [9, 13, 14, 6]. Although these can be useful, they do not solve the problem completely and can sometimes lead to ad hoc solution. Any arbitrary mechanism for resolution of ambiguity would mis-represent the state of affairs in the real-world, thus compromising the criterion of faithful representation of the world by the database. In this paper we will assume that the information about the changes in the world may be distributed over several updates, and the update that can make them deterministic has not been presented to the database yet. We thus need to consider a sequence of updates and the evolution of the database under this sequence. In [19] the notion of cumulative updates is proposed, where update ambiguities are resolved by combining information from several updates. However, [19] does not propose a transaction model that enables such update mechanisms to be implemented.

Another important case of non-ideal supply of information to the database is when this arrives in an order that does not necessarily correspond to the order in which changes have occurred in the real-world. In such a case we say that the database is receiving *non-chronological updates*. An underlying assumption of traditional update and transaction models is that updates are chronological (full fledged temporal databases are an exception to this rule [18]). This assumption may be too strong for databases in advanced applications. To illustrate the problem consider again the above example and assume that after U_1 we get another update U_2= {*delete* $Manager(Mary, Video)$, *add* $Manager(Mary, Marketing)$}. If this change in the world

recorded by U_2 occurred before that of U_1 then U_1 must be realized in the third way. But if U_2 is treated by the database as a later change, because it is presented to the database after U_1, there is no guarantee that U_1 will be resolved in the same way. Thus, a database after a series of updates may or may not reflect the state of the world, depending upon the order in which such updates are committed.

It is clear from the foregoing that both of the above scenarios require updates to be handled in a more flexible way, by exchanging information between different update requests. In this paper, we develop a model, called FlexUp, suitable for the non-ideal situations of non-deterministic and non-chronological updates by allowing more flexibility in realising them. The basic idea is to allow non-deterministic update requests the possibility to wait (*pending* updates) for more information that will make them deterministic. When a new update is submitted to the system, carrying some deterministic information, this may help some of the pending updates to become deterministic and therefore commit. The model also uses information concerning the time at which the update becomes valid in the real-world, referred to as *valid time* (in temporal database parlance [18]), to allow updates to commit irrespective of the order in which they are submitted while respecting their chronological ordering.

An indirect effect of the non-ideal supply of information to the database is the fact that an update request may be inconsistent at the time when it is presented to the database due to some information that has not arrived at the database yet. The FlexUp model allows such updates to pend in the same way as the non-deterministic updates, waiting for information that would render them consistent. If such information arrives then the update can be realised on the database otherwise it is rejected as in the standard database system.

Looking at the problem of deductive database updates from the point of view of *transactions*, we can identify some important criteria that an update model should satisfy. An update can be seen as a special case of a write-only transaction (i.e. a transaction that does not depend on the state of the extensional database) that changes the state of the database. Under this view, each update/transaction should satisfy ACID-like properties developed by the traditional database community [10]. It turns out that our model satisfies the atomicity and consistency properties. Therefore, the FlexUp model is in accordance with traditional transaction models. Each transaction/update either commits or aborts as a single unit without any dependencies or post conditions. In this sense our approach also contributes ideas for developing transaction models capable to cope with situations of non-ideal information supply. This is an important issue as many kinds of transaction models have been proposed, motivated by advanced applications from various domains [8, 17, 15, 16]. These models attempt to relax the ACID properties of transactions in order to achieve greater functionality and higher efficiency [3]. However, most of the commercial database systems rely only on the standard transaction model satisfying ACID properties [10].
It should be pointed out that, in general, a database is updated through more complex forms of transactions that could involve reading from the database and depending on what is read, different sets of updates are selected. In these cases the problem of handling non-ideal information supply in the database, becomes more complex, and an extension of FlexUp would be needed.

Summarizing, the main features of the FlexUp model are the following.

- It handles non-deterministic updates by keeping these pending until complementary information from other update requests helps resolve the ambiguities.
- It exploits the valid time to handle non-chronological updates.
- Guarantees the consistency of the database according to the specified integrity constraints.
- It enjoys ACID-like properties.

2 The Update model

In this section we describe the FlexUp model and illustrate its behaviour with examples. We start by introducing some basic concepts that are used by FlexUp.

A database contains a set of *extensional* relations. An *extensional* database is a set of ground facts on these extensional relations. A database may also have *view or intensional* relations defined from the extensional relations. The intensional relations are not explicitly stored in the database.

Definition 2.1 *A basic update operation on the database has the form add(p) or delete(p) where "p" is a ground fact of an extensional relation. When add(p) (resp. delete(p)) is effected, the resulting extensional database contains "p" (resp. does not contain "p").*

An update on extensional relations is a set of basic update operations together with an associated time called the valid time of the update. Updates on view (intensional) relations must be translated into a set of basic update operations. Such a translation is not always unique, but instead there can be many different sets of basic operations for the same update (see e.g. [12]). This is the phenomenon of *non-determinism* in the update requests. For instance, in the context of updating intensional relations through abduction (e.g. [12]), every abductive explanation of the update gives us a different way of realizing this update. Abstracting from the details of how view update requests are related and translated to basic operations (these details are not important in the study of this paper) we adopt the following definition of an update on the database.

Definition 2.2 *A* realization *is a set of basic update operations. An* update *is a pair* $U =< R, vt >$ *, where* R *is a set of realizations and* vt *is a time point called the valid time of* U*.*

The *deterministic part* of an update is the set of basic operations that are common in all of its realizations. Hence, no matter how this update will be realized, once it commits, the basic update operations in the deterministic part will be effected on the database.

Definition 2.3 *The* deterministic part $DP(U)$ *of an update* $U =< R, vt >$ *is defined to be* $DP(U) = \bigcap_{i=1}^{n} R_i$*, where* $R_1, \ldots R_n$ *are the realizations of* U *contained in* R*.*

We will assume that for any set of updates with the same valid time the union of their deterministic parts does not contain both $add(p)$ and $delete(p)$ for some p.

Within the FlexUp model an update U, can be in one of three states: *Pending, Commit,* or *Abort*. A FlexUp update is kept pending if it is non-deterministic or inconsistent, until further updates provide enough information to resolve the non-determinism or inconsistency in the pending update. Upon the arrival of another update the FlexUp procedure tries to identify a set of updates that are mutually deterministic and consistent. Then these updates move into the commit status and are committed. The new committed updates start this process again with the possibility of other updates to commit. First we give some preliminary definitions.

Definition 2.4 *A* state *of a database is a triple* $S_i = (DB_0, CU, PU)$*, where* DB_0 *is an extensional database,* CU *is a set of pairs of the form* $< U_j, R_j >$*, where* U_j *is an update committed via the realization* R_j*, and* PU *is a set of pending updates.*

The extensional DB_0 is to be thought as the state at some initial time of reference and CU (respectively PU) the sets of committed (respectively pending) updates at some time after this initial time.

Definition 2.5 *The extensional database wrt some database state* $S = (DB_0, CU, PU)$ *and some valid time* t*, denoted by* DB_t^S*, is the extensional database obtained after committing, in their valid time order, all the realizations in* CU *that are associated with updates with valid time less or equal to* t*. The current extensional database, denoted by* DB^S*, is the database* $DB_{t_f}^S$*, where* t_f *is the greatest valid time amongst the committed updates* CU*.*

We next give the definition of the key notion of *committable* updates which defines which (pending) updates have become ready to commit.

Definition 2.6 *Let* $S = (DB_0, CU, PU)$ *be a state of the database. Then a set of updates* $U \in PU$ *is* committable *wrt the state* S *iff the following conditions hold:*
a) All updates in U *refer to the same valid time* t*,*
b) Every update $U_i \in U$ *has a realization* r_{ij} *such that for every* $add(p) \in r_{ij}$ *(resp.* $delete(p)$*) either* $p \in DB_t^S$ *(resp.* $p \notin DB_t^S$*) or* p *belongs to the deterministic part of another update* $U_k \in U$*.*
c) Let U' *be the set of pairs* $< U_i, r_{ij} >$*, where* $U_i \in U$ *and* r_{ij} *is the realization of the update* U_i *selected in (b). Then for every valid time* $t' \geq t$ *and the state* $S' = (DB_0, CU \cup U', PU - U)$*,* $DB_{t'}^S$ *is consistent.*

This notion of committable combines two elements. The first element, captured by conditions (a, b), simulates the conditions required to hold for the whole set of updates in U to be accepted (and committed), by a conventional update model, when these are not distributed over the different updates but are given all together as one update to the database DB_t referring to the valid time "t" of interest. It therefore requires that any non-determinism that may exist in the updates of this set can be resolved using the collected information distributed over the different updates in U and information from DB_t. The second element (condition c) checks that the realizations of the updates in the set U do not violate any of the integrity constraints at any valid time starting from the valid time of U up to the latest valid time.

Example 2.7 *Consider a database that contains the integrity constraint* $\neg(s, r)$*, the fact* r *in its initial extensional state* DB_0*, with no committed updates yet, and a set of pending updates* $U_1 =< \{(add(p), add(q)), (add(p), add(l))\}, 4 >$, $U_2 =< \{(add(m), add(s)), (add(m), add(t))\}, 4 >$, $U_3 =< \{(add(s))\}, 4 >$*. Assume that the latest update that has arrived at the database is* $U_4 =< \{(add(q))\}, 4 >$*.*

We first note that U_1 and U_2 are pending because they are non-deterministic and that U_3 is pending because it is inconsistent. When U_4 arrives then the set of updates $\{U_1, U_4\}$ is committable since both refer to valid time 4, the first realization of U_1 belongs to the deterministic part of U_4, and if they are realized on DB_0, the extensional database remains consistent. Note that U_3 can not be included in the set of committable updates, as it violates, together with the fact r, the integrity constraint. Consequently, U_2 can not also be committable since it remains

non-deterministic. Notice that if a new update $U_5 =< \{(delete(r), 3 >$ comes after and commits, U_3 and so also U_2 become committable.

We note here that in the definition of committable we require that the valid time of different updates must be exactly the same for these to be able to exchange information among their deterministic parts. This is a formal idealization of a notion of "closeness" of valid times that is dependent on the specific domain of the application for which the database is built. In general, we can employ a suitable temporal reasoning mechanism to project (forwards and backwards) information from updates along the valid time axis.

The basic FlexUp model consists of two main procedures. The first one, called Process-New, is activated when a new update arrives at the system. The procedure checks if the information that the update carries, helps to make some of the pending updates (with valid time equal to the valid time of the new one) committable. When this happens the committable updates are committed and the sets of pending and committed updates are changed accordingly. In detail, the procedure Process-New is as follows.

Assume that the database starts at some initial time with an extensional state DB_0 and is currently in the state $S = (DB_0, CU, PU)$ when the new update, U_{new}, arrives.

Procedure Process-New (U_{new}, PU, CU)
t_e := valid time of T_{new}; $PU := PU \cup U_{new}$;
CS_{t_e} := Subset of updates of PU with valid time t_e that are committable wrt $S = (DB_0, CU, PU)$;
If $CS_{t_e} \neq \emptyset$ **then**
begin
For each $U_i \in CS_{t_e}$ do
$CU = CU \cup < U_i, R_i >$, where R_i is the deterministic part of the update U_i;
$PU := PU - CS_{t_e}$;
Process-Rest (PU, CU);
end

Note that although we do not add the whole realization of a committed update (but only its deterministic part), the rest of the realization belongs to the deterministic part of the other updates which will be committed together with this. As we will see in Section 3, this means that updates that commit have the atomicity property.

When a new set of updates is committed by the procedure Process-New, the state of the database changes. This means that other updates pending either due to non-determinism or inconsistency before this activation of procedure Process-New, may now they also become committable. This job is carried out by the procedure Process-Rest which checks the whole set of the pending updates (in their valid time order) to determine if some of them can commit. This is repeated until no additional updates become committable.

Procedure Process-Rest (PU, CU)
finished:=false;
While not finished **do**
begin finished:=true;
t_e := earliest valid time in PU; t_{last} := latest valid time in PU;
While $t_e \leq t_{last}$ **do**
begin
CS_{t_e} := Subset of updates of PU with valid time t_e that are committable wrt $S = (DB_0, CU, PU)$;
If $CS_{t_e} \neq \emptyset$ **then**
begin
finished:=false;
For each $U_i \in CS_{t_e}$ do
$CU = CU \cup < U_i, R_i >$, where R_i is the deterministic part of the update U_i;
$PU := PU - CS_{t_e}$;
end;
$t_e := t_e + 1$;
end; **end**;

The model as described so far keeps track of all the updates submitted to the system (committed and pending). This might be unrealistic for practical applications. For this reason the system may periodically "time out" all the pending updates with valid time earlier than a time t by removing them from the set of pending updates and aborting them. Also all committed updates with valid time earlier than t are removed form the set of committed updates. This time t then becomes the next initial time and the extensional database DB_t the next initial extensional database. Any new update that arrives and has valid time earlier than t will be aborted without further processing.

Let us illustrate the behaviour of FlexUp with the following examples. The first example is a simple example that shows how non-determinism is resolved using information from later updates.

Example 2.8 *Consider a database of a company that consists of the view relation* new-stock(id, ammount) *where* id *is the identification number of a new stock and* "ammount" *the quantity of the stock. Furthermore assume that there are three extensional relations,* stock(id, ammount) *that records the quantity of each stock, the relation* retail(id) *that records which stock is of retail type, and the relation* wholesale(id) *that records the stock of wholesale type.*

Suppose that at valid time 1 a new update U_1 arrives where we want to update the database with the fact, $new - stock(id123, 50)$, on this view relation. Also assume that the type of stock id123 has not been decided yet, and therefore there are two possible realizations of this update, $U_1 =< \{(add(stock(id123, 50), add(retail(id123)), (add(stock(id123, 50), add(wholesale(id123))\}, 1 >$. So the update is non-deterministic and is added to the pending updates. After some time we decide that id123 will be of wholesale type and give the update $U_2 =< \{(add(wholesale(id123))\}, 1 >$ which then renders U_1 and U_2 committable.

The second example given below is representative of the way non-chronological updates are handled within FlexUp model and how updates can be pending due to inconsistency.

Example 2.9 *Consider a database that consists of the extensional relation* rank(Name, Rank) *that records the rank of the faculty of a university and the relation* pay(Name, Document, Ammount) *that records reimbursements of faculty traveling expenses. The database also contains the integrity constraint that only tenured faculty get reimbursed for traveling and the fact* rank(Tom, Assistant).

Assume that at valid time 5 Tom gets promoted to Associate Professor but the relevant update is not given to the system (until e.g. the Senate approves his promotion). At time 7 Tom goes to a conference and the update request $U_1 =< \{add(pay(Tom, Receipt-101, 1200))\}, 7 >$ comes to the system. This update can not be realized since Tom is still an Assistant Professor in the database and the integrity constraint is violated. So U_1 is added to the pending updates. At some later point the update $U_2 =< \{(add(rank(Tom, Associate)),\ delete(rank(Tom, Assistant)))\}, 5 >$ comes and it commits. Now update U_1 can also commit since it is now consistent wrt the constraint.

We now give a more complex example that demonstrates the general behaviour of FlexUp model in complicated situations.

Example 2.10 *Consider the following set of updates:*
$U_1 =< \{(add(p), add(s)), (add(p), add(t))\}, 1 >$
$U_2 =< \{(add(q))\}, 2 >$ $\qquad$ $U_3 =< \{(delete(p), add(r))\}, 2 >$
$U_4 =< \{(add(s), add(c)), (add(s), add(d))\}, 1 >$
$U_5 =< \{(add(m))\}, 4 >$ $\qquad$ $U_6 =< \{(add(c))\}, 1 >$
$U_7 =< \{(delete(a), add(b))\}, 3 >$ $\qquad$ $U_8 =< \{(add(k))\}, 2 >$
Suppose that the initial extensional database is $DB_0 = \{a\}$ *and that the database has the integrity constraints* $\neg(a, m)$ *and* $\neg(b, k)$. *Also assume that the updates are submitted in the order* $(U_1, U_2, \ldots, U_8)$.

When U_1 arrives to the system it becomes pending since it is non-deterministic and there is not enough information to resolve this non-determinism. Hence the set of pending updates becomes $PU = \{U_1\}$. Then U_2 comes, with valid time 2, and since it is deterministic and consistent it commits. The set of committed updates CU becomes $CU = \{U_2\}$. Then U_3 arrives which is deterministic and consistent and commits. Hence, $CU = \{U_2, U_3\}$ and the current (wrt valid time 2) extensional database is $DB = \{a, q, r\}$. When U_4 is submitted FlexUp sees that its non-determinism can not be resolved with the information we have, hence $PU = \{U_1, U_4\}$. Note that U_4 carries the deterministic information $add(s)$ that can make U_1, which is pending, to commit, but this is not possible since U_4 itself is not committable.
When U_5 arrives, although this is a deterministic update it is inconsistent since a was in the database before valid time 4 and adding m will violate the constraint $\neg(a, m)$. Hence U_5 becomes pending and $PU = \{U_1, U_4, U_5\}$. When U_6 comes we see that U_4 becomes deterministic (since U_4 can now commit on its first realization using deterministic information provided by U_6) and in turn U_1 becomes also deterministic. Hence all of U_1, U_4, U_6 are committable since also they do not violate any of the integrity constraints. The set of committed updates becomes $CU = \{U_1, U_4, U_6, U_2, U_3\}$ and the current (wrt valid time 2) extensional database is $DB = \{a, q, r, s, c\}$. Now U_7 arrives which is deterministic and consistent and commits. This then makes the pending update U_5 consistent, since a is deleted, and therefore commits as well. The current extensional

database is $DB = \{q, r, s, c, b, m\}$ and $PU = \emptyset$. Next U_8 is submitted which is deterministic but it can not be committed since it is inconsistent with the database of valid time 3 that contains (due to U_7) b. Hence it will be added to the pending updates.

3 Properties of the FlexUp Update Model

In this section we present some of the formal properties of FlexUp, and discuss briefly their significance.

We first note that FlexUp keeps the current extensional database consistent. Furthermore, each extensional database referring to any valid time, is also consistent.

Proposition 3.1 *For any resulting state of the database $S = (DB_0, CT, PT)$ and for any valid time t, the extensional database DB_t is consistent.*

The resolution of non-determinism by FlexUp *does not involve any arbitrary choice* within the set of committed updates. The realization selected for a particular update is supported by the deterministic information in other committed updates.

Proposition 3.2 *Let DB denote the extensional database resulting after a sequence of updates SU has been submitted to FlexUp on an database with initial state DB_0. Then $p \in DB$ (resp. $p \notin DB$) if $p \in DB_0$ (resp. $p \notin DB_0$) or $add(p)$ (resp. $delete(p)$) belongs to the deterministic part of of a committed update $U \in SU$.*

Another property of FlexUp is that this assimilates information about changes in the world while *respecting the time order in which these changes happened in the world* (their valid times). The following two results demonstrate this property and show an order independence of FlexUp amongst the committed updates.

Proposition 3.3 *Let SU be a sequence of updates ordered by their valid times such that if submitted to FlexUp all the updates in SU commit resulting to a final extensional database DB_f. Let SU' be a permutation of SU submitted to FlexUp. Then FlexUp will commit all the updates in SU' resulting to the same final extensional database DB_f.*

Let us denote by $Do(DB, R)$ the extensional database obtained when effecting a realization R on the database DB.

Theorem 3.4 *Let SU be a sequence of updates submitted to FlexUp on an initial state DB_0. Let also $U' \subseteq SU$ be the updates that have been committed within a time period t and $(U_1, \ldots, U_n)$ a permutation of the updates in U' ordered by their valid time. Then there exists a sequence $(R_1, \ldots, R_n)$ where each R_i is a realization of the update U_i, such that the extensional database, DB, at the end of period t, is $Do(\ldots Do(Do(DB_0, R_1), R_2) \ldots R_n)$.*

This last theorem shows that the set of updates U' can be given in any order but when they commit their effect is as if they were given in their valid time order. The theorem also shows that at any time *the set of committed updates satisfies the atomicity property*, in the sense that each committed update has in effect been completely realized (by one of its realizations) on the initial database.

The above results can be put together to give the following corollary that characterises uniquely the extensional database at any time.

Corollary 3.5 *Let SU be a sequence of updates submitted to FlexUp on an initial database DB_0. Let also $U' \subseteq SU$ be the updates that have been committed within a time period t and $(U_1, \ldots, U_n)$ a permutation of the updates in U' ordered by their valid time. The extensional database DB, at the end of the period t is equal to $Do(\ldots Do(Do(DB_0, DP(U_1)), DP(U_2)) \ldots DP(U_n))$, where $DP(U_i)$ is the deterministic part of the update U_i.*

Hence, FlexUp simulates, for the committed updates, an ideal supply of information.

Finally, we note that when information arrives at the database in an ideal way so that all the updates can commit under the standard update model, FlexUp will exhibit the same behaviour. For example, if all the updates arrive in their valid time order and are all consistent (but not deterministic) then all updates committed under the standard model will also commit under FlexUp.

4 Concluding remarks

In this paper we introduced the FlexUp model that supports non-deterministic and non-chronological updates. FlexUp resolves update ambiguities and assimilates information in non-chronological order. We showed that FlexUp has the following properties: (a) It does not make any arbitrary choices in its resolution of non-determinism. (b) It assimilates information about changes in the world while respecting the chronological order of these changes. (c) When the update requests are chronological it performs as well as the standard model. (d) It satisfies ACID-like properties.

We emphasise that FlexUp is a conservative extension of the standard update model: we have used a very simple temporal reasoning model to transfer information from one update to another. In part, this model was motivated by the desire to stay close to traditional transaction systems. But the main reason for its simplicity is the pragmatic need for the high-level of reliability and accuracy of the information that is held by a database: in practice, we can not have a recovery mechanism from incorrect data that has been extracted from the database and acted upon. Our work has shown that even with such simple temporal reasoning mechanisms it is a non-trivial task to define a suitable update model with the desired database properties. Further study is needed to understand how more advanced forms of temporal reasoning, that would allow a more general form of exchange of information between updates at different valid times can be incorporated in the update model allowing the same degree of reliability. The non-monotonic nature of such frameworks often leads to the need of belief revison that may not be possible in the pragmatic context of databases and hence these frameworks will need to be specially adapted and refined for their use in database evolution.

References

[1] Halfeld Ferrari Alves, M., Laurent, D., Spyratos, N., Update Rules in Datalog Programs. *Proc. 2nd Int. Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'95*, LNAI 928, Springer Verlag, 1995.

[2] Bry, F., Intensional updates: abduction via deduction. *7th ICLP*, Jerusalem,561–575, 1990.

[3] Chrysanthis, P.K., Ramamritham, K. ACTA: A framework for specifying and reasoning about transaction structure and behaviour. In *Proc. ACM SIGMOD Conference*, 1990.

[4] Console, L., Sapino, M.L., Theseider Dupré, D., The role of abduction in database view updating. *Journal of Intelligent Information Systems*,Vol 4, pp. 261-280, 1995.

[5] Date and White *An introduction to DB2*, 1989, Prentice Hall.

[6] Dekhtyar, M., Dikovsky, A., Spyratos, N., On Conservative Enforced Updates. *Proc. 4th Int. Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'97*, Springer Verlag, 1997.

[7] Decker, H., Drawing Updates from Derivations *In proceedings of ICTP'90.*

[8] Elmagarmid, A.K. (ed.). *Database Transaction Models for Advanced Applications*, Morgan Kaufmann.

[9] Fagin, R., Ullman, J. and Vardi, M., On the Semantics of Updates in Databases *Proc. 2nd ACM PODS*, 1983.

[10] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques* Morgan Kaufmann, 1993.

[11] Guessoum, A. and Lloyd, J.W., Updating Knowledge Bases *New Generation Computing*, 1990.

[12] Kakas, A. C., Mancarella, P., Database updates through abduction. *Proc. 16th International Conference on Very Large Databases, VLDB'90*, Brisbane, Australia (1990)

[13] Katsumo, H., Mendelzon, A., Propositional Knowledge Base Revision and Minimal Change *Artificial Intelligence, Vol 52, 1991.*

[14] Marek, V.W., Truszcynski, M. Revision Programming, Database Updates and Integrity Constraints *International Conference on Database Theory, ICDT'95* LNCS 893, 1995.

[15] Moss, J.E.B. Nested Transactions: An approach to reliable distributed computing. Ph.D.Thesis, MIT, April 1981.

[16] Nodine, M., Zdonik, S. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings 10th VLDB Conference*, 1984.

[17] Pu, C., Kaiser, G., and Hutchinson, N. Split Transactions for Open-Ended activities. In *Proc. 14th VLDB Conference*, August 1988.

[18] Snodgrass, R. et al. *The TSQL2 Language*, Kluwer Publishers, 1995.

[19] Sripada, S.M. and Wuethrich, B. Cumulative Updates. In *Proc. of the 20th VLDB Conference*, September 1994.

[20] Tomasic, A., View Update Annotation in Definite Deductive Databases *Proc. ICDT'88*, Springer Verlag, 1988.

Knowledge Assimilation and Proof Restoration through the Addition of Goals

Hisashi Hayashi

Department of Computer Science
Queen Mary and Westfield College
University of London
Mile End Road, London E1 4NS, U.K.
email: hisashi@dcs.qmw.ac.uk

Abstract

Normal proof procedures in abductive logic programming assume that a given program does not change until the proof is completed. However, while a proof is being constructed, new knowledge which affects the proof might be acquired. This paper addresses two important issues: 1. How is it confirmed that the proof being constructed is not affected by the addition of a clause? 2. If affected, how are the invalid parts of the proof restored? The abductive proof procedure used in this paper is Kakas and Mancarella's procedure and is extended to prepare for proof checking and proof restoration. It is shown that any invalid part of a proof can be restored if some additional goals are solved. These additional goals can be added before a proof is completed.

Keywords: Abduction, Logic Programming, Knowledge Acquisition.

1 Introduction

In most literature on logic programming, the design of a proof procedure does not take into account the dynamic nature of a program. If this kind of proof procedure is used, every time the program is revised and changed, the goal has to be proved again under the new program. However, it is easy to imagine that if the program is changed only a little bit, the old proof is still valid in a lot of cases.

The purpose of this project is to reuse the same proof unless it becomes invalid and to change only limited parts of the proof if the proof becomes invalid. Indeed

in a lot of cases, it takes time to make a proof. When a whole proof or a part of a proof needs to be constructed within limited time, it is not a good idea to make a brand new proof from the beginning every time programs are corrected.

Consider planning in robotics. Suppose that a robot is always gathering information through its sensors and that the database of the robot is always changing.

Suppose that the robot starts constructing a plan at time, say, 10:00:00 and finishes at 10:01:15. If the plan is made based only on the information at 10:00:00, it is impossible to say that the plan is still valid at 10:01:15 because the database might be updated between 10:00:00 and 10:01:15. This means that the robot is not sensing the outer world while making the plan. (Even if the robot senses the outer world, it is not using new information during that time.)

For this reason, some criteria are needed by which to confirm that the plan is still valid after updating the database. Also it is essential to reuse some parts of the plan to avoid reconstructing a brand new plan and save time.

In *partial order planning*, *causal links* or *protected links* are used for replanning. (See standard textbooks such as [20].) In this paper, a more general theory which can be applied to any proof in abductive logic programming will be introduced. Of course, it is assumed that the contents of the databases (or programs) being used are updated as the time goes on.

The proof procedure discussed in the present paper is for abductive logic programming. The abductive proof procedure adopted in this paper is based on the Eshghi-Kowalski (E-K) procedure [4] and the Kakas-Mancarella (K-M) procedure [11, 12]. The E-K procedure is an extension of SLD resolution and the K-M procedure is an extension of the E-K procedure.

The rest of the paper is organised as follows. In Section 2, abductive logic programming is introduced briefly. In Section 3, the K-M procedure is introduced. The procedure is extended to prepare for the expansion of the program. Based on the additional information which is obtained by the extended K-M procedure, a proof restoration procedure is introduced in Section 4 which is the most important section. This proof restoration procedure can also check the validity of a proof. After showing a result of experiments in Section 5 and the related works are discussed in Section 6, the conclusion is discussed in Section 7.

2 Abductive Logic Programming

Before defining an abductive framework, some basic words are defined. It is assumed that the readers of this paper are familiar with the concepts of logic programming.

In this paper, variables are expressed by letters and numerals starting with an upper case letter. Constants, predicate symbols, and function symbols are expressed by letters and numerals starting with a lower case letter. Intuitively,

the negation *not* is so called *negation as failure* but it is not, strictly speaking, in the sense that negative literals are treated as hypotheses.

Definition 2.1 *A* **literal** *is either a positive literal or a negative literal. A* **positive literal** *is an atom. A* **negative literal** *is of the form not P where P is an atom. The* **contrary** *of a positive literal P is the negative literal not P. The contrary of a negative literal not P is the positive literal P. The contrary of a literal L is expressed as L^*.*

Definition 2.2 *A* **clause** *is of the form $L \Leftarrow L_1, ..., L_n$ where L is an atom and $L_1, ..., L_n$ are literals. The clause $L \Leftarrow$ can be expressed as L.*

Note that $\Leftarrow$ is used only for clauses of logic programs and $\leftarrow$ refers to the implication of classical logic.

Definition 2.3 *An* **abductive framework**[1] *is a tuple $\langle P, Ab \rangle$, where P is a* **program**, *a set of clauses and Ab is a set of* **abducibles**, *a set of literals, such that P does not include a clause of the form $L \Leftarrow L_1, ..., L_n$ such that L is an element of Ab. An abducible is a* **positive abducible** *if it is a positive literal. An abducible is a* **negative abducible** *if it is a negative literal.*

To avoid transforming the negative literal *not* p to the positive literal $neg(p)$ which is called a **non-base abducible** in [11], all negative literals whose atoms are mentioned in P or Ab are normally regarded as abducibles unless otherwise mentioned.

As far as the semantics of the procedure in the present paper is concerned, the completion semantics is used. The completion of the program P and abducibles Ab is defined as follows.

Definition 2.4 *Given a program P and a set of literals Ab (abducibles), $comp(P, Ab)$ is the least set such that for any positive literal q which is not in Ab,*

- *all the clauses in P defining q are:*

$$q \Leftarrow L_{1,1}, ..., L_{1,n_1} \ \ ... \ \ q \Leftarrow L_{m,1}, ..., L_{m,n_m}$$

if and only if

- *$comp(P, Ab)$ includes:*

$$q \leftrightarrow (L_{1,1} \wedge ... \wedge L_{1,n_1}) \vee ... \vee (L_{m,1} \wedge ... \wedge L_{m,n_m})$$

[1] Integrity constraints are not included for simplicity.

3 An Abductive Proof Procedure

In this section, the abductive proof procedure based on the procedure introduced by Kakas and Mancarella [11, 12] is introduced. The Kakas-Mancarella (K-M) proof procedure is an extension of the Eshghi-Kowalski (E-K) proof procedure [4] which simulates SLDNF by abduction. The E-K procedure is an extension of SLD resolution.

In the original definition, an abductive derivation and a consistency derivation are defined separately. In the following definition, however, they are defined as a single derivation. Although the following definition is the same as the original definition in essence, a new set called defending set is used to prepare for the expansion of the given program. Recording additional information in a defending set does not affect the proof procedure at all. Therefore, the following proof procedure works in the same way as the original one.

Definition 3.1 *The* **resolvent** *of the set of literals* $\{P, L_1, ..., L_n\}$ *on* P *by the clause* $P \Leftarrow L_{n+1}, ..., L_m$ *is the set of literals* $\{L_1, ..., L_m\}$.

Definition 3.2 *A* **goal** *is either of the form* (pos, Ls) *where* Ls *is a set of literals, or of the form* (neg, Cs) *where* Cs *is a set of sets of literals.*

Intuitively, $(pos, \{L_1, ..., L_n\})$ means that the goal is to prove $L_1 \wedge ... \wedge L_n$ and $(neg, \{\{L_{1,1}, ..., L_{1,n_1}\}, ..., \{L_{m,1}, ..., L_{m,n_m}\}\})$ means that the goal is to prove $\neg(L_{1,1} \wedge ... \wedge L_{1,n_1}) \wedge ... \wedge \neg(L_{m,1} \wedge ... \wedge L_{m,n_m})$.

Definition 3.3 *A* **goal list** *is of the form* $[G_1, ..., G_n]$ *where* $G_1, ..., G_n$ $(n \geq 0)$ *are goals.*

Intuitively, the goals are proved in this order. (i.e. from G_1 to G_n) Note that [] is one of goal lists.

Definition 3.4 *A* **defending set** *is a set whose elements are of the form* (L, Ls) *where* L *is a literal and* Ls *is a set of literals.*

Intuitively, the element of a defending set (L, Ls) means that the set of literals $\{L\} \cup Ls$ (which means that $\leftarrow L \wedge Ls$ has to be proved) and the positive literal $L (\in \{L\} \cup Ls)$ were selected when the consistency derivation rule c1 was applied. (The consistency derivation rules will be defined soon.) The procedure tried to prove that L does not hold and if L is proved to be true, one of the literals in LS has to be proved to be false. Therefore, when a clause defining L is added to the program, the defending sets have to be checked.

Definition 3.5 *A* **goal set** *is of the form* (GL, Δ, R) *where* GL *is a goal list,* Δ *is a set of literals (abducibles), and* R *is a defending set*[2].

[2]In [7], a set of defending sets is called *Reject* and R is used often to refer to a set of defending sets.

The goal set (GL, Δ, R) means that all the goals in GL have to be proved with the abducibles in Δ assumed to be true. R will be used to restore the correctness when a clause is added to the program.

Definition 3.6 *A* **derivation** *from (G_1, Δ_1, R_1) to (G_n, Δ_n, R_n) under the abductive framework $\langle P, Ab \rangle$ is a sequence of goal sets:*

$$(GL_1, \Delta_1, R_1), ..., (GL_n, \Delta_n, R_n)$$

such that for each $i (1 \leq i \leq n-1)$, GL_i is of the form $[F_{i,1}, ..., F_{i,x_i}]$ $(x_i > 0)$, if $F_{i,1}$ is of the form $(pos, \{L_1, ..., L_k\})$ $(k > 0)$ and $L_u (\in \{L_1, ..., L_k\})$ is selected, then $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ is obtained by one of the following **abductive derivation rules**,

a1 *If L_u is not an abducible in Ab and a clause in P whose head is L_u is chosen, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(pos, Ls), F_{i,2}, ..., F_{i,x_i}]$ where Ls is the resolvent of $\{L_1, ..., L_k\}$ on L_u by the chosen clause.*

a2 *If L_u is an abducible in Ab and $L_u \in \Delta_i$, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(pos, Ls), F_{i,2}, ..., F_{i,x_i}]$ where $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.*

a3 *If L_u is a positive abducible in Ab, $L_u \notin \Delta_i$, and not $L_u \notin \Delta_i$, then $\Delta_{i+1} = \{L_u\} \cup \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(pos, Ls), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ where $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.*

a4 *If L_u is a negative abducible in Ab, $L_u \notin \Delta_i$, and $L_u^* \notin \Delta_i$, then $\Delta_{i+1} = \{L_u\} \cup \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(neg, \{\{L_u^*\}\}), (pos, Ls), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ where $Ls = \{L_1, ..., L_{u-1}, L_{u+1}, ..., L_k\}$.*

else if $F_{i,1}$ is of the form $(neg, \{C_1, ..., C_k\})$ $(k > 0)$, $C_v (\in \{C_1, ..., C_k\})$ is selected, C_v is of the form $\{L_{v,1}, ..., L_{v,y}\}$ $(y > 0)$, and $L_{v,l} (\in \{L_{v,1}, ..., L_{v,y}\})$ is selected, then $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ is obtained by one of the following **consistency derivation rules**,

c1 *If $L_{v,l}$ is not an abducible in Ab, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = \{(L_{v,l}, \{L_{v,1}, ..., L_{v,l-1}, L_{v,l+1}, ..., L_{v,y}\})\} \cup R_i$, and $GL_{i+1} = [(neg, Cs \cup \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$ where Cs is the set[3] of all the resolvents of C_v each of which is obtained by resolving C_v on $L_{v,l}$ by a clause in P.*

c2 *If $L_{v,l}$ is an abducible in Ab and $L_{v,l} \in \Delta_i$, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_v \setminus \{L_{v,l}\}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.*

c3 *If $L_{v,l}$ is an abducible in Ab and $L_{v,l}^* \in \Delta_i$, then $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, and $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.*

[3]This can be an empty set.

c4 *If* $L_{v,l}$ *is a positive abducible in Ab,* $L_{v,l} \notin \Delta_i$*, and not* $L_{v,l} \notin \Delta_i$*, then* $\Delta_{i+1} = \{not\ L_{v,l}\} \cup \Delta_i$*,* $R_{i+1} = R_i$*, and* $GL_{i+1} = [(neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.

c5 *If* $L_{v,l}$ *is a negative abducible in Ab,* $L_{v,l} \notin \Delta_i$*, and* $L^*_{v,l} \notin \Delta_i$*, then* $\Delta_{i+1} = \Delta_i$*,* $R_{i+1} = R_i$*, and* $GL_{i+1} = [(pos, \{L^*_{v,l}\}), (neg, \{C_1, ..., C_{v-1}, C_{v+1}, ..., C_k\}), F_{i,2}, F_{i,3}, ..., F_{i,x_i}]$.

and else if $F_{i,1}$ *is of the form* $(X, \emptyset)$*, where* X *is either pos or neg, then* $(GL_{i+1}, \Delta_{i+1}, R_{i+1})$ *is obtained by the following* **controlling derivation rule**.

cdr $\Delta_{i+1} = \Delta_i$, $R_{i+1} = R_i$, $GL_{i+1} = [F_{i,2}, ..., F_{i,x_i}]$

From a derivation, it is possible to trace the proof construction. Abductive derivation rules try to prove that the selected literal is true. Consistency derivation rules try to prove that the selected literal is false or at least one literal in the selected set of literals is false.

For *safety* reason, in the predicate case, it is prohibited to select a nonground abducible when one of the rules [a2], ..., [a4], [c2], ..., [c5] is applied. This restriction prevents so called "floundering". Note that even if the selected (non-abducible) literal is not ground, [a1] and [c1] can be applied by using *unification.*

Definition 3.7 *A derivation from* $([(pos, \{L_1, ..., L_n\})], \emptyset, \emptyset)$ *to* $([], \Delta_m, R_m)$ *under the abductive framework* $\langle P, Ab\rangle$ *is a* **(complete) proof** *for* $L_1 ... L_n$ *under the abductive framework* $\langle P, Ab\rangle$

Definition 3.8 *A derivation from* $([(pos, \{L_1, ..., L_n\})], \emptyset, \emptyset)$ *to* (GL_m, Δ_m, R_m) *under the abductive framework* $\langle P, Ab\rangle$ *is an* **incomplete proof** *for* $L_1 ... L_n$ *under the abductive framework* $\langle P, Ab\rangle$.

Note that complete proofs are the limiting case of incomplete proofs.

Example 3.1 Consider the following program where negative literals, $ab(V)$, and $super(W)$ are abducibles.

clause 1: $fly(X) \Leftarrow bird(X), not\ ab(X)$
clause 2: $fly(penguin) \Leftarrow super(penguin)$
clause 3: $bird(penguin)$

A proof for $not\ fly(penguin)$ is as follows.

ab 1 $?not\ fly(penguin)$

co 1.1 $?\{fly(penguin)\}$ $(not\ fly(penguin) \in \Delta)$

co 1.2 $?\{bird(penguin), not\ ab(penguin)\}, \{super(penguin)\}$
$((fly(penguin), \emptyset) \in R)$

co 1.3 $?\{not\ ab(penguin)\}, \{super(penguin)\}$
$((bird(penguin), \{not\ ab(penguin)\}) \in R)$

ab 1.3.1 $?ab(penguin)$

ab 1.3.2 success $(ab(penguin) \in \Delta)$

co 1.4 $?\{super(penguin)\}$

co 1.5 failure $(not\ super(penguin) \in \Delta)$

ab 2 success

4 Clause Assimilation and Proof Restoration

When or after a proof is constructed by the abductive proof procedure shown in the previous section, if a new clause is added to the given program, the validity of the complete or incomplete proof is no longer guaranteed. However, by proving additional goals, the validity of the proof can be restored.

Definition 4.1 An **added goal** by checking the clause $L \Leftarrow L_1, ..., L_n$ and the set of defending sets R is a set of literals of the form $\{L_1, ..., L_m\}$ such that $(L, \{L_{n+1}, ..., L_m\}) \in R$.

Theorem 4.1 *For any incomplete proof for* $L_1, ..., L_k$ *from* $([(pos, \{L_1, ..., L_k\})], \emptyset, \emptyset)$ *to* $([F_{t,1}, ..., F_{t,n}], \Delta_t, R_t)$ *under the abductive framework* $\langle P, Ab\rangle$ *and for any clause* C *which does not define an atom in* Ab*, if there exists a derivation from* $([(neg, AGs), F_{t,1}, ..., F_{t,n}], \Delta_t, R_t)$ *to* $([], \Delta_s, R_s)$ *under the abductive framework* $\langle P \cup \{C\}, Ab\rangle$*, where* AGs *is the set*[4] *of all the added goals by checking* C *and* R_t*, then* $comp(P \cup \{C\}, Ab) \cup \Delta_s \models \{L_1, ..., L_k\}$.

> **Proof**[5]*:* The only derivation rule which might be affected by the addition of a clause to a program is the consistency derivation rule c1. The application of c1 is affected by the addition of a clause to a program if and only if the clause defines an atom which was selected when c1 was applied.
>
> Suppose that the set of literals $\{P, Q_1, ..., Q_n\}$ was selected and the literal P was selected from the set when c1 was applied. After the application of c1, $\{P, Q_1, ..., Q_n\}$ was replaced with the set of all the resolvents each of which is a resolvent of $\{P, Q_1, ..., Q_n\}$ on P by a clause in the program. This information is recorded in R_t because $(P, \{Q_1, ..., Q_n\}) \in R_t$ If a clause of the form $P \Leftarrow Q_{n+1}, ..., Q_m$ is

[4]This set is empty if the given (incomplete) proof is not affected by the assimilation of the clause.

[5]More precise proof is in the full paper.

added to the program, the only thing which has to be done is to prove the goal $(neg, \{\{Q_1, ..., Q_m\}\})$. This is done by adding (neg, AGs) to the goal list.

Example 4.1 Consider the following program where negative literals are abducibles.

clause 1: $innocent(X) \Leftarrow not\ guilty(X)$
clause 2: $guilty(X) \Leftarrow law(L), against(X, L)$
clause 3: $law(eu)$
clause 4: $law(uk)$
clause 5: $against(a, japan)$

A proof for $innocent(a)$ is shown below.

ab 1 $?innocent(a)$

ab 2 $?not\ guilty(a)$

co 2.1 $?\{guilty(a)\}$ $(not\ guilty(a) \in \Delta)$

co 2.2 $?\{law(L), against(a, L)\}$ $((guilty(a), \emptyset) \in R)$

co 2.3 $?\{against(a, eu)\}, \{against(a, uk)\}$
$((law(L), \{against(a, L)\}) \in R)$

co 2.4 $?\{against(a, uk)\}$ $((against(a, eu), \emptyset) \in R)$

co 2.5 failure $((against(a, uk), \emptyset) \in R)$

ab 3 success

If $law(un)$ is added to the program after co 2.3 is derived, this proof becomes invalid and this invalidity is detected because $law(L)$ is recorded in R. The proof is restored by proving $\leftarrow against(a, un)$ under the updated program.

If $law(japan)$ is added to the program after co 2.3 is derived, this proof becomes invalid and this invalidity is detected because $law(L)$ is recorded in R. This proof cannot be restored because $\leftarrow against(a, japan)$ cannot be proved.

5 Experiments

In the case of depth first search, as long as the added clause is not chosen when applying the abductive derivation rule a1, it has been confirmed by experiments that the proof restoration method in Theorem 4.1 is generally faster than the naive method which restarts constructing a proof from the beginning when the program is updated. This is due to the fact that the procedure in the present paper has pruned some branches of the search tree when the naive method restarts

constructing a proof from scratch. The proof restoration method in the present paper just adds additional branches which affects the validity of proofs. The exceptional case is the case where it takes little time before a clause is added to the program. In this case, our procedure is nearly the same as the naive method as far as the time to construct a proof is concerned. One of the results of the experiments is shown below.

Example 5.1 Consider the following program in Prolog.

```
cl(hold(F,Tnext), [given+(F,T)]):-Tnext>0, T is Tnext-1.
cl(hold(F,Tnext), [hold(F,T), not(broken(F,Tnext))]):-
   Tnext>0, T is Tnext-1.
cl(broken(pos(F),Tnext), [given+(neg(F),T)]):-T is Tnext-1.
cl(broken(neg(F),Tnext), [given+(pos(F),T)]):-T is Tnext-1.
cl(given+(neg(alive),T), [do(shoot,T), hold(pos(loaded),T)]).
cl(given+(pos(loaded),T), [do(load,T)]).
cl(given+(pos(alive),0), []).
cl(given+(neg(loaded),0), []).
cl(do(load,1000), []).
cl(do(shoot,2000), []).
```

The compiler reads the above program and regards a clause of Prolog of the form:

$$\mathtt{cl(A,[L_1,...,L_n])} : -\mathtt{C_1},...,\mathtt{C_m}.$$

as the clause of an abductive logic program:

$$A \Leftarrow L_1^{km},...,L_n^{km}$$

if $\mathtt{C_1},...,\mathtt{C_m}$ hold where for each i $(1 \leq i \leq m)$, if $\mathtt{L_i}$ is of the form $\mathtt{not(F)}$, L_1^{km} is *not* F, else if $\mathtt{L_i}$ is of the form $\mathtt{F}$, L_1^{km} is F. The only abducibles are negative literals.

The program[6] expresses so called the Yale Shooting Problem. At first (at time 0), the turkey is "alive" and the gun is not "loaded". The action "load" makes the gun loaded. When the gun is "loaded", if someone "shoots" the turkey, the turkey is not "alive" afterwards. The action "load" is taken at time 1000. The action "shoot" is taken at time 2000.

given+(pos(F),T) means that F becomes true immediately after T and it continues to be true, by default, until it becomes false. *given+(neg(F),T)* means that F becomes false immediately after T and it continues to be false until it becomes true.

The test query shown below is if the turkey is "not alive" at time 4000. This query will succeed unless the program is changed.

[6]This program is made so that it takes a lot of time to construct a proof.

$? \; hold(neg(alive), 4000)$

After applying (abductive or consistency or controlling) derivation rules 10000 times, the following clause[7] is added to the abductive logic program.

$given+(neg(loaded),\, T) \Leftarrow do(unload, T)$

This does not affects the truth value of the query because "$do(unload, T)$" does not hold for any T.

It took 856.116699 cpu-time[8] for the naive method to answer the query while our method answered in 579.649963 cpu-time. While the naive method applied derivation rules 31992 times, our procedure applied derivation rules 22242 times.

6 Related Works

Truth maintenance (TM) systems have close relationships with the semantics of abductive logic programming as is surveyed in [10]. The TM system records what have been calculated and reuses them afterwards. Even if the database is updated, only limited parts of the records are corrected. There are mainly two TM systems, the justification-based TM system (JTMS) [2] and the assumption-based TM system (ATMS) [1], both of which are propositional logic. The JTMS records only one set of assumptions whereas the ATMS records different sets of assumptions at one time. Although the ATMS cannot use negation in justifications which correspond to clauses in logic programming, the JTMS can use negation corresponding to negation as failure. The relationships between the JTMS and abductive logic programming are discussed in [3, 6, 12]. The relationships between the ATMS and abductive logic programming are discussed in [18]. A proof procedure to compute generalised stable models was developed [21] using the JTMS. An abductive proof procedure for the ATMS was developed in [8]. Non-monotonic extensions of the ATMS were developed in [19, 9]. A Prolog-like theorem prover for predicate SLD resolution based on the TM systems was developed in [22, 23]. None of the above TM systems allows updates of justifications during the calculation.

A number of abductive planners [15, 17, 14, 25] use the cycle procedure [13] to cope with dynamic environments. It is straightforward to combine the cycle procedure with the procedure in the present paper. In [15, 17, 14], definitions of predicates cannot be changed. Although the treatment of undefined predicates is written in [15], the occurrences of "observation predicates" are restricted to integrity constraints. In [25], when the invalidity of a plan is detected, the plan is constructed from the beginning.

[7] This clause corresponds to "`cl(given+(neg(loaded), T), [do(unload, T)])`." in the above Prolog program. This Prolog clause can be added by using the built-in predicate "`assert`".

[8] The Prolog compiler used in this experiment is ECLiPSe.

7 Conclusions and Future Works

In Section 4, it was shown that even if some parts of the complete or incomplete proof become invalid, these invalidities can be detected and the validity can be restored by proving added goals. The extended K-M procedure and the proof restoration procedure have been tested using Prolog as is shown in Section 5. By the experiments, it was found that when the K-M procedure spends long time before a clause is added to the program, our procedure can save a lot of time in the depth first search if the added clause is not chosen when the abductive derivation rule a1 is applied.

Various planning algorithms [5, 16, 24] use abduction. As a future work, the algorithm in the current paper will be applied to planning so that it can replan efficiently.

Acknowledgements

I am grateful to Dr. Murray Shanahan for discussion.

References

[1] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[2] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[3] P. M. Dung. An abductive foundation for non-monotonic truth maintenance. In *World Conference on Fundamentals of Artificial Intelligence*, 1991.

[4] E. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In *International Conference on Logic Programming*, pages 234–254, 1989.

[5] K. Eshghi. Abductive planning with event calculus. In *International Conference and Symposium on Logic Programming*, pages 562–579, 1988.

[6] L. Giordano and A. Martelli. Generalized stable models, truth maintenance and conflict resolution. In *International Conference on Logic Programming*, pages 421–441, 1990.

[7] H. Hayashi. Abductive proofs in dynamic databases. Technical Report 744, Department of Computer Science, Queen Mary and Westfield College, University of London, 1997.

[8] K. Inoue. An abductive procedure for the CMS/ATMS. In *ECAI90 International Workshop on Truth Maintenance*, 1990.

[9] U. Junker. A correct non-monotonic ATMS. In *International Joint Conference on Artificial Intelligence*, pages 1049–1054, 1989.

[10] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 5, 1997.

[11] A. C. Kakas and P. Mancarella. Database updates through abduction. In *International Conference on Very Large Databases*, pages 650–661, 1990.

[12] A. C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Pacific Rim International Conference on Artificial Intelligence*, pages 438–443, 1990.

[13] R. Kowalski. Using meta-logic to reconcile reactive with rational agents. In *Meta-Logic and Logic Programming*, pages 227–242, 1995.

[14] R. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. Department of Computer Science, Imperial College, University of London, 1996.

[15] R. A. Kowalski and F. Sadri. An agent architecture that unifies rationality with reactivity. Department of Computer Science, Imperial College, University of London, 1997.

[16] L. R. Missiaen, M. Denecker, and M. Bruynooghe. CHICA, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, 1995.

[17] J. A. Dávila Quintero. *Agents in logic programming.* PhD thesis, Department of Computer Science, Imperial College, University of London, 1997.

[18] R. Reiter and J. deKleer. Foundations of assumption-based truth maintenance systems: preliminary report. In *AAAI87*, pages 183–188, 1987.

[19] W. L. Rodi and S. G. Pimentel. A non-monotonic ATMS using stable bases. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 485–495, 1991.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[21] K. Satoh and N. Iwayama. Computing abduction using the TMS. In *International Conference on Logic Programming*, pages 505–518, 1990.

[22] M. Shanahan. *Exploiting dependencies in search and inference mechanisms.* PhD thesis, King's College, University of Cambridge, 1987.

[23] M. Shanahan. An incremental theorem prover. In *International Joint Conference on Artificial Intelligence*, pages 987–989, 1987.

[24] M. Shanahan. Event calculus planning revisited. In *European Conference on Planning*, pages 390–402, 1997.

[25] M. Shanahan. Reinventing shakey. In *Working Notes of the AAAI Fall Symposium on Cognitive Robotics*, to appear.

Appendix

In this paper, the deletion of clauses from a program is not considered. However, the deletion of clauses can be simulated by the addition as follows. The program P is changed to $P!$ for this purpose.

Definition 7.1 *Given a set of clauses P, $P!$ is the least set of clauses such that for each clause C in P:*

$$a \Leftarrow L_1, ..., L_n$$

$P!$ contains the clause $C!$:

$$a \Leftarrow L_1, ..., L_n, not\ x$$

where x is a new atom called a **deletion atom** *which does not occur in any other clause in $P!$. $C!$ is called the* **corresponding clause** *of C.*

Similarly, the addition of a clause is as follows.

Definition 7.2 *Whenever a clause C:*

$$a \Leftarrow L_1, ..., L_n$$

where $a, L_1, ..., L_n$ mention none of the deletion atoms of clauses in $P!$, is added to the program P, the clause $C!$:

$$a \Leftarrow L_1, ..., L_n, not\ x$$

is added to $P!$ where x is a new atom called a **deletion atom** *which does not occur in any other clause in $P!$. $C!$ is called the* **corresponding clause** *of C.*

The deletion of a clause can be simulated by the addition of the deletion atom of the corresponding clause.

Definition 7.3 *Whenever a clause C is removed from the program P, the clause x is added to the program $P!$ where x is the deletion atom of the corresponding clause of C.*

Example 7.1 *Consider the following program P.*

clause 1*:* $fly(X) \Leftarrow bird(X), not\ ab(X)$
clause 2*:* $fly(penguin) \Leftarrow super(penguin)$
clause 3*:* $bird(penguin)$

$P!$ is as follows.

clause 1!*:* $fly(X) \Leftarrow bird(X), not\ ab(X), not\ del_1$
clause 2!*:* $fly(penguin) \Leftarrow super(penguin), not\ del_2$
clause 3!*:* $bird(penguin) \Leftarrow not\ del_3$

None of del_1, del_2, del_3 holds and all of not del_1, not del_2, not del_3 hold. The deletion of clause 2 *from P is simulated by the addition of the clause del_2 to $P!$.*

Making Logic Programs Reactive

James Harland
Department of Computer Science
RMIT
GPO Box 2476V
Melbourne, 3001
Australia
jah@cs.rmit.edu.au
Phone: +61 3 9925 2348
Fax: +61 3 9662 1617

Michael Winikoff
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
winikoff@cs.mu.oz.au
+61 3 9344 9100
+61 3 9348 1184

June 15, 1998

Abstract

Logic programming languages based on linear logic have been of recent interest, particularly as such languages provide a logical basis for programs which execute within a dynamic environment. Most of these languages are implemented using standard resolution or *backward-chaining* techniques. However, there are applications for which the use of *forward-chaining* techniques within a dynamic environment are appropriate, such as genetic algorithms, active databases and agent-based systems, and for which it is difficult or impossible to specify an appropriate goal in advance. In this paper we discuss the foundations for a forward-chaining approach (or in logic programming parlance, a bottom-up approach) to the execution of linear logic programs, which thus provides forward-chaining within a dynamic environment. In this way it is possible not only to execute programs in a forward-chaining manner, but also to combine forward- and backward-chaining execution. We describe and discuss the appropriate inference rules for such a system, the formal results about such rules, the role of search strategies, and applications.

1 Introduction

In recent years there has been a significant amount of interest in logic programming languages based on linear logic [7], a logic designed with bounded resources in mind. Due to the resource-sensitive nature of linear logic, such languages such as Lygon[8], Lolli[9], Forum[12], ACL[10], LC[19], and LO[3] have been applied to concurrency, updates, knowledge representation, logical interpretations of actions and graph search algorithms. These languages are generally implemented in a top-down manner,[1] using the standard techniques of resolution and unification, in that given a program P and a goal $\mathcal{G}$, the key question is to determine whether or not $P \vdash \mathcal{G}$, and hence computation proceeds by decomposing $\mathcal{G}$ into a sufficiently simple form, and then applying resolution to generate a new goal. The precise details of the computational method vary from system to system, depending on

[1] Here we use the term in the usual logic programming sense, in which top-down is synonymous with backward-chaining. However, as proof trees are written upside-down by logic programmers, and the right way up by proof theorists, such terms can be quite confusing. In this paper, we adhere to the logic programming usage, i.e. that top-down execution refers to backward-chaining techniques, and bottom-up to forward-chaining ones.

exactly what restrictions are placed on the formulae in P and $\mathcal{G}$, and what search strategy is used. However, these methods, apart from ACL, all generally use the "query-and-answer" paradigm, in that a query (the goal) is posed to the program, and if the goal succeeds, then some kind of answer information that has been calculated in the course of the proof is returned.

When compared with traditional logic programming languages such as Prolog, these languages have the advantage of introducing dynamic behaviour within a logical framework. However, there are some applications in which such *backward-chaining* techniques[2] are not as natural as *forward-chaining* ones[3]. For example, in [8] it is discussed how problems which involve rule-based changes of state, such as the blocks world, bin-packing problems, or the Yale shooting problem, fit more naturally into a paradigm in which the output of the computation is a multiset of formulae (i.e. resources) rather than an answer substitution or something similar. The most natural outcome of a bin-packing program is a representation of the final state of the bins, or for the blocks world, the most natural outcome is a representation of the final state of the world. This approach to logic programs is somewhat different from the query-and-answer paradigm, in that there is no real notion of success or failure, but rather a series of state changes which result in some particular outcome. This approach is more akin with that of ACL; in our case, we are interested in the ability to mix the paradigms together, and for the largest possible class of formulae.

The combination of forward-chaining and a dynamic environment provides a way for logic programs to be *reactive*, i.e. able to take action depending on the circumstances found at the time, rather than adhering to a predetermined plan. Applications such as genetic algorithms, active databases and agent systems appear to fit naturally into this paradigm, as they generally do not have a specific goal in mind, such as "move these blocks to that location" or "sell 10% of IBM stock", but are more open-ended, such as requiring that the movement of blocks be managed to minimise storage requirements, or that stock be bought and sold in order to maximise profits. For such applications it would seem that a bottom-up execution model would be appropriate. Clearly, it will be useful to be able to combine both bottom-up and top-down execution, which may be thought of as providing both reactivity and *rationality* within a dynamic environment, so that both unplanned and planned actions may be performed. Similar observations have been made about Mixlog [16], a system which uses aspects of linear logic to model both top-down and bottom-up execution methods in the same computational framework. However, in our case we are interested in allowing an interaction between both execution methods, rather than incorporating both into the one computational mechanism as is done in Mixlog.

Hence the key technical question is then to find the appropriate computational framework for bottom-up evaluation of linear logic programs. Unlike the situation for languages based on classical logic, in which there is no internal notion of dynamics, the linear logic case involves evaluating a program in a dynamic environment, and hence the program will generally change during computation. Hence it is not so much a matter of converting implicit information into explicit information (as, for example, happens in the well-known T_P^ω construction [5]) as changing the program itself to reflect the outcomes of the computation.

One of the key differences between the top-down and bottom-up approaches is that backtracking is used in the top-down paradigm to implement the "don't know" non-deterministic choices that need to be made. In the bottom-up case, it is not clear that such action is appropriate, but there does need to be some mechanism to cope with such choices. As we shall see, this comes down to an appropriate choice of the result of evaluating the program.

In addition, it seems important to permit top-down execution where appropriate, as is done

[2] i.e. given $A \to B$, conclude that to prove B it is sufficient to prove A

[3] i.e. given $A \to B$, conclude that if A has been derived, then B can be derived.

in Mixlog and in deductive databases systems which use bottom-up execution, such as Aditi [18]. As we shall see, our framework allows significant flexibility in the balance between top-down and bottom-up execution.

This paper is organised as follows. In Section 2 we give a brief introduction to linear logic, and in Section 3 we discuss the appropriate form of the inference rules of the system and in particular the role of the implication rule. In Section 4 we give a formal presentation of the inference rules and informal discussion of their properties. Section 5 contains a discussion of transitivity issues and Section 6 contains the formal soundness and completeness results. Section 7 presents the possibilities for normalization of derivations in our system, and the following section has a brief discussion of search strategies and applications. Finally we present our conclusions and possibilities for further work in Section 9.

2 Linear Logic

Linear logic contains two forms of conjunction: one which is "cumulative", i.e. for which $p \otimes p \not\equiv p$, and one which is not, i.e. $p \,\&\, p \equiv p$. Roughly speaking, the former is what allows linear logic to deal with resource issues, whilst the latter allows for these issues to be overlooked (or, more precisely, for an "internal" choice to be made between the resources used), as, by default, each formula in linear logic represents a resource which must be used exactly once.

Consider the following menu from a restaurant:

fruit or seafood (in season)
main course
all the chips you can eat
tea or coffee

Note that the first choice, between fruit and seafood, is a classical disjunction; we know that one or the other of these will be served, but we cannot predict which one, which may be thought of as an "external" choice, in that someone else makes the decision. On the other hand, the choice between tea and coffee is an "internal" choice — the customer is free to choose which one shall be served. Note the internal choice is a conjunction; in order to satisfy this, the restauranteur has to be able to supply *both* tea and coffee, and not just one of them. The chips course clearly involves a potentially infinite amount of resources, in that there is no limit on the amount of chips that the customer may order. We represent this situation by prefixing such formulae with a !. Note also that the meal consists of four components, and hence we connect the components with $\otimes$. Hence we have the following representation of the menu:

$$(\texttt{fruit} \oplus \texttt{seafood}) \otimes \texttt{main} \otimes \,! \,\texttt{chips} \otimes (\texttt{tea} \,\&\, \texttt{coffee})$$

where we write $\oplus$ for the classical disjunction. Note that the use of ! makes it possible to recover classical reasoning, as formulae beginning with ! with ? in a succedent behaves classically, in that such formulae may used arbitrarily many times, including 0, rather than exactly once. Hence `chips` corresponds to *exactly* one serving of chips, ! `chips` corresponds to an arbitrary number of servings (including 0). In this way we may think of a formula $!F$ in linear logic as representing an unbounded resource, i.e. one that may be used as many times as we like. Thus classical logic may be seen as a particular fragment of linear logic, in that there is a class of linear formulae which precisely matches classical formulae.

Linear logic also contains a negation, which behaves in a manner reminiscent of classical negation. The negation of a formula F is written as $F^{\perp}$. As there are two conjunctions, there are two corresponding disjunctions, as well as a dual to ! denoted as ?. The following laws, reminiscent of the de Morgan laws, all hold:

$$(F_1 \otimes F_2)^{\perp} \equiv (F_1)^{\perp} \parr (F_2)^{\perp}$$
$$(F_1 \parr F_2)^{\perp} \equiv (F_1)^{\perp} \otimes (F_2)^{\perp}$$
$$(F_1 \oplus F_2)^{\perp} \equiv (F_1)^{\perp} \,\&\, (F_2)^{\perp}$$
$$(F_1 \,\&\, F_2)^{\perp} \equiv (F_1)^{\perp} \oplus (F_2)^{\perp}$$

Each of these four connectives also has a unit, which, for $\otimes$ and $\&$ are written as $\mathbf{1}$ and $\top$, and which may be thought of as generalisations of the boolean value true, and for $\parr$ and $\oplus$ are written as $\perp$ and $\mathbf{0}$, and which may be thought of as generalisations of the boolean value false.

There is far more to linear logic than can be discussed in this paper; for a more complete introduction see the papers [7, 17, 15, 1], among others.

3 Specifying Bottom-up Computation

A bottom-up evaluation of a program involves producing a new program based on the old one. Hence, unlike the top-down case, the notion of a goal is of minimal interest per se; the main technical point is to determine how the new program is derived from the old. Hence we will denote by $P \rightsquigarrow P'$ the statement that P' can be derived in an appropriate manner from P. Naturally $\rightsquigarrow$ must respect soundness, i.e. that if $P \rightsquigarrow P'$ then $P \vdash \otimes P'$. The key point is then to find the appropriate inference system for $\rightsquigarrow$.

It should be noted that another significant difference from the top-down case is that whilst we will be a given a program P to evaluate, there will (generally) be no specification of P'. Hence the evaluation will be a forward-chaining one, and the rules for $\rightsquigarrow$ will be in the style of Plotkin's Structural Operational Semantics [13], in that given a rule such as

$$\frac{P \rightsquigarrow P''}{P \rightsquigarrow P'}\ R$$

[4] where P is known but P' is not, we will use the premise (and any appropriate sub-proofs) to evaluate P to P'', and then using the rule R we will then evaluate P to P'. Hence we will generally use the rules of the system below in such a way that P is known before the proof is attempted, and, if the proof is successful, P' is calculated in the course of the proof.

In the top-down case, the root of the proof is $\mathcal{P} \vdash \mathcal{G}$, where both $\mathcal{P}$ and $\mathcal{G}$ are known, and hence whatever calculations arise from the proof are in the form of witnesses for existentially quantified variables, or choices for disjunctive branches in the proof (and are hence dependent, at least partially, on $\mathcal{G}$). In the bottom-up case, the outcome of the computation is a multiset of formulae (P'), and this is dependent only on the original program P, and not on any other formulae. Hence the bottom-up approach encompasses a different computational paradigm that the top-down approach.

One aspect of this difference may be found in the way that "unused" resources are treated.[5] For example, we expect that $p, q, p \multimap r \rightsquigarrow q, r$; a top-down execution of the same program with

[4] Note that we assume, for simplicity, that R is a unary rule.

[5] Note that in linear logic, we do not have that $p, q \vdash p$, as the q has not been accounted for. We do, however, have that $p, q \vdash p \otimes q$.

the goal $q \otimes r$ would succeed, but both of the goals r and q would fail. Hence in the bottom-up case unused resources are not a source of failure, but are "added extras" in the outcome of the computation.

It should be noted that the above form of rules is similar to the notion of conditional rewriting systems, in that we may think of the above rule as stating that P can be re-written to P' under the condition that the premise of the rule holds.

Our technical aim, then, is to find the appropriate rules for $\rightsquigarrow$. Clearly we want the evaluation process to be complete, in that for some class of goal formulae, we have that there is a proof of a goal G from the original program just in the case that there is a proof of G from the evaluated program.

A natural starting point for this analysis is the rule of *modus ponens*. The linear form of this rule may be stated as $A, A \multimap B \vdash B$. An important point to note is that both A and $A \multimap B$ are consumed by this process, and hence the linear form has a different effect in the context of a proof than the corresponding rule in either classical or intuitionistic logic. In particular, whilst the rule looks the same, in the classical or intuitionistic case, the presence of the structural rules of weakening and contraction mean that we can actually use a stronger form of the rule, i.e. $A, A \rightarrow B \vdash A \wedge B$. Note that $A \wedge (A \rightarrow B) \equiv A \wedge B$ (in both classical and intuitionistic logics), and hence a bottom-up execution process based on this rule will preserve equivalence. This strong property ensures that such a process will never need to backtrack, as the preservation of equivalence ensures that anything provable from the original program is provable from the evaluated one. Hence, the T_P construction [5] and similar constructions which are used to give semantics to logic programs may be used without any consideration of either the order in which the rules of the program are used, or the completeness of the evaluation mechanism.

In the linear case, the equivalence property does not hold. For example, the program $p, p \multimap q$ will clearly evaluate to q, and whilst $p, p \multimap q \vdash q$, it is clearly not the case that $q \vdash p \otimes (p \multimap q)$. As a result, we need to consider how we may achieve an appropriate completeness property for the linear version.[6]

Note that the issue of completeness is complicated by the interaction between strictly linear formulae and !. For example, consider the program $p, !(p \multimap q)$. Note that $p, !(p \multimap q) \vdash p$ and $p, !(p \multimap q) \vdash q$ (but of course $p, !(p \multimap q) \not\vdash p \otimes q$). Hence we have that $p, !(p \multimap q) \vdash p \,\&\, q$, and so completeness requires that $p, !(p \multimap q) \rightsquigarrow p \,\&\, q$. Note that this is a way of integrating the choice of whether or not to make use of a particular rule with a means of making the rewriting system confluent.

Note also that we would want $!p, !(p \multimap q) \rightsquigarrow !q$, rather than the weaker form $!p, !(p \multimap q) \rightsquigarrow q$. One way to achieve this is to use the weaker form, but to have an explicit rule to achieve the stronger one where possible. We will return to this point below.

Hence the main computational metaphor will be that given a program P containing a clause $G \multimap D$, we wish to find P_1, P_2 such that $P = P_1 \cup P_2 \cup \{G \multimap D\}$ and $P_1 \vdash G$ so that we have $P_1, P_2, G \multimap D \rightsquigarrow P_2, D$. The main focus is then on the appropriate form of the implication rule. Note that we do not assume that D is atomic; as we shall see, this is a natural choice for bottom-up evaluation, as well as being very useful for applications.

It should be noted that the rules for $\rightsquigarrow$ are similar to the notion of conditional rewriting systems, in that we may think of the above rules as stating that P can be re-written to P' under the condition that the premises hold.

One key technical issue is how to determine whether $P_1 \vdash G$ or not. Clearly this will generally require right rules, such as in the program $p, q, (p \otimes q) \multimap r$, for which we require that $p, q, (p \otimes q) \multimap$

[6] As noted above, any reasonable evaluation should be sound, i.e. if $P \rightsquigarrow P'$, then $P \vdash \otimes P'$.

$r \rightsquigarrow r$. It is clear that in this case we have $P_2 = \emptyset$, $P_1 = \{p, q\}$, and it is straightforward to show that $p, q \vdash p \otimes q$. However, we would expect the same result from the program $D_1, D_2, G_1 \multimap p, G_2 \multimap q, (p \otimes q) \multimap r$ where $D_1 \vdash G_1$ and $D_2 \vdash G_2$. Note that in this case the proofs of $D_1 \vdash G_1$ and $D_2 \vdash G_2$ may be arbitrarily large.

The question is then how we write the implication rule. One possibility is the one below.

$$\frac{P_1 \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D}$$

However, this is not particularly satisfactory. This does not provide for any bottom-up evaluation of P_1, and whilst we wish to retain the possibilities for interaction between top-down and bottom-up computations, we also want to allow the possibility that our computations are as bottom-up as possible. Hence we propose the rule below:

$$\frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D} \multimap$$

When compared to the rules of the linear sequent calculus, this rule corresponds to a particularly localised form of the $\multimap$L rule, which may be specified as below:

$$\frac{P_1 \vdash G, \Delta_1 \quad P_2, D \vdash \Delta_2}{P_1, P_2, G \multimap D \vdash \Delta_1, \Delta_2} \multimap L$$

Clearly our rule corresponds to the case in which the body of the clause can be evaluated without any external context (i.e. Δ_1). As we shall see, this will require some restrictions on the proofs (and hence classes of formulae) that we consider.

Note also the flexibility of the $\multimap$ rule when it comes to the balance between top-down and bottom-up evaluation. For example, we may choose to maximally evaluate P_1 before attempting to prove $P' \vdash G$, and hence make the proof of $P' \vdash G$ as simple as possible. On the other hand, we may choose not to evaluate P_1 at all, and hence choose $P_1 = P'$, and proceed in an entirely top-down manner. Naturally, we may choose a more intermediate course, as well.

Another possibility for the implication rule is given below.

$$\frac{P \rightsquigarrow P' \quad P_1 \vdash G \quad P_2, D \rightsquigarrow P''}{P_1, P_2, G \multimap D \rightsquigarrow P''} \multimap'$$

This version of the rule encodes the transitivity of $\rightsquigarrow$ into the rule, and as we shall see, $\multimap'$ is a derived rule of the system given below.

The rules for the other connectives which may appear in programs are generally straightforward copies of those for the left rules in the linear sequent calculus. The most interesting one of these is the rule for $\parr$. The rule in the sequent calculus is

$$\frac{, _1, F_1 \vdash \Delta_1 \quad , _2, F_2 \vdash \Delta_2}{, _1, , _2, F_1 \parr F_2 \vdash \Delta_1, \Delta_2}$$

This suggests the following rule for $\rightsquigarrow$.

$$\frac{P_1, D_1 \rightsquigarrow P_1' \quad P_2, D_2 \rightsquigarrow P_2'}{P_1, P_2, D_1 \parr D_2 \rightsquigarrow (\otimes P_1') \parr (\otimes P_2')} \parr$$

Here we split the program, as indicated by the sequent calculus rule, evaluate each part separately, and then combine the results.

Another point to note is that we do not require that the heads of clauses (i.e. the D in $G \multimap D$) be atomic. In some top-down systems, this assumption is made in order to simplify the goal-reduction process, i.e. given a sequent $\mathcal{P} \vdash \mathcal{G}$, one wants to reduce $\mathcal{G}$ as much as possible, and hopefully to a multiset of atomic formulae, before applying left rules, including the resolution rule [14, 9]. There are some top-down systems in which the heads of clauses need not be atomic, such as Forum and LO, in which there may be occurrences of $\parr$, but this still corresponds to a simple method of determining the next goal. In the bottom-up case, the emphasis is more on finding connections within the program, and producing the appropriate results. This may involve the derivation of atomic formulae, but there is no obvious reason for restricting clauses to contain only atomic heads (or, for that matter, to any other subclass of definite formulae) . For example, in a genetic algorithm application, it is often desirable to replace a pair of chromosomes with a new pair; in this case, it is clearly more natural to express this change in the form $(A \otimes B) \multimap (C \otimes D)$ than in a fragment in which the right hand side of $\multimap$ is restricted to atoms.

We need also to consider some extra properties, and hence rules, for our system. For example, we would insist that the system be transitive, so that if $P_1 \rightsquigarrow P_2$ and $P_2 \rightsquigarrow P_3$, then $P_1 \rightsquigarrow P_3$. It may be possible to establish such a result by proving directly the appropriate properties of the rules, but it seems appropriate to add the necessary rule to the system, and then consider how it may be eliminated, in the manner of cut elimination [6]. In addition, we would expect that the system would allow weakening in some global sense, in that if $P_1 \rightsquigarrow P_2$, then $P_1, P_3 \rightsquigarrow P_2, P_3$. This suggests the following rules:

$$\frac{P_1 \rightsquigarrow P_2 \quad P_2 \rightsquigarrow P_3}{P_1 \rightsquigarrow P_3}\ Cut \qquad \frac{P_1 \rightsquigarrow P_2}{P_1, P_3 \rightsquigarrow P_2, P_3}\ Weak$$

Note that these rules ensure that $\rightsquigarrow$ is transitive and monotonic (essentially localising the evaluation of program clauses). Note also that Cut corresponds to the cut rule in the linear sequent calculus, and Weak to a particular form of weakening. As we shall see, these rules play an important role in the system.

There are two other rules of note. As noted above, we will require that $p, !(p \multimap q) \rightsquigarrow p \,\&\, q$. In order to achieve this, it seems reasonable to allow a "backtrackable" choice to be made at some point in the derivation, and then "collect" the different choices together when necessary. Hence if we find that $P \rightsquigarrow P_1$ and $P \rightsquigarrow P_2$, then we can conclude that $P \rightsquigarrow (\otimes P_1) \,\&\, (\otimes P_2)$. In effect this allows us to choose to execute a particular branch when desired, and to draw together various different branches as required. Another way to think of this rule is that if $P_1 \rightsquigarrow P_2$ and $P_3 \rightsquigarrow P_4$ then it seems reasonable that $(\otimes P_1) \,\&\, (\otimes P_2) \rightsquigarrow (\otimes P_3) \,\&\, (\otimes P_4)$. When $P_1 = P_2$, this is just $P_1 \rightsquigarrow (\otimes P_3) \,\&\, (\otimes P_4)$, as $(\otimes P_1) \,\&\, (\otimes P_1) \equiv (\otimes P_1)$. As noted above, it is this rule that addresses the problem of confluence in the presence of "don't know" non-determinism.

The other rule of note is the one which deals with !. As noted above, we wish to have a means of recovering the stronger conclusion where appropriate, such as determining that $!p, !(p \multimap q) \rightsquigarrow !q$ from $!p, !(p \multimap q) \rightsquigarrow q$. Hence we add an appropriate rule to the system, based on the corresponding rule in the sequent calculus, and address the eliminability of this rule below.

Hence the main technical points are concerned with implication, and "meta-properties", such as transitivity, confluence and the strength of the conclusions reached. The following section contains a formal definition of the inference system.

4 Inference Rules

Below we present the inference rules for the system.

Definition 1 *A* derivation tree *is proof tree governed by the following rules:*

$$\frac{}{P \rightsquigarrow P}\ \text{Axiom} \qquad \frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, P_2, G \multimap D \rightsquigarrow P_2, D}\ \multimap$$

$$\frac{P, D_i \rightsquigarrow P'}{P, D_1 \,\&\, D_2 \rightsquigarrow P'}\ \& \qquad \frac{P_1, D_1 \rightsquigarrow P_1' \quad P_2, D_2 \rightsquigarrow P_2'}{P_1, P_2, D_1 \parr D_2 \rightsquigarrow (\otimes P_1') \parr (\otimes P_2')}\ \parr$$

$$\frac{P, D_1, D_2 \rightsquigarrow P'}{P, D_1 \otimes D_2 \rightsquigarrow P'}\ \otimes \qquad \frac{P, D \rightsquigarrow P'}{P, !D \rightsquigarrow P'}\ ! \qquad \frac{P, !D, !D \rightsquigarrow P'}{P, !D \rightsquigarrow P'}\ C!$$

$$\frac{P \rightsquigarrow P'}{P, \mathbf{1} \rightsquigarrow P'}\ \mathbf{1} \qquad \frac{P, D[t/x] \rightsquigarrow P'}{P, \forall x D \rightsquigarrow P'}\ \forall$$

$$\frac{P_1 \rightsquigarrow P_2 \quad P_2 \rightsquigarrow P_3}{P_1 \rightsquigarrow P_3}\ Cut \qquad \frac{P \rightsquigarrow P'}{P, Q \rightsquigarrow P', Q}\ Weak$$

$$\frac{P \rightsquigarrow P_1 \quad \ldots \quad P \rightsquigarrow P_n}{P \rightsquigarrow (\otimes P_1) \,\&\, \ldots \,\&\, (\otimes P_n)}\ collect \qquad \frac{!P \rightsquigarrow P'}{!P \rightsquigarrow !P'}\ !M$$

We will often use $P \rightsquigarrow P'$ as a shorthand for the statement that $P \rightsquigarrow P'$ has a derivation tree.

Note that for the right-hand premise of the $\multimap$ rule, we assume that the standard rules of the linear sequent calculus are used.

Note also that the following rule for implication is equivalent to the one above in the presence of Weak:

$$\frac{P_1 \rightsquigarrow P' \quad P' \vdash G}{P_1, G \multimap D \rightsquigarrow D}$$

Whilst this may appear to be a simpler form of the rule, we use the above form for ease of comparison with the corresponding rule in the sequent calculus.

It is interesting to note that the $\multimap$ rule is akin to a directed cut, in that P' does not appear anywhere in the conclusion of the rule, but that there is a computational means of deriving P' from P. Hence P' corresponds to an interpolant formula for P and G [4].

Note also that as written, the rules for $\rightsquigarrow$ do not necessarily specify a bottom-up computation; the generality of the $\multimap$ rule means that the left-hand premise could be trivial. The bottom-up interpretation comes from a derivation tree together with a particular execution strategy, such as requiring that the $P \rightsquigarrow P'$ premise of $\multimap$ be maximally evaluated. Such a strategy can be specified by a syntactic rule such as P' being required to be a multiset of formulae of a particular kind, or by requiring that the proof of $P \vdash G$ have a particular property, such as bounding the number of right rules that can be used by the number of connectives in G.

Another consideration (foreshadowed above) is that the implication rule seems to require some form of restriction. In particular, it seems necessary to insist that in the $\multimap$L rule of the linear sequent calculus, i.e.

$$\frac{, _1 \vdash F_1, \Delta_1 \quad , _2, F_2 \vdash \Delta_2}{, _1, , _2, F_1 \multimap F_2 \vdash \Delta_1, \Delta_2}$$

we have that $\Delta_1 = \emptyset$. This means that we can look entirely within the program to satisfy F_1, rather than having some external context (Δ_1) make a contribution. The precise properties of this rule are then clearly of interest. Hence we come to the definition below.

Definition 2 *Let Θ be a proof in the linear sequent calculus. We say Θ is* $\multimap$-localised *if every occurrence of the rule $\multimap L$ in Θ is of the form*

$$\frac{, _1 \vdash F_1 \quad , _2, F_2 \vdash \Delta}{, _1, , _2, F_1 \multimap F_2 \vdash \Delta}$$

Our particular interest is to determine an appropriate class of formulae for which $\multimap$-localised proofs are complete (i.e. a proof exists iff a $\multimap$-localised proof exists). One way in which to obtain such a result is to restrict our attention to intuitionistic linear logic, i.e. the fragment in which every succedent contains at most one formula. However, this seems too drastic, especially as the restriction only applies to one rule, rather than to all rules. In addition, it is possible to re-write certain proofs which contain multiple formulae in succedents into $\multimap$-localised proofs. Consider the sequent $p \parr q \parr r, r \multimap s \vdash s, p, q$ which has a proof as below:

$$\dfrac{\dfrac{\dfrac{p \vdash p \quad q \vdash q}{p \parr q \vdash p, q}\parr L \quad r \vdash r}{p \parr q \parr r \vdash r, p, q}\parr L \quad s \vdash s}{p \parr q \parr r, r \multimap s \vdash s, p, q} \multimap L$$

Note that the occurrence of $\multimap$L is not $\multimap$-localised, as $\Delta_1 = \{p, q\}$, $\Delta_2 = \{s\}$. However, note that there is an $\multimap$-localised proof of this sequent, given below:

$$\dfrac{\dfrac{p \vdash p \quad q \vdash q}{p \parr q \vdash p, q}\parr L \quad \dfrac{r \vdash r \quad s \vdash s}{r, r \multimap s \vdash s} \multimap L}{p \parr q \parr r, r \multimap s \vdash p, q, s}\parr L$$

Note also that we have $p \parr q \parr r, r \multimap s \rightsquigarrow p \parr q \parr s$, as below:

$$\dfrac{p \parr q \rightsquigarrow p \parr q \quad \dfrac{r \rightsquigarrow r \quad r \vdash r}{r, r \multimap s \rightsquigarrow s} \multimap}{p \parr q \parr r, r \multimap s \rightsquigarrow p \parr q \parr s}\parr$$

Hence it is possible to show by a permutation argument (sketched below) that $\multimap$-localised proofs are complete for goal formulae in which formulae of the form $?G$ and $\bot$ are absent. The reason for this restriction is that if we allow the corresponding rules of the sequent calculus to arbitrarily add formulae to the succedent, then we cannot be guaranteed to rearrange the left-hand premise of the $\multimap$L rule to be a singleton multiset. Consider for example the following proof:

$$\dfrac{\dfrac{\dfrac{q \vdash q}{q \vdash ?p, q}W?R}{q \vdash ?p \oplus r, q}\oplus R \quad s \vdash s}{q, (?p \oplus r) \multimap s \vdash s, q} \multimap L$$

It should be clear that there can be no $\multimap$-localised proof of this sequent. The problem is that $?p$ is needed for the construction of the formula in the endsequent, and that in order for this to happen, we require the presence of q in the succedent of the left premise of $\multimap$L. Note that when considering proof-search which begins at the root of the proof, the W?R and $\bot$R rules may be

thought of as reducing the number of formulae in the succedent. However, if we do not allow such rules, then the only way in which a succedent which contains more than one formula can be reduced to the succedent of an axiom (which must be a singleton) is for each occurrence of $\parr$R to be matched by an occurrence of $\parr$L which reduces the size of the succedent.

To see an example of this, consider the provable sequent $p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t$ which has the following proof (which is not $\multimap$-localised):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{q \vdash q \quad r \vdash r}{r, q \vdash q \otimes r}\otimes R \qquad t \vdash t}{r, t \parr q \vdash q \otimes r, t}\parr L \qquad p \vdash p}{p \parr r, t \parr q \vdash p, q \otimes r, t}\parr L}{p \parr r, t \parr q \vdash p \parr (q \otimes r), t}\parr R \qquad s \vdash s}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s, t}\multimap L}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t}\parr R$$

Note that the upper occurrence of $\parr$L can be permuted downwards past the $\multimap$L rule, whereas the lower occurrence of $\parr$L cannot be permuted past the occurrence of $\parr$R, as the formulae p and $q \otimes r$ are on different branches of the occurrence of $\parr$L. Hence we can rearrange this proof into the one below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p \vdash p \quad \dfrac{r \vdash r \quad q \vdash q}{r, q \vdash q \otimes r}\otimes R}{p \parr r, q \vdash p, q \otimes r}\parr L}{p \parr r, q \vdash p \parr (q \otimes r)}\parr R \qquad s \vdash s}{p \parr r, q, (p \parr (q \otimes r)) \multimap s \vdash s}\multimap L \qquad t \vdash t}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s, t}\parr L}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \vdash s \parr t}\parr R$$

Note that this sequent has the following derivation in our system:

$$\dfrac{\dfrac{\dfrac{p \rightsquigarrow p \quad q, r \rightsquigarrow q, r}{p \parr r, q \rightsquigarrow p \parr (q \otimes r)}\parr \qquad p \parr (q \otimes r) \vdash p \parr (q \otimes r)}{p \parr r, q, (p \parr (q \otimes r)) \multimap s \rightsquigarrow s}\multimap \qquad t \rightsquigarrow t}{p \parr r, t \parr q, (p \parr (q \otimes r)) \multimap s \rightsquigarrow s \parr t}\parr$$

A formal statement of the correctness of this process is in Section 6.

5 Transitivity Rules

As mentioned above, we want $\rightsquigarrow$ to be transitive and monotonic. However, it is not clear that Cut and Weak are the only rules which will serve this purpose (although it should be clear that they are reasonable choices). In addition, we would also expect some slightly stronger properties of $\rightsquigarrow$ to hold, in that we would expect both of the following rules to be admissible:

$$\dfrac{P_1 \rightsquigarrow P_1' \quad P_2 \rightsquigarrow P_2'}{P_1, P_2 \rightsquigarrow P_1', P_2'}\ \text{Mix} \qquad \dfrac{P_1 \rightsquigarrow P_2 \quad P_2, P \rightsquigarrow P_3}{P_1, P \rightsquigarrow P_3}\ \text{ICut}$$

Note that there is a counterpart to Mix in the classical sequent calculus, which is often used to simplify proofs, such as cut elimination. Note also that ICut is similar to the version of the cut rule used in the intuitionistic sequent calculus LJ.

It is easy to see that Cut is an instance of ICut, and that Weak is an instance of Mix (as $P \rightsquigarrow P$). Hence it is clear that the combination of Mix and ICut is at least as powerful as that of Cut and Weak. Interestingly, the converse is also the case, in that Mix and ICut are derived rules in the presence of Cut and Weak as below:

$$\frac{P_1 \rightsquigarrow P_1' \quad P_2 \rightsquigarrow P_2'}{P_1, P_2 \rightsquigarrow P_1', P_2'}\ \text{Mix} \qquad \frac{\dfrac{P_1 \rightsquigarrow P_1'}{P_1, P_2 \rightsquigarrow P_1', P_2}\ Weak \quad \dfrac{P_2 \rightsquigarrow P_2'}{P_1', P_2 \rightsquigarrow P_1', P_2'}\ Weak}{P_1, P_2 \rightsquigarrow P_1', P_2'}\ Cut$$

$$\frac{P_1 \rightsquigarrow P_2 \quad P_2, P_3 \rightsquigarrow P_4}{P_1, P_3 \rightsquigarrow P_4}\ \text{ICut} \qquad \frac{\dfrac{P_1 \rightsquigarrow P_2}{P_1, P_3 \rightsquigarrow P_2, P_3}\ Weak \quad P_2, P_3 \rightsquigarrow P_4}{P_1, P_3 \rightsquigarrow P_4}\ Cut$$

Hence we get the result below.

Proposition 1 *Cut + Weak ≡ Mix + ICut*

Hence the combination of Cut and Weak seems to be a good way to get simple, elegant rules with the appropriate power. We would want both of these to be ultimately eliminable, however, in order to simplify the process of finding proofs. However, note that due to the forward-chaining (or "rewriting") nature of the rules, any lack of eliminability is not as drastic as it may seem. For example, with the Cut rule, given P_1, it is possible to calculate P_2 via the left premise, and once P_2 is known, it is possible to calculate P_3 from the right premise. Hence, whilst eliminability is desirable, a lack of it does not necessarily mean that the system is inherently intractable.

6 Results

In this section we give the formal results about $\rightsquigarrow$.

In what follows, we assume that implications are not allowed in goals. This is for reasons of simplicity. For example, if at some stage in the derivation there was an occurrence of the implication rule of the form

$$\frac{P_1 \rightsquigarrow P' \quad P' \vdash D' \multimap G}{P_1, (D' \multimap G) \multimap D \rightsquigarrow D}$$

then the proof of $P' \vdash \multimap G$ could conclude with $\multimap$R [7], making the premise $P', D \vdash G$. Clearly this sequent is not problematic per se, but in a completeness proof based on an induction on the size of the proof, this means that we can "push" the bottom-up behaviour into the right-hand premise of the $\multimap$ rule as well, in that as the proof of $P', D \vdash G$ is necessarily shorter than that of the endsequent, we can then deduce that $\exists P''$ such that $P', D \rightsquigarrow P''$ and $P'' \vdash G$. Note that whatever the hypothesis implies about $P', D \vdash G$ will also be true for $P'' \vdash G$, and so we can recursively apply the hypothesis all the way up the proof tree. This suggests the addition of the following rule to the inference system:

$$\frac{P \rightsquigarrow P' \quad P' \vdash \mathcal{G}}{P \vdash \mathcal{G}}$$

[7] In fact, as the $\multimap$R rule is asynchronous (in the terminology of [2]) then the proof may be assumed to be of this form.

Note that in this form, this rule is precisely a directed cut. This may be thought of as a "top-down" conclusion from the premises $P \rightsquigarrow P'$ and $P' \vdash \mathcal{G}$, whereas the implication rule gives $P, G \multimap D \rightsquigarrow D$, which may be considered the corresponding "bottom-up" conclusion. However, a full analysis and discussion of this point would take us beyond the scope of this paper, and so for now we omit the possibility that goals may contain implications to simplify the discussion.

Hence we arrive at the class of formulae below.

Definition 3 *We define the following classes of formulae:*

Goal *formulae:* $G ::= A \mid G \otimes G \mid G \parr G \mid G \,\&\, G \mid G \oplus G \mid \mathbf{1} \mid \bot \mid \forall x G \mid \exists x G \mid !G \mid ?G$

Strong goal *formulae:* $G ::= A \mid G \otimes G \mid G \parr G \mid G \,\&\, G \mid G \oplus G \mid \mathbf{1} \mid \forall x G \mid \exists x G \mid !G$

Definite *formulae:* $D ::= A \mid D \otimes D \mid D \parr D \mid D \,\&\, D \mid \mathbf{1} \mid \bot \mid \forall x D \mid !D \mid G \multimap D$

where G is a goal formula.

Strong definite *formulae are definite formulae in which all formulae of the form $G \multimap D$ are such that G is a strong goal formula.*

In all cases above, A is an atomic formula.

Definition 4 *We define the* depth *of a formula F as follows:*

depth(A) = 0 where A is atomic

depth($\clubsuit F$) = 1 + depth(F) where $\clubsuit$ is a unary connective

depth($F_1 \heartsuit F_2$) = 1 + max(depth(F_1), depth(F_2)) where $\heartsuit$ is a binary connective

We define depth($\mathcal{F}$) = $\Sigma_{F \in \mathcal{F}} \mathrm{depth}(F)$.

Proposition 2 (Soundness) *Let P be a multiset of definite formulae. Then if $P \rightsquigarrow P'$ then $P \vdash \otimes P'$.*

Corollary 1 *Let P be a multiset of definite formulae and let $\mathcal{G}$ be a multiset of goal formulae. Then if $P \rightsquigarrow P'$ and $P' \vdash \mathcal{G}$ then $P \vdash \mathcal{G}$.*

Proof: By the above proposition, $P \vdash \otimes P'$, and as $\otimes P' \vdash \mathcal{G}$, we have that $P \vdash \mathcal{G}$. □

The statement of a completeness result is more subtle. It is tempting to state the completeness of $\rightsquigarrow$ as follows: if $P \vdash \mathcal{G}$ then $\exists P'$ such that $P \rightsquigarrow P'$ and $P' \vdash G$. However, this is trivially true, as $P \rightsquigarrow P$. Hence the statement of completeness need to be somewhat more careful, and will depend upon the precise execution properties desired. Below we present one version of completeness which seems appropriate for a "maximally bottom-up" computation.

and

Proposition 3 (Definite Completeness) *Let P be a multiset of strong definite formulae and $\mathcal{G}$ be a multiset of strong goal formulae. If $P \vdash \mathcal{G}$ has an $\multimap$-localised proof, then there is a multiset P' of D formulae such that $P \rightsquigarrow P'$ and $P' \vdash \mathcal{G}$ where the number of right rules in the proof of $P' \vdash \mathcal{G}$ is not more than depth($\mathcal{G}$).*

It is clear that the amount of mandatory bottom-up evaluation is proportional to the degree of freedom allowed in the proof of $P' \vdash \mathcal{G}$; the greater the freedom, the more scope there is for

top-down evaluation. Hence strategies which are more intermediate than the strict bottom-up approach will require an appropriate statement of the requirements for this proof.

Note that the restriction to strong definite formula is necessary, as the presence of contraction on the right may violate the property about right rules and the depth of the succedent. Whilst this may seem a little strong, the proposition below introduces another reason for restricting our attention to this class of formula, as we require that weakening not be present in order to show the completeness of $\multimap$-localised proofs.

Proposition 4 *Let P be a multiset of strong definite formulae and $\mathcal{G}$ be a multiset of strong goal formulae. Then $P \vdash \mathcal{G}$ is provable iff $P \vdash \mathcal{G}$ has an $\multimap$-localised proof.*

Hence we have that $\rightsquigarrow$ is complete for strong definite formulae. As noted above, there may be other forms of completeness which are of interest, but the one given above seems appropriate for bottom-up execution.

Note that in order to determine search strategies for our inference rules, we consider their permutation properties, i.e. the ability to interchange adjacent rules without altering the overall proof. Many of these follow immediately from the properties of the corresponding rules in the linear sequent calculus [11]. Our interest is in the properties of the rules peculiar to our system, and in particular the consequences of the permutation properties for the eliminability of Cut and Weak (and for that matter, collect and !M) and the possibilities for normal forms for proofs.

A full analysis is beyond the scope of this paper, but we briefly discuss these below. Our Cut rule generally cannot be permuted upwards, but can always be permuted downwards, expect past other occurrences of Cut. Hence the Cut rule is not eliminable, but we can limit its occurrences to being closer to the endsequent than any other rule. As mentioned above, this does not mean that our rules are inherently intractable, as it is possible to use P to calculate P'. In fact, this may be considered as an appropriate proof-theoretic characterization of the transitivity of $\rightsquigarrow$, in that a derivation of $P_1 \rightsquigarrow P_2$ can always be "supplemented" by the derivation of $P_2 \rightsquigarrow P_3$ and hence $P_1 \rightsquigarrow P_3$, no matter how long or complicated the derivation of $P_1 \rightsquigarrow P_2$ may be.

The Weak rule also cannot be generally eliminated. It is eliminable at the axioms, and permutes upwards past all rules except $\parr$, collect and !M. Hence this suggests that the three rules $\parr$, collect and !M should be modified to incorporate the Weak rule. In such a modified system, the Weak rule is eliminable (note also that our form of the $\multimap$ rule effectively already has Weak encoded into it). Note also that Weak is a more general rule than the corresponding one in the linear sequent calculus, as the Weak rule does not place any restrictions on the formulae which may occur in the rule. In this sense, and particularly given the impermutabilities of the Weak rule, our inference system has some characteristics of *affine logic*, i.e. linear logic with arbitrary weakening rules added. It is interesting to note that in affine logic, the weakening rules cannot always be permuted upwards, akin to the lack of upward permutability of Weak past !M. In a computational sense, this indicates when it is necessary to "localise" the current search.

The collect rule has the interesting property of permuting downwards past all rules except Weak, and on the right-hand side of Cut. Hence in a system in which Weak is eliminable, collect can always be permuted downwards, except past certain occurrences of Cut. This suggests that a normal form would position the occurrences of collect immediately above the occurrences of Cut.

It should also be noted that the standard problems of contraction apply here, and hence so can the standard solutions, such as requiring proofs not to contain occurrences of weakening immediately above contraction where the two rules do not occur in permutation position, i.e. that there is a formula introduced by the weakening rule which is eliminated by the contraction.

We also note that the permutation behaviour of Mix and ICut is significantly more restrictive than that of Cut and Weak, which acts as a post-fact justification of our choice of the latter two rules.

Using these observations, it is not hard to come up with an equivalent "normalized" system, in which the above suggestions are incorporated, and in which:

- there are no occurrences of Weak.
- for each occurrence of Cut, the only rules between the Cut and the endsequent are other occurrences of Cut.
- for each occurrence of collect, the only rules between the collect and the endsequent are other occurrences of collect or Cut.

7 Computational Issues

As noted above, the normal form for proofs consists of an arbitrary number of occurrences of the Cut rule at the root of the proof. From a computational point of view, this means that we may arbitrarily extend any derivation of $P_1 \rightsquigarrow P_2$ to $P_1 \rightsquigarrow P_3$ provided that we can show that $P_2 \rightsquigarrow P_3$, and so that computation can continue whenever the appropriate reductions can be found. This indefiniteness of termination is clearly appropriate for a forward-chaining application such as a genetic algorithm, in which it is desirable to continue computation until a certain threshold is reached.

The properties of the collect rule pose some interesting implementation issues. Clearly if it is desired to determine all possible consequences of a particular set of circumstances (such as finding all possible states which would result from a particular setting of valves in a power plant), then it will be necessary to have many occurrences of the collect rule, and hence a potentially large number of different computational branches. On the other hand, if the aim of the computation is to find some appropriate final state (as distinct from all such possible states), such as an appropriate serialisation of a set of banking transactions, then it may not be appropriate to use the collect rule at all, as the entire set of possibilities does not need to be examined. Hence it would seem natural to allow the user to explore the different branches lazily (i.e. each branch is generated on demand) rather than eagerly (i.e. all branches are evaluated in advance).

The issues concerning the other rules are fairly standard for the implementation of linear logic programming languages, and hence will not be discussed here. One exception to this is the !M rule, which, in this case, will function as a trigger of its own: whenever the formulae on the left of $\rightsquigarrow$ contain only formulae of the form $!D$, then "promote" the current outcome. As computation will proceed by having P specified in advance, it is straightforward to determine when this promotion should occur. For example, given the program $!D, !(G' \multimap D')$, it should be clear that for any P' such that $!D, !(G' \multimap D') \rightsquigarrow P'$, then $!D, !(G' \multimap D') \rightsquigarrow !P'$. Hence the !M rule could be implemented as a flag of some sort, rather than as an explicit search rule per se.

There are some remaining sources of non-determinism, including the choice of the formula to be used in rules such as $\multimap$. An interesting observation that can be made is that there are some cases in which a "breadth-first" approach would be desired. For example, in a genetic algorithm, the use of aggregate functions (such as sum, count, maximum, minimum, etc.) implies that the use of atoms must be breadth-first within a predicate; for example, the fitness of the overall is generally required, and may be used as an input to the next step in the algorithm. Clearly then the data has to be processed in a breadth-first manner to achieve this. On the other hand, in an application such

as an active database, a depth-first approach may be more appropriate, as it is generally important to follow through on a particular course of action rather than allowing alternatives to be pursued.

8 Acknowledgements

The authors are indebted to David Pym for many stimulating discussions about this paper.

References

[1] V. Alexiev. Applications of Linear Logic to Computation: An Overview. *Bulletin of the IGPL* 2(1):77-107, 1994.

[2] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. Logic Computat.* 2(3), 1992.

[3] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. Proceedings of the International Conference on Logic Programming, 496-510, Jerusalem, June, 1990.

[4] P. Andrews. *An Introduction to Mathematical Logic and Type Theory.* Academic Press, 1986.

[5] M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the Association for Computing Machinery* 23:4:733-742, October, 1976.

[6] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift* 39, 176-210, 405-431, 1934.

[7] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50, 1-102, 1987.

[8] J. Harland, D. Pym, and M. Winikoff. Programming in Lygon: An Overview. Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology 391-405, Munich, July, 1996.

[9] J. Hodas, D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic: Extended Abstract. Proceedings of the Symposium on Logic in Computer Science, 32-42, Amsterdam, July, 1991.

[10] N. Kobayashi, A. Yonezawa. ACL - A Concurrent Linear Logic Programming Paradigm. Proceedings of the International Logic Programming Symposium, 279-294, Vancouver, October, 1993.

[11] P.D. Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Stanford University, August 1992.

[12] D. Miller. A multiple-conclusion meta-logic. Proc. of the IEEE Symposium on *Logic in Computer Science* 272–281, Paris, June, 1994.

[13] G. Plotkin. Structural Operational Semantics (lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981 (reprinted 1991).

[14] D.J. Pym, J.A. Harland. A Uniform Proof-theoretic Investigation of Linear Logic Programming. Journal of Logic and Computation 4:2:175-207, April, 1994.

[15] A. Ščedrov. A Brief Guide to Linear Logic. in *Current Trends in Theoretical Computer Science.* G. Rozenberg and A. Salolmaa (eds.), World Scientific, 1993.

[16] D. Smith. Mixlog: A Generalised Rule-based Language. Manuscript, 1997.

[17] A. Troelstra. Lectures on Linear Logic. CSLI Lecture Notes No. 29, 1992.

[18] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland. The Aditi Deductive Database System. *VLDB Journal* 3:2:245-288, April, 1994.

[19] P. Volpe. Concurrent Logic Programming as Uniform Linear Proofs. In G. Levi and M. Rodríguez-Artalejo (eds.), *Algebraic and Logic Programming*, 133–149. Springer, 1994.

Maintaining and Restoring Database Consistency with Update Rules

Sofian Maabout[††]
University of Antwerp (UIA)
maabout@uia.ac.be

Abstract

Maintaining database consistency is one of the major applications of active databases. One of the problems encountered in this case is the non termination of rules execution. Then, the general solution is to undo the modifications and roll back to the initial database which is *assumed* to be consistent. In this paper, we present an algorithm that generates update rule programs from constraints expressions. The semantics of these rules is given by means of a logic program model, namely the well founded model. The derived programs can serve to maintain and restore database consistency. Maintaining database consistency assumes its consistency before user updates are requested while restoring consistency does not consider this hypothesis.

1 Introduction

Active databases use ECA rules (Event, Condition, Action) in order to specify which actions must be triggered when some event and condition are met. Database integrity management is one of the most prominent application fields of active databases. The intuitive idea is that whenever some violating updates arise, if integrity constraints are violated, then the system must achieve some correcting actions in order to restore database consistency. A large amount of research has been done in the integration of active rules in this area, see e.g. [CFPT94, CW90, LML97, ST96, UD92]. We also refer to [AART97, DDS98, ALS95, MT98, TO95] among works that have considered logic programming paradigm in the context of integrity constraints management.

Ceri & Widom[CW90] consider integrity constraints (IC) expressed as SQL queries with possibly aggregates. The derived rules from these IC guarantee database consistency whenever rules execution terminates. Schewe & Thalheim [ST96] characterize a class of IC for which there exists a program of active rules that restores database consistency without invalidating users updates when it is possible, e.g. let $p(a) \Rightarrow q(a)$ be an IC and let $\Delta = \emptyset$ be a database instance where we want to insert $p(a)$. This transition violates the IC. In order to restore consistency either $q(a)$ is inserted or $p(a)$ is deleted. The first alternative preserves the effect of the user transition, thus it is *preferred*. Inserting $p(a)$ is in this spirit, a *repairable* transition because there exists a consistent database state containing $p(a)$. Notice that the transaction

[††]Author's address: University of Antwerp (UIA). Departement Wiskund en Informatica. Universiteitplein 1. B-2610 Antwerpen (Wilrijk). Belgium

$+p(a); \neg q(a)$ is not repairable. [ST96] does not address the problem of termination. Ludäscher *et al* [LML97, LM98] consider the particular case of referential IC's as those proposed in SQL92 standard. They give a translation of these IC's into logic programs and show that the well founded semantics [vGRS91] may be used in order to maintain database consistency. They use *Statelog* programs which are Datalog programs where predicates are augmented with a state attribute that may contain the successor function +1. Because of this use of functions, the programs in *Statelog* must deal with infinite models. Among the works on the termination of rules execution, we refer to [AWH92, BCP95] and the recent work [BDR98]

In this paper, we propose an algorithm that derives logic rules from declarative expression of IC's. Although we use well founded semantics _which is based on a three valued logic_ we show that the models we obtain are always bi-valued. This paper is an extension of [BM97] where we have considered incomplete databases.

Motivating example. Consider the following IC:

$$c_1 \equiv \forall X : \ p(X) \Rightarrow q(X)$$
$$c_2 \equiv \forall X : \ r(X) \Rightarrow \neg q(X)$$

and assume that we have the following update rules

$$\rho_1 : \ p(X), \neg q(X) \longrightarrow +q(X)$$
$$\rho_2 : \ r(X), q(X) \longrightarrow -q(X)$$

ρ_1 maintains c_1 and the second c_2. If $p(a)$ and $p(b)$ are both inserted in the empty (coherent) database, then the rules are fired in order to restore consistency. ρ_1 is executed first then ρ_2 becomes executable and after executing it, ρ_1 becomes again executable. Thus, we enter a non terminating rule execution. The generally adopted solution in this case, is to stop rules execution and roll back to the initial state of the database, and since this state is a coherent one, so we have a consistent final state.

In this paper, we try to avoid the consistency of the initial state hypothesis. Doing so, we must find a way to associate active rules to IC's whose execution never enter non terminating loops.

Paper organization. The first section introduces the concepts used throughout the paper. Then, we propose an algorithm that generates a set of update rules from a set of IC's. We show that these programs can maintain database consistency. Then, we refine the algorithm in order to obtain rules that can maintain **and** restore database consistency.

The ability to restore database consistency is quite useful in a context where IC's can be added during the life of the database.

2 Preliminaries

A database Δ is a set of facts. The constraints we consider in this paper are those that can be expressed by denials. A *denial* is a rule of the form

$$p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_n(\tilde{Y}_n) \longrightarrow \texttt{inc} \qquad (1)$$

This rule must be *safe* in a sense that variables appearing in negative literals must also appear in a positive literal. `inc` is a 0-ary predicate, hence a denial can be seen as a boolean query. If $\mathcal{C}$ is a set of such IC's, then a database Δ satisfies $\mathcal{C}$ iff $\Delta \cup \mathcal{C} \not\models \texttt{inc}$ or equivalently, $\Delta \cup \mathcal{C} \models \neg\texttt{inc}$.

Now suppose that there exists $c \in \mathcal{C}$ such that $\Delta \models body(\mathcal{C})$. This means that Δ is inconsistent. To restore consistency, either we insert some tuples in the predicates q_j or we delete some ones from the predicates p_i. Thus, from the denial (1), we may derive one rule among the $m+n$ rules of the form

$$p_1(\tilde{X}_1), \ldots, p_{i-1}(\tilde{X}_{i-1}), p_{i+1}(\tilde{X}_{i+1}), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_n(\tilde{Y}_n) \longrightarrow -p_i(\tilde{X}_i)$$
$$\text{or}$$
$$p_1(\tilde{X}_1), \ldots, p_m(\tilde{X}_m), \neg q_1(\tilde{Y}_1), \ldots, \neg q_{j-1}(\tilde{Y}_{j-1}), \neg p_{j+1}(\tilde{X}_{j+1}), \ldots \neg q_n(\tilde{Y}_n) \longrightarrow +q_j(\tilde{Y}_j)$$

Example 1 Let us consider the referential IC:

$$employee(X, Y), \neg dept(Y) \longrightarrow \texttt{inc}$$

From this denial, we can derive two update rules

$$employee(X, Y) \longrightarrow +dept(Y) \text{or} \neg dept(Y) \longrightarrow -employee(X, Y)$$

Notice that the second rule is not safe nor range restricted; Y and X do not appear in any positive literal in the body of the rule. But this is not a problem since we consider active domain semantics [AHV95]. The problem of domain independence of update rules has been addressed in [BM97]. □

The example above shows how we can derive update rules from denials. We don't give this straightforward algorithm here. The interested reader may find the details in [BM97] where we considered a more general class of IC's.

Now, we recall some basic definitions we use throughout the paper. We consider a set P of normal (base) predicates and SP the set of signed or update predicates such that $p \in P \Rightarrow \{+p, -p\} \subseteq SP$. The definitions of facts, atoms and literals are the classical ones [CGT90]. $Lit(P)$ (resp. $Lit(SP)$) is the set of literals obtained by using the predicates in P (resp. SP). An update rule ρ is a Datalog rule where $head(\rho)$ is an atom in $Lit(SP)$ and $body(\rho) \cap Lit(SP) = \emptyset$. An update program is a set of update rules. A partial interpretation I is a set of ground literals such that, if $pos(I)$ and $neg(I)$ are respectively the set of positive and negative literals of I, and if $\neg.neg(I) = \{p : \neg p \in neg(I)\}$, then $pos(I) \cap \neg.neg(I) = \emptyset$. Let $\mathcal{A}$ be a set of atoms, then I is *total* with respect to $\mathcal{A}$ if for all $p \in \mathcal{A}$, $\{p, \neg p\} \cap I \neq \emptyset$. I is conflicting if there exists p such that $\{+p, -p\} \subseteq I$.

Definition 1 (Entailment) I entails a ground positive literal p (denoted $I \models p$) if $I \supseteq \{p, \neg -p\}$ or $I \supseteq \{+p\}$. Conversely, $I \models \neg p$ iff $I \supseteq \{\neg p, \neg +p\}$ or $I \supseteq \{-p\}$. I satisfies a ground rule ρ ($I \models \rho$) iff whenever $I \models body(\rho)$, then $Head(\rho) \in I$. $Inst(\mathcal{P})$ is the set of all ground rules obtained from the program $\mathcal{P}$ by replacing all variables by the constants appearing in $\mathcal{P}$. I is a model of $\mathcal{P}$ if I satisfies every rule in $Inst(\mathcal{P})$.

□

Definition 2 (Update) An update δ is a set of ground update literals. A **user update** is a non conflicting set of ground update atoms.

□

Definition 3 (Operator $\mathcal{T}$) Let I be an interpretation and $\mathcal{P}$ an update program.

- $T^{\in}_{\mathcal{P}}(I) = \{head(\rho) \ : \ \rho \in inst(\mathcal{P}) \text{ and } I \models body(\rho)\}$.
- $NF_{\mathcal{P}}(I) = \{\ell \ : \ \forall \rho \in inst(\mathcal{P}), head(\rho) = \ell \Rightarrow I \models \neg body(\rho)\}$.

The operator $\mathcal{T}$ is defined by the composition of these two operators:

$$\mathcal{T}_{\mathcal{P}}(I) = I \cup \mathcal{T}^{\in}_{\mathcal{P}}(I) \cup \neg.NF_{\mathcal{P}}(I)$$

□

It is easy to see that $\mathcal{T}$ is monotone, so the sequence $(\mathcal{T}^n_{\mathcal{P}})_{n\geq 0}$ defined by

$$\begin{array}{rcl} \mathcal{T}^0_{\mathcal{P}}(I) & = & I \\ \mathcal{T}^n_{\mathcal{P}}(I) & = & \mathcal{T}^{n-1}_{\mathcal{P}}(I) \end{array}$$

admits a limit. This limit is called *the update model of* $\mathcal{P}$ and is denoted by $\mathcal{M}_I(\mathcal{P})$ or simply by $\mathcal{M}(\mathcal{P})$ when I is understood.

Intuitively, the update process can be described as follows: given a user update δ, an initial database Δ, and an update program $\mathcal{P}$:

1. The application of δ gives a new state (set of facts) Δ_{user},
2. compute the update model $\mathcal{M}_{\Delta_{user}}(\mathcal{P})$. This contains a set of update literals which we call "derived updates", then
3. the application of the derived updates to Δ_{user} gives a new database Δ_{final}.

Formally, consider Δ, a user update δ and a program $\mathcal{P}$.

- $\Delta_{user} = \Delta \cup \{p \mid +p \in \delta\} - \{p \mid -p \in \delta\}$.

- Let $\mathcal{H}$ be the herbrand base of $\mathcal{P}$. Then, $\texttt{Apply}(\delta, \Delta, \mathcal{P}) = \Delta_{final}$ where
  - if $\mathcal{M}_{\Delta_{user}}(\mathcal{P})$ is total wrt $\mathcal{H}$ and not conflicting then $\Delta_{final} = \{\Delta_{user} \cup \{p \mid +p \in \mathcal{M}_{\Delta_{user}}(\mathcal{P})\} \setminus \{p \mid -p \in \mathcal{M}_{\Delta_{user}}(\mathcal{P})\}$,
  - otherwise $\Delta_{final} = \Delta$.

Example 2 Let $\Delta = \{r, s\}, \delta = \{+p, +q\}$ and $\mathcal{P}$ be the program

$$\begin{array}{rcl} p, r & \longrightarrow & -s \\ q, s & \longrightarrow & -r \end{array}$$

$\Delta_{user} = \{p, q, r, s\}$ and it is embedded into $\mathcal{P}$ by adding the rules

$$\begin{array}{rcl@{\qquad\qquad}rcl} \longrightarrow & p; & & \longrightarrow & q \\ \longrightarrow & r; & & \longrightarrow & s \end{array}$$

Let's call the new program by $\mathcal{P}_{user}$. Now, let us compute the update model

$$\begin{array}{rcl} T^{\in}_{\mathcal{P}_{user}}(\emptyset) & = & \{p, q, r, s\} \\ NF_{\mathcal{P}_{user}}(\emptyset) & = & \{\neg\text{–}p, \neg\text{–}q, \neg\text{–}r, \neg\text{–}s, \neg\text{+}p, \neg\text{+}q, \neg\text{+}r, \neg\text{+}s\} \\ I_1 & = & T^{\in}_{\mathcal{P}_{user}}(\emptyset) \bigcup NF_{\mathcal{P}_{user}}(\emptyset) \end{array}$$

$$\begin{array}{rcl} T^{\in}_{\mathcal{P}_{user}}(I_1) & = & \{p, q, r, s\} \\ NF_{\mathcal{P}_{user}}(I_1) & = & \{\neg\text{–}p, \neg\text{–}q, \neg\text{–}r, \neg\text{–}s, \neg\text{+}p, \neg\text{+}q, \neg\text{+}r, \neg\text{+}s\} \\ I_2 & = & I_1 \end{array}$$

I_2 is the fixpoint thus it is the update model. It is not total because both $\text{–}s$ and $\text{–}r$ are "unknown". So $\Delta_{final} = \Delta$. One should notice that for example, even if $\text{+}p$ does not appear in $\mathcal{P}_{user}$, it is considered as being in the herbrand base. indeed, as soon as p, $\text{+}p$ or $\text{–}p$ appears in $\mathcal{P}_{user}$, $\{p, \text{+}p, \text{–}p\}$ is in the "extended herbrand base" of $\mathcal{P}_{user}$.

□

The following proposition characterizes the cases where the derived rules from IC's can maintain database consistency.

Proposition 1 Let $\mathcal{C}$ be a set of constraints and $\mathcal{P}_{\mathcal{C}}$ its associated update program. Then, for each user update δ and initial database Δ, if $\mathcal{M}$ is total and not conflicting then $\texttt{Apply}(\delta, \Delta, \mathcal{P}_{\mathcal{C}})$ is consistent.

□

From the definition of `Apply`, if $\mathcal{M}$ is not total or is conflicting then $\texttt{Apply}(\delta, \Delta, \mathcal{P}_{\mathcal{C}}) = \Delta$. Thus, if Δ is not initially consistent, we end up with a non consistent database. So, $\mathcal{P}_C$ can maintain database consistency but in some cases, it cannot restore it.

The fact that the update model of $\mathcal{P}$ can be not total is due to the presence of negative cycles in the dependency graph of the program $\bar{\mathcal{P}}$ defined below.

Definition 4 Let $\mathcal{P}$ be an update program. $\bar{\mathcal{P}}$ is the program obtained from $\mathcal{P}$ by:

1. if $head(\rho) = \text{+}p(X)$ then add $\neg\text{–}p(X)$ into $body(\rho)$,
2. if $head(\rho) = \text{–}p(X)$ then add $\neg\text{+}p(X)$ into $body(\rho)$,
3. replace each $p(X) \in body(\rho)$ by $(p(X) \wedge \neg\text{–}p(X)) \vee \text{+}p(X)$, and
4. replace each $\neg p(X) \in body(\rho)$ by $(\neg p(X) \wedge \neg\text{+}p(X)) \vee \text{–}p(X)$. □

Lemma 1 If $\bar{\mathcal{P}}$ is stratified, then $\forall \delta, \Delta : \mathcal{M}_{\Delta_{user}}(\mathcal{P})$ is total and not conflicting.

□

In fact, we can distinguish between two kinds of negative cycles: negatively recursive programs (NR) and potentially conflicting (PC) programs. NR programs are those containing negative cycles after transformations 3. and 4. in the definition above. PC programs are those containing negative cycles after transformations 1. and 2. These are illustrated by the following examples:

Example 3 Let us consider the following sets of IC's:

$$\mathcal{C}_1 = \left\{ \begin{array}{rcl} p, \neg q & \longrightarrow & \texttt{inc} \\ r, q & \longrightarrow & \texttt{inc} \end{array} \right\} \text{ and } \mathcal{C}_2 = \left\{ \begin{array}{rcl} r, p, q & \longrightarrow & \texttt{inc} \\ \neg s, q, p & \longrightarrow & \texttt{inc} \end{array} \right\}$$

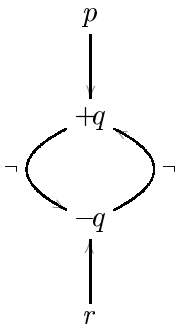
We first derive respectively $\mathcal{P}_{\mathcal{C}_1}$ and $\mathcal{P}_{\mathcal{C}_2}$ such that:

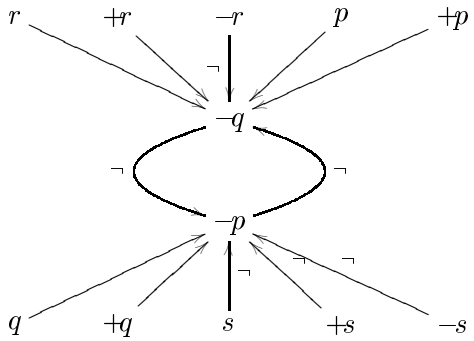
$$\mathcal{P}_{\mathcal{C}_1} = \left\{ \begin{array}{rcl} p & \longrightarrow & +q \\ r & \longrightarrow & -q \end{array} \right\} \text{ et } \mathcal{P}_{\mathcal{C}_2} = \left\{ \begin{array}{rcl} r, p & \longrightarrow & -q \\ \neg s, q & \longrightarrow & -p \end{array} \right\}$$

$\mathcal{P}_{\mathcal{C}_1}$ is a PC program; let $\mathcal{P}_1$ be the program obtained from $\mathcal{P}_{\mathcal{C}_1}$ after transformations 1 and 2. $\mathcal{P}_1$ contains a cycle where $+q$ depends on $\neg -q$ and $-q$ depends on $\neg +q$.
$\mathcal{P}_{\mathcal{C}_2}$ is a NR program because the program $\mathcal{P}_2$ obtained from $\mathcal{P}_{\mathcal{C}_2}$ after transformations 3 and 4 presents a negative cycle where $-q$ depends on $\neg -p$ and $-p$ depends on $\neg -q$.

$$\mathcal{P}_1 = \left\{ \begin{array}{rcl} p, \neg -q & \longrightarrow & +q \\ r, \neg +q & \longrightarrow & -q \end{array} \right\} \text{ and } \mathcal{P}_2 = \left\{ \begin{array}{rcl} r, \neg -r, p, \neg -p & \longrightarrow & -q \\ +r, p, \neg -p & \longrightarrow & -q \\ r, \neg -r, +p & \longrightarrow & -q \\ +r, +p & \longrightarrow & -q \\ \neg s, \neg +s, q, \neg -q & \longrightarrow & -p \\ -s, q, \neg -q & \longrightarrow & -p \\ \neg s, \neg +s, +q & \longrightarrow & -p \\ -s, +q & \longrightarrow & -p \end{array} \right\}$$

The dependency graphs of these programs are





□

Now one may wonder whether for each $\mathcal{C}$ there exists $\mathcal{P}_{\mathcal{C}}$ such that $\bar{\mathcal{P}}_{\mathcal{C}}$ is stratified. The following example shows that this is not the case.

Example 4 Consider the functional dependency $X \longrightarrow Y$ expressed by the denial

$$c : p(X, Y), p(X, Y'), Y \neq Y' \longrightarrow \texttt{inc}$$

The predicate p is *repeated* in c. The only update rule we may infer is

$$\rho_c : p(X, Y), Y \neq Y' \longrightarrow -p(X, Y')$$

which is negatively recursive:

$$\bar{\rho_c} = \left\{ \begin{array}{rcl} p(X,Y), \neg -p(X,Y), Y \neq Y' & \longrightarrow & -p(X,Y') \\ +p(X,Y), Y \neq Y' & \longrightarrow & -p(X,Y') \end{array} \right.$$

□

Definition 5 Let c be a constraint. p is a *repeated* predicate in c if there exists two literals in c both constructed from p and both are positive or negative. □

In the following, we assume that none of the IC's we consider contain repeated predicates.

An other case where it is not possible to derive programs that guarantee total non conflicting update models is the case where the IC's are themselves inconsistent, i.e. there does not exist an instance Δ that satisfies $\mathcal{C}$. e.g,

$$\mathcal{C} = \left\{ \begin{array}{lrcl} c_1: & p(X), q(X) & \longrightarrow & \texttt{inc} \\ c_2: & \neg p(a) & \longrightarrow & \texttt{inc} \\ c_3: & \neg q(a) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

It is generally admitted that "interesting" practical IC's are those satisfied at least by the empty database. From the particular form of the IC's considered in this paper, a sufficient condition such that the empty database would be coherent is to require that the body of c contains at least one positive literal. c_2 and c_3 above are not satisfied by the empty database. In the following, we suppose that all IC's satisfy this condition.

3 Deriving programs from constraints

In this section, we will first give an algorithm that derives a first program $\mathcal{P}^1_{\mathcal{C}}$ which is not negatively recursive. Then, we present a way to transform $\mathcal{P}^1_{\mathcal{C}}$ into $\mathcal{P}_{\mathcal{C}}$ such that for each Δ and each update δ, $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_{\mathcal{C}})$ is bivalued.

3.1 Non negatively recursive programs

In order to derive non negatively recursive programs, we will use the notion of IC graph.

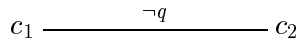
Definition 6 (Constraints graph) Let $\mathcal{C}$ be a set of constraints. $\mathcal{G}_{\mathcal{C}}$ is a non oriented graph $\langle V, E \rangle$ where V is the set of IC's and $(c_i, c_j) \in E$ labeled with p iff the predicate p appears in both c_i and c_j with the same sign, i.e. positive or negative. If p appears in opposite signs the edge is labeled with $\neg p$.

□

Example 5 Let

$$\mathcal{C} = \left\{ \begin{array}{lrcl} c_1: & p(X), q(X) & \longrightarrow & \texttt{inc} \\ c_2: & r(X), \neg q(X) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

$\mathcal{G}_\mathcal{C}$ is given below:



Notice that from $\mathcal{C}$, we can derive a potentially conflicting program i.e. one rule that inserts into Q and another whose action is a deletion from Q.

□

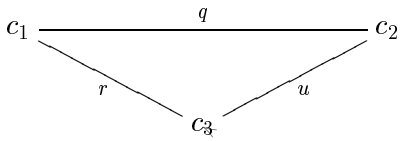
Lemma 2 Let $\mathcal{C}$ be a set of constraints and $\mathcal{G}_\mathcal{C}$ its associated graph. If $\mathcal{G}_\mathcal{C}$ does not contain a negative edge, then each $\mathcal{P}_\mathcal{C}$ is not potentially conflicting.

□

Example 6 Let

$$\mathcal{C} = \left\{ \begin{array}{lll} c_1: & p(X), q(X), r(X) & \longrightarrow \texttt{inc} \\ c_2: & s(X), q(X), \neg u(X) & \longrightarrow \texttt{inc} \\ c_3: & v(X), \neg u(X), r(X) & \longrightarrow \texttt{inc} \end{array} \right\}$$

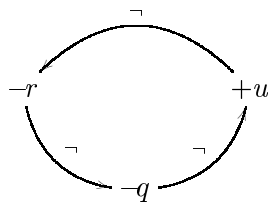
The corresponding graph is



Notice that the graph contains a cycle. Notice also that we can derive a negatively recursive program from $\mathcal{C}$, namely the program:

$$\mathcal{P}_\mathcal{C} = \left\{ \begin{array}{lll} p(X), r(X) & \longrightarrow & -q(X) \\ s(X), q(X) & \longrightarrow & +u(X) \\ v(X), \neg u(X) & \longrightarrow & -r(X) \end{array} \right\}$$

The dependency graph of $\bar{\mathcal{P}}_\mathcal{C}$ contains the negative cycle



□

Lemma 3 If $\mathcal{G}_\mathcal{C}$ does not contain cycles, then $\mathcal{P}_\mathcal{C}$ is negatively recursive only if there are constraints with repeated predicates.

□

Definition 7 (p_Clique) A *p_Clique* in $\mathcal{G}_\mathcal{C}$ is a clique where all edges are labeled with p. □

All IC's that belong to the same p_clique, can be maintained by issuing the same operation (insertion or deletion) into p. In Example 6, $\{c_1, c_2, c_3\}$ is a clique, but not a p_clique. However, $\{c_1, c_2\}$ is a q_clique. If we decide to maintain these two IC's by performing deletions from q, then this is equivalent to delete the edge between c_1 and c_2, the so reduced graph does not contain any remaining cycle. So whatever the action we decide to perform for maintaining c_3, the obtained program is not negatively recursive. This gives an idea of how $\mathcal{G}_\mathcal{C}$ is used in order to derive non negatively recursive programs.

1st algorithm. The following algorithm derives non negatively recursive programs but which may contain conflicting rules.

Input: A set of constraints $\mathcal{C}$.
Output: A program $\mathcal{P}_\mathcal{C}$ non negatively recursive.
Begin
  $\mathcal{P}_\mathcal{C} := \emptyset$;
  Let $\mathcal{G}_\mathcal{C}$ be the graph of $\mathcal{C}$;
  While $\mathcal{G}_\mathcal{C}$ contains connected vertices **Do**:
    Choose a maximal p_clique CL
    For each $c \in CL$ **Do**
      Derive a rule $\rho(c)$ which updates p;
      $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
      Remove c from $\mathcal{G}_\mathcal{C}$
    End For
    For each $c \in \mathcal{G}_\mathcal{C}$ that depends on p **Do**
      Derive a rule $\rho(c)$ which updates p;
      $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
      Remove c from $\mathcal{G}_\mathcal{C}$
    End For
  End While
  For each remaining constraint c **Do**
    Derive a rule $\rho(c)$;
    $\mathcal{P}_\mathcal{C} := \mathcal{P}_\mathcal{C} \cup \{\rho(c)\}$;
    Remove c from $\mathcal{G}_\mathcal{C}$
  End For
  Return $\mathcal{P}_\mathcal{C}$
End

Proposition 2 Let $\mathcal{C}$ be a set of IC's. Then $\mathcal{P}_\mathcal{C}$ the output of the above algorithm is not negatively recursive. □

3.2 Managing conflicting rules

In this section, we show how to transform a potentially conflicting but non negatively recursive program in such a way that it insures a total well founded model.

We first consider the simple case of two rules and illustrate it with the following example:

Example 7 Let $\mathcal{C} = \{p(X), \neg q(X) \longrightarrow \texttt{inc};\ \ r(X), q(X) \longrightarrow \texttt{inc}\}$ and

$$\mathcal{P}_{\mathcal{C}} = \left\{ \begin{array}{l} p(X) \longrightarrow +q(X) \\ r(X) \longrightarrow -q(X) \end{array} \right\}$$

$\mathcal{P}_{\mathcal{C}}$ is potentially conflicting. Consider Δ, where

p
a
b

r
a
c

q
a
b
c

$\mathcal{M}_{\Delta}(\mathcal{P}_{\mathcal{C}}) = \Delta \cup \{\neg p(c), \neg r(b), +q(b), -q(c), \neg -q(b), \neg +q(c)\}$. The literals $+q(a)$ and $-q(a)$ are unknown. It is easy to see that $\mathcal{C}$ is equivalent to:

$$\mathcal{C}' = \left\{ \begin{array}{ll} p(X), \neg r(X), \neg q(X) & \longrightarrow \texttt{inc} \\ r(X), \neg p(X), q(X) & \longrightarrow \texttt{inc} \\ p(X), r(X) & \longrightarrow \texttt{inc} \end{array} \right\}$$

From $\mathcal{C}'$, we derive the program:

$$\mathcal{P_C}' = \left\{ \begin{array}{lll} p(X), \neg r(X) & \longrightarrow & +q(X) \\ r(X), \neg p(X) & \longrightarrow & -q(X) \\ r(X) & \longrightarrow & -p(X) \end{array} \right\}$$

The obtained database by using this program is $\Delta' = \{p(b), r(a), r(c), q(b)\}$.
Notice that:

1. Δ' satisfies $\mathcal{C}$,
2. The IC's maintained by $\mathcal{P_C}'$ are exactly the same as those maintained with $\mathcal{P}_{\mathcal{C}}$; i.e. the expression

$$\forall X : [\underline{(p(X) \Rightarrow r(X) \vee q(X))} \wedge \underline{(r(X) \wedge q(X) \Rightarrow p(X))} \wedge \underline{(r(X) \wedge p(X) \Rightarrow)}]$$

$\mathcal{C}'$, is equivalent to

$$\forall X : [\underline{(p(X) \Rightarrow q(X))} \wedge \underline{(r(x) \wedge q(X) \Rightarrow)}].$$

Finally, notice that $\mathcal{P_C}'$ is not the only program we may derive. Indeed, with

$$\mathcal{P_C}'' = \left\{ \begin{array}{lll} p(X), \neg r(X) & \longrightarrow & +q(X) \\ r(X), \neg p(X) & \longrightarrow & -q(X) \\ p(X) & \longrightarrow & -r(X) \end{array} \right\}$$

we obtain $\Delta'' = \{p(a), p(b), r(c), q(a), q(b)\}$ which is also consistent.

□

3.2.1 The case of two conflicting rules

Consider the IC's

$$\mathcal{C}_1 = \left\{ \begin{array}{lll} \ell_1, \ldots, \ell_m, p(\tilde{X}), c_1(\tilde{X}_1) & \longrightarrow & \texttt{inc} \\ m_1, \ldots, m_n, \neg p(\tilde{X}), c_2(\tilde{X}_2) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

where ℓ_i and m_j are base literals, and c_i is a conjunction of interpreted literals (e.g. $X = 3, Y > 20, X = Y, X \neq Y$). From these IC's, we derive the rules

$$\mathcal{P}_{\mathcal{C}_1} = \left\{ \begin{array}{lll} \rho_a : \ \ell_1, \ldots, \ell_m, c_1(\tilde{X}_1) & \longrightarrow & -p(\tilde{X}) \\ \rho_b : \ m_1, \ldots, m_n, c_2(\tilde{X}_2) & \longrightarrow & +p(\tilde{X}) \end{array} \right\}$$

These rules may, à priori, introduce conflicts. Now Consider the IC's:

$$\mathcal{C}_2 = \left\{ \begin{array}{lll} \ell_1, \ldots, \ell_m, c_1(\tilde{X}_1) & \longrightarrow & p'(\tilde{X}') \\ m_1, \ldots, m_n, c_2(\tilde{X}_2) & \longrightarrow & p''(\tilde{X}'') \\ p'(\tilde{X}'), \neg p''(\tilde{X}''), p(\tilde{X}) & \longrightarrow & \texttt{inc} \\ p''(\tilde{X}''), \neg p'(\tilde{X}'), \neg p(\tilde{X}) & \longrightarrow & \texttt{inc} \\ \ell_1, \ldots, \ell_m, c_1(\tilde{X}_1), m_1, \ldots, m_n, c_2(\tilde{x_2}) & \longrightarrow & \texttt{inc} \end{array} \right\}$$

Where $\tilde{X}'$ is the set of variables in $\tilde{X}_1$ and $\tilde{X}'' = \tilde{x}$. From this new set of IC's, we derive

$$\mathcal{P}_{\mathcal{C}_2} = \left\{ \begin{array}{lrcl} \rho_1 : & \ell_1, \ldots, \ell_m, c_1(\tilde{X}) & \longrightarrow & p'(\tilde{X}') \\ \rho_2 : & m_1, \ldots, m_n, c_2(\tilde{X}) & \longrightarrow & p''(\tilde{X}'') \\ \rho_3 : & p'(\tilde{X}'), \neg p''(\tilde{X}'') & \longrightarrow & -p(\tilde{X}) \\ \rho_4 : & p''(\tilde{x}''), \neg p'(\tilde{x}') & \longrightarrow & +p(\tilde{X}) \\ \rho_5 : & m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n, c_2(\tilde{X}), \ell_1, \ldots, \ell_m, c_1(\tilde{X}) & \longrightarrow & op|m_i| \end{array} \right\}$$

Where $op|m_i| = \left\{ \begin{array}{ll} +(\neg m_i) & \text{if } m_i \text{ is negative} \\ -m_i & \text{otherwise} \end{array} \right\}$

Lemma 4 $\mathcal{C}_1$ is equivalent to $\mathcal{C}_2$. □

The body of the rules ρ_3 and ρ_4 in $\mathcal{P}_{\mathcal{C}_2}$, cannot both be true (they mutually exclude each other). Thus, we cannot be in a situation where we want to insert and delete the same tuple from p. Is this enough to say that we have avoided all unknown updates? In other words, can we be sure that the new program is not negatively recursive? The answer to this question depends on the rule ρ_5. Indeed, the new program is negatively recursive only if ρ_5 itself is.

Lemma 5 ρ_5 cannot be negatively recursive. □

Poof: First, notice that ρ_5 cannot be negatively recursive because of a dependency between m_i and an m_j, otherwise this would mean that m_j and m_i have the same sign which contradicts the fact that we don't have repeated predicates.

However, if there exists an ℓ_j s.t. ℓ_j and m_i can be unified (i.e. there exists a substitution θ with $\ell_j\theta = m_i\theta$), then ρ_5 is negatively recursive. Notice that in this case, ℓ_i and m_j are built from the same predicate. We can assume without loss of generality that for every predicate p and every litteral appearing in denials bodies of the form $p(\tilde{X})$ the terms appearing in $\tilde{X}$ are only variables and always the same, i.e. $p(X, a)$ can be replaced by $p(X, Y) \wedge Y = a$ and $p(X, X)$

is replaced by $p(X,Y) \wedge X = Y$. Notice that this assumption is no longer true when accepting repeated predicates. Hence, ℓ_i and m_i are identical. Which means that the condition $\ell_i \wedge m_j$ is equivalent to m_j. This allows us to eliminate ℓ_j from the body of ρ_5. This way, it is not negatively recursive any more. ■

3.3 The case of multiple conflicting rules

In the previous section, we have shown how to solve the problem of conflicting updates where we have only two rules. In this section, we generalize that method to the case of multiple rules. Consider the program:

$$\begin{array}{lll} \rho_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X_p}) & \longrightarrow \quad -p(\tilde{X}) \\ \hline \rho_{p+1} : & m_1^1, \ldots, m_{n_1}^1, c_{p+1}(\tilde{X}_{p+1}) & \longrightarrow \quad +p(\tilde{X}) \\ & \ldots & \\ \rho_{p+q} : & m_1^q, \ldots, m_{n_1}^q, c_{p+q}(\tilde{X}_{p+q}) & \longrightarrow \quad +p(\tilde{X}) \end{array}$$

The rule ρ_p conflicts with the other rules. The solution is to proceed in a way such that the condition (i.e. body) of ρ_p and those of the other rules cannot be true at the same time. For this purpose, we transform the program as follows:

$$\begin{array}{lll} \rho'_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X_p}) & \longrightarrow \quad p'(\tilde{X}) \\ \hline \rho'_{p+1} : & m_1^1, \ldots, m_{n_1}^1, c_{p+1}(\tilde{X}_{p+1}) & \longrightarrow \quad p''(\tilde{X}) \\ \ldots & & \\ \rho'_{p+q} : & m_1^q, \ldots, m_{n_q}^q, c_{p+q}(\tilde{X}_{p+q}) & \longrightarrow \quad p''(\tilde{X}) \\ \hline\hline \\ \rho''_p : & \ell_1, \ldots, \ell_m, c_p(\tilde{X_p}), \neg p''(\tilde{X}) & \longrightarrow \quad -p(\tilde{X}) \\ \hline \rho''_{p+1} : & m_1^1, \ldots, m_{n_1}^1, c_{p+1}(\tilde{X}_{p+1}), \neg p'(\tilde{X}) & \longrightarrow \quad +p(\tilde{X}) \\ & & \ldots \\ \rho''_{p+q} : & m_1^q, \ldots, m_{n_q}^q, c_{p+q}(\tilde{X}_{p+q}), \neg p'(\tilde{X}) & \longrightarrow \quad +p(\tilde{X}) \\ \hline \end{array}$$

The only dependencies generated by these rules are those already existing in the initial program. Thus, if the first program is initially non negatively recursive, then the new one is also non negatively recursive.

The remaining case is that where the condition of the rule which makes an insertion and the condition of one of the rules making a deletion are both true, i.e we should consider the IC's of the form

$$c_i : body(\rho_p), body(\rho_{p+i}) \longrightarrow \texttt{inc}$$

To maintain this IC, we must derive a rule while preserving the property of non negative recursion. The following may arise:

- Suppose that there exists a literal M in $body(c_i)$ s.t. there exists ρ with $body(\rho) \ni L$ where L and M are two literal from the same predicate Q. This means that we can update Q in order to maintain c_i without creating a negative cycle.
- Suppose now that their does not exist such an M. It is easy to see that this contradicts the hypothesis that the initial program was non negatively recursive.

In this section, we have first proposed an approach that allows to derive update programs that are not negatively recursive. Then, we have presented a method that solves the problem of potentially conflicting rules. The combination of the two gives programs that satisfy the following proposition.

Proposition 3 Let $\mathcal{C}$ be a set of constraints and $\mathcal{P}_\mathcal{C}$ the program obtained after the two steps described in this section. Then, for each update δ and instance Δ, the update model $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_\mathcal{C})$ is total and non conflicting.

□

So, we can use $\mathcal{P}_\mathcal{C}$ to maintain and restore database consistency. Concerning the complexity of this process is however exponential. The first algorithm is in linear time w.r.t the number of IC's, while the second part of the derivation is exponential.

Now we address the well known property of minimal change [KM91]. Let δ be a user update submitted to the database Δ. Δ_{user} may be seen as the database the user wants to get by his/her update. However, Δ_{user} may be not consistent. So in order to restore consistency, Δ_{user} should be updated. Let Δ_{final} be the coherent database obtained after updating Δ_{user}. Intuitively, the minimal change property says that the "quantity of change" we make to Δ_{user} to reach a consistent database must be as little as possible. Formally,

Definition 8 Δ_{final} fulfills the minimal change property iff $\forall \Delta'$coherent : $\Delta_{user} \div \Delta' \subseteq \Delta_{user} \div \Delta_{final} \Rightarrow \Delta_{user} \div \Delta' = \Delta_{user} \div \Delta_{final}$, where $\div$ is the symmetrical set difference.

□

Proposition 4 If $\mathcal{M}_{\Delta_{user}}(\mathcal{P}_\mathcal{C})$ is total and not conflicting (which is the case with the programs we derive), then Δ_{final} fulfills the minimal change property.

□

4 Expressiveness

Let's now address the expressive power of the update semantics with update rules. An update μ in a language $\mathcal{L}$ can be considered as a sequence of two operations ϕ, ψ where ϕ is a query and ψ is an assignment. Hence, $\mathcal{L} \equiv \mathcal{Q}; FO^{:=}$, where $\mathcal{Q}$ is a query language and $FO^{:=}$ is first order logic with destructive assignment. For instance, consider the operation of deleting all nodes for which there is a cycle passing through. We first compute the nodes that are not in any cycle

(the query part), then we assign this set to the set of nodes (the assignment part). Thus, the expressiveness of an update language is fully determined by it's query language part. In our update semantics, we first we derive updates, then we apply them. The expressiveness of the first part is that of *Fixpoint* [AHV95] since we use well founded semantics which is equivalent to *Fixpoint*. Notice that the combination of the two parts is a subset of the *While* language[1]. Let $\mathcal{U}$ denote our update semantics. Then

$$\mathcal{U} \subset While$$

since $\mathcal{U}$ clearly belongs in PTIME while the language $While$ is in PSPACE and it is conjectured that PTIME $\neq$ PSPACE.

5 Conclusion

In this paper, we have proposed an algorithm that derives update programs from IC's. These programs can not only maintain database consistency but also restore it. This is of a particular interest when IC's can be added dynamically.

In Example 4 we saw that it is not possible to restore the functional dependency constraint while guaranteeing the minimal change property with a deterministic language (the case of well founded Datalog). This example suggests the use of the *choice* construct [GPSZ97] which introduces a kind of controlled non determinism while leaving the computational complexity in PTIME which is not the case of the stable semantics[GL88] used by [MT98]. In fact, there are two kinds of non determinism: tuple and predicate non determinism. We solved the latter during the update rules derivation.

As future research we would like to investigate is the following problem. let's say that an update language $\mathcal{L}$ can *maintain* a class of IC's $\mathcal{C}$, denoted $\mathfrak{M}^{\mathcal{C}}_{\mathcal{L}}$ iff for all $C \subseteq \mathcal{C}$, there exists a program $\mathcal{P} \in \mathcal{L}$ such that for all Δ satisfying C and for each update δ, if Δ_{user} is defined as before and $\mathcal{P}(\Delta_{user})$ is the final database, then $\mathcal{P}(\Delta_{user}) \div \Delta_{user}$ is minimal. We say that $\mathcal{L}$ can *restore* $\mathcal{C}$, denoted $\mathfrak{R}^{\mathcal{C}}_{\mathcal{L}}$ if the above definition holds when Δ is not supposed consistent and $\delta = \emptyset$. In other words, $\mathcal{P}$ can restore database consistency with minimal change. For instance, for the update rule

$$p(X, Y), Y \neq Y' \longrightarrow \neg p(X, Y')$$

derived from the functional dependency, the update models that can restore database consistency with minimal change are exactly those which are stable models. Giving general relationships between $\mathcal{L}$ and $\mathcal{C}$ could be an interesting direction of future research.

In [FP97] the authors address the following problem: given a set of IC's $\mathcal{C}$, for each $c \in C$ there may be many ways to maintain c. To express his/her preferences, the DB administrator assigns a weight to each alternative, the problem is to find a program $\mathcal{P}$ which is guaranteed to terminate while maximizing the global weight. The authors show that this is a NP hard problem. In our paper, we have not considered user preferences when deriving programs. It could be interesting to use the change metrics between the instances generated by two programs as the distance between them and use this definition to generate programs which are among the closest to those proposed by the user.

[1] Actually, *While* can be considered as both a query and an update language.

Aknowledgment

The author would thank the referees for their careful reading and fruitful comments which helped to improve the presentation of this work.

References

[AART97] D. Aquilino, P. Asirelli, C. Renso, and F. Turini. Applying restriction constraints to deductive databases. *Annals of Mathematics and Artificial Intelligence*, 19(I-II):3–25, 1997.

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

[ALS95] M. Halfeld Ferrari Alves, D. Laurent, and N. Spyratos. Update rules in datalog programs. In *Proceedings of LPNMR'95, Logic Programming and Non Monotonic Reasoning*, LNAI 928. Springer Verlag, 1995.

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinisme. In *Proceedings of the ACM Sigmod conference*, San Diego, USA, June 1992.

[BCP95] E. Baralis, S. Ceri, and S. Paraboshi. Run-time detection of non-terminating active rule systems. In T. Ling, A. Mendelzon, and L. Vieille, editors, *Proceedings of DOOD conference*, volume 1013 of *LNCS*, pages 38–54. Springer, 1995.

[BDR98] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of PODS conference*, 1998.

[BM97] N. Bidoit and S. Maabout. A model theoretic approach to update rule programs. In *Proceedings of ICDT Conference*, LNCS. Springer, 1997.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboshi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM TODS*, 19(2):367–422, 1994.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases.* Springer, 1990.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th VLDB conference*, 1990.

[DDS98] M. Dekhtyar, A. Dikovsky, and N. Spyratos. On Logically Justified Updates. In *Proceedings of JICSLP conference.* MIT Press, 1998.

[FP97] P. Fraternali and S. Paraboschi. Ordering and selecting production rules for constraint maintenance. Complexity and heuristique solution. *IEEE TKDE*, 9(1):173–178, 1997.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Proceedings of the 5th international symposium on logic programming.* MIT Press, 1988.

[GPSZ97] F. Giannotti, D. Pedreshi, D. Saccà, and C. Zaniolo. Programming with Non-determinism in Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(3-4), 1997.

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledgebase and revisiting it. In *Proceedings of Intrl. Conf. on Knowledge Represention and Reasoning (KR'91)*, 1991.

[LM98] B. Ludäscher and W. May. Referential actions: From logical semantics to implementation. In *Proceedings of EDBT'98 conference*, volume 1377 of *LNCS*, Valencia, Spain, Mar. 1998.

[LML97] B. Ludäscher, W. May, and G. Lausen. Referential actions as logic rules. In *Proceedings of PODS conference*, 1997.

[MT98] V. Marek and M. Truszczyǹski. Revision programming. *Theor. Computer Science(TCS)*, 190(2):241–277, 1998.

[ST96] K. D. Schewe and B. Thalheim. Active consistency enforcement for repairable database transitions. In *Proceedings Int. Workshop on foundation of models and languages for databases and objects: Integrity in Databases*, Sept. 1996.

[TO95] E. Teniente and A. Olivé. Updating knowledge bases while maintaining their consistency. *VLDB journal*, 4(2):193–241, 1995.

[UD92] S.D. Urban and M. Desideiro. CONTEXT: a CONnsTraint EXplanation Tool. *Data & Knowledge Enginieering*, 8:153–183, 1992.

[vGRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38, 1991.