Formalising Existential Rule Treatment in the Automatic Synthesis of Update Transactions in Deductive Databases

Joan A. Pastor

Report LSI–94–27–R

Formalising Existential Rule Treatment in the Automatic Synthesis of Update Transactions in Deductive Databases*

Joan A. Pastor
Universitat Politècnica de Catalunya
Dept. de LSI - Facultat d'Informàtica
Pau Gargallo, 5
08028 Barcelona, Catalonia
pastor@lsi.upc.es

ABSTRACT

We propose a new method for generating consistency-preserving transaction programs for (view) updates in deductive databases. The method augments the deductive database schema with a set of transition and internal events rules, which explicitly define the database dynamic behaviour in front of a database update. At transaction-design-time, a formal procedure can use these rules to automatically generate parameterised transaction programs for base or view-update transaction requests. This is done in such a way that those transactions will never take the database into an inconsistent state. In this report we extend a previous version of the method by incorporating existentially defined rules, and formalising their treatment. Within this context, synthesis outputs and processes are provided. The method, implemented in Prolog using meta-programming techniques, draws from our previous work in deductive databases, particularly in view updating and integrity constraints checking.

KEYWORDS

Deductive databases, Database Software Synthesis, Transaction synthesis,
View updating, Integrity checking, Integrity enforcement

---

* To make it self-contained, this report includes a slightly modified version of [Pas94b], plus an additional new Appendix that concentrates on the most formal concepts and procedures of the method.

1. INTRODUCTION AND PREVIOUS WORK

First relational DBMSs were only able to handle facts, stored in *base relations*. More recent systems permit the definition and management of limited forms of derived information, the so called *views*, and of *integrity constraints*. In a similar way that queries to the database are handled by a query processing system, the DBMS includes a *transaction processing system* that provides the users with a uniform interface, through which they can update at least base relations. However, in the presence of views and integrity constraints, additional features of the transaction system should be the direct processing of view updates and the checking of integrity constraints. Furthermore, besides just checking, the system could try to enforce consistency by doing some proactive work in order to prevent and/or cure a consistency violation.

Thus we come to the driving origins of this work: View updating, Integrity checking and Integrity enforcement. We address these three strongly related problems in relational and deductive databases through an integrative solution consisting in the automatic generation of *Treks* (**T**ransactions for the **e**nforcement of **k**nowledge). These are *transactions programs* synthesised from a database schema and a proposed update transaction request (possibly view-update). Treks are built in such a way that they incorporate the necessary integrity checks and all possible integrity repairs that can be drawn out of the schema. Thus the database will never be taken into an inconsistent state, as long as run-time updates are only effected through the instantiation of the previously obtained Treks.

Treks may be generated to be used in three related contexts. First, they may be embedded by the application designer in his/her application programs as modules to update the database. Second, the database administrator may make them available to application designers and/or end-users as the only way to update the database. Finally, they might be eventually synthesised by a future DBMS on request from an end-user, application designer or database administrator. In this last case, Treks can be seen as 'update plans' to perform the request.

Before addressing the databases treated and the illustration of our method, let us briefly comment on the three database problems mentioned above and on previous related work, including our own.

The problem of *view updating* in databases is concerned with determining how a request to update a view can be appropriately translated into one or more updates of the underlying base relations. The main objective of most of the methods worked out for view updating (see [TO92] and the references therein) is that of providing effective ways for doing such a translation. This means that most methods have left for subsequent research the efficiency objective, specially those methods of an interpretative or run-time oriented nature. Regarding our group's work on the subject, in [TO92,Ten92] a new view updating method was proposed for deductive databases. Although being more effective that previous methods for relational and deductive databases, the efficiency issue was not yet considered.

Integrity checking is concerned with detecting when an update operation on the database will cause this one to become inconsistent. Integrity constraints (ICs from now on) are a way to state such inconsistent situations in a declarative and high-level way. A great amount of research has been devoted to the field of ICs checking during the last fifteen years (see [BMM90] for a state-of-the-art survey). Methods developed so far differ in the kind of databases considered, in the kind of ICs they

can check, in the type of transactions and updates allowed and, of course, in the particular approach taken by each method. All approaches share the aim of evaluating efficiently the ICs. Our group's previous work on ICs checking, also in the context of deductive databases, resulted in a powerful run-time oriented method [Oli89,Oli91], whose efficiency was proposed to be further improved in [OP90] with some precompiled work, and in [UO92] in the context of change computation optimisation. Unfortunately, that method shares with the previous ones a very limited form of response in case of consistency violation, i.e the rejection of the update.

In front of a transaction containing one or more consistency violating updates, the classical treatment simply amounts to rolling back the transgressing update or transaction. This simple rejection solution alone is clearly unsatisfactory for most real databases. Instead, the request should optionally be treated in a more co-operative or helpful way by the ICs maintenance module. For example, the system could try to compensate the user's violating updates with further updates drawn from the ICs. These extra updates could be prepared at compile-time and used to extend the user's transactions in a preventive manner. Surprisingly, only a few methods have recently appeared with this objective in mind (see section 6 for a brief description of some of them). We like to refer to this integrity maintenance activity as to *Integrity enforcement*.

View updating, Integrity checking and Integrity enforcement are strongly related for natural reasons. For example, in generating base updates from view update requests, the former have to be checked for consistency. Doing this results in procedures applying similar underlying ideas as the ones found in the Integrity enforcement arena. On the other way around, the representational power of ICs becomes much powerful when these are used in conjunction with views. In fact, both relational [CW90] and deductive [ML91] integrity enforcement methods happen to run into the view update problem.

In [PO94,Pas94a] we proposed to treat all the three problems in an integrative transaction-design-time oriented method, which we extend here to deal with a much more general case, that of synthesising transaction programs from database schemes with existentially defined rules. To make it self-contained, this report includes a slightly modified version of [Pas94b]. Additionally, a new Appendix concentrates on the most formal concepts and procedures of the method.

This report is organised as follows. Next section defines the deductive database schemes currently accepted by our method and introduces the example that will be used throughout the report. Section 3 reviews the components of the augmented database schema, a key concept for the method. Section 4 presents our method for generating consistency-preserving transactions and illustrates it through several detailed examples. In section 5 we comment on some additional features of the method not exemplified before. In section 6 we relate our approach with previous comparable methods. Finally, in section 7 we present our conclusions and comment on future work. After the references, an Appendix includes the detailed formalisation of the central core phase of our method.

2. DEDUCTIVE DATABASE SCHEMES CONSIDERED

We define here the kind of deductive database schemes treated in this report. Since ours is a compile-time method, most of the information we use comes precisely from the database schema. In the sequel, we give the intuitive meaning as well as formal definitions and examples for the most important concepts. We use F.O.L. as the main formalism.

A deductive database schema DBS consists of three finite sets: a set B of base predicates, a set D of deductive rules, and a set I of ICs. Base predicates are the schemes of the facts explicitly stored in the database, which form the so called extensional database. Derived predicates (or views) are schemes representing information that, unless materialised, is not stored in the database but can be derived using deductive rules. ICs are used to specify unwanted database states and forbidden database transitions.

We assume that database predicates are either base or derived predicates. A base predicate appears only in the extensional database, as a ground atom or fact, and (eventually) in the body of deductive rules and within ICs. A derived predicate appears as head of deductive rules, and (eventually) also in the body of deductive rules and ICs. Deductive rules and ICs can be defined in terms of base, derived and evaluable predicates.

Before providing more formal definitions for some of the previous concepts, let us introduce the base predicates corresponding to the database example that we will be using throughout the report. They are shown in Fig. 2-1, together with their intended meaning. Our example, inspired upon those of [KSS87,Qia93], is a database for an "Employment Office" that arranges labour interviews between its registered job applicants and employer companies. For the people administered by the office, it also keeps track of employees and, for legal reasons, of nationality status and of the existence of criminal records.

Fig. 2-1

Base predicate	Base predicate meaning
Emp(x)	'x' is an employee
App(x)	'x' is a job applicant
Eco(y)	'y' is an employer company
Int(x,y)	'x' has an interview with 'y'
Cit(x)	'x' is a citizen
Ra(x)	'x' is a registered alien
Cr(x)	'x' has some criminal record

2.1 Deductive rules

Formally, a deductive rule is a formula of the form:

$$A \leftarrow L_1 \wedge ... \wedge L_n \quad \text{with } n \geq 1$$

where A is an atom denoting the conclusion or derived predicate, and the L_1 ,..., L_n are literals representing the conditions, which can be base, derived or evaluable predicates, possibly negated. Evaluable predicates are system predicates, such as the comparison or arithmetic predicates, that can be evaluated without accessing the database. Any variables in A, L_1 ,..., L_n are assumed to be universally quantified over the whole formula. The terms in the conclusion must be distinct variables,

and the terms in the conditions must be variables or constants. That is, we include rules with existential variables (i.e. those variables not appearing in the conclusion, also called local variables). As usual, we require that the database schema is allowed, that is any variable that occurs in a deductive rule has an occurrence in at least one of its positive conditions.

Fig. 2-2 shows the two derived predicates of our example with their corresponding deductive rules. The right of residence status of a person is defined using two deductive rules. One existential rule is used for defining job candidates.

Fig. 2-2

Derived predicate + deductive rules	Derived predicate meaning
$Rr(x) \leftarrow Ra(x) \wedge \neg Cr(x)$ $Rr(x) \leftarrow Cit(x)$	'x' has right of residence if s/he is either a registered alien with no criminal record or a citizen
$Cand(x) \leftarrow Int(x,y) \wedge Eco(y)$	'x' is considered a job candidate when s/he has an interview with an employer

In this report we extend our previous work to deal with the general case of allowed deductive database schemes with existential rules. A first tentative approach including existential rules was proposed in [Pas92] for the case relational databases with flat views and ICs (i.e. defined only in terms of base predicates). [PO94] considered deductive database schemes without existential rules.

2.2 Integrity constraints

Integrity constraints (ICs) are conditions that the database is required to satisfy at all times. ICs are either *state* (or static), when they must be satisfied in any state of the database, or *dynamic* , when they involve the evolution between two or more database states. Dynamic ICs compelling only one transition between two successive states are further called *transition* ICs. Our method works with state and transition ICs.

Formally, an IC is a closed first-order formula that the database is required to satisfy. We deal with constraints that have the form of a denial:

$$\leftarrow L_1 \wedge ... \wedge L_n \qquad \text{with } n \geq 1$$

where the L_i are literals (i.e. positive or negative base, derived or evaluable predicates) and variables are assumed to be universally quantified over the whole formula. For the sake of uniformity, we associate to each IC an inconsistency predicate Ic*n* , with or without terms, thus taking the same form as deductive rules. We call them integrity rules.

We will use in our example the three state ICs shown in Fig. 2-3. The set of employees is a subset of that of right residents (Ic1) and is disjoint with the set of applicants (Ic2), which is a superset of candidates (Ic3). Although neither of the integrity rules has existential variables note that Ic3(x) is defined in terms of the existentially-defined view Cand(x).

Fig. 2-3

Integrity rule	Integrity constraint meaning
$Ic1(x) \leftarrow Emp(x) \wedge \neg Rr(x)$	Employees must be legal residents
$Ic2(x) \leftarrow Emp(x) \wedge App(x)$	Employees cannot be applicants
$Ic3(x) \leftarrow Cand(x) \wedge \neg App(x)$	Candidates must be applicants

3. THE AUGMENTED DATABASE SCHEMA

In this section we shortly present and define the concepts and terminology of internal events, transition and internal events rules. These are key concepts in our method since we use them to augment the original database schema in order to later on synthesise transactions from them.

Conceptually, internal events, transition rules and internal events rules are meta-level constructs describing the dynamic behaviour of a deductive database when confronted with updates. For that reason, they are explained in run-time terms, as if we had some specific ground updates running against a particular database extension. However, the resulting rules depend only on the deductive database schema. They are independent from the base facts stored in the database, and from any particular update. Their implied dynamic update behaviour is not represented by the database schema solely.

In section 4, we will discuss the use of transition and internal events rules at "transaction-design-time" for transaction synthesis. The following presentation is an adaptation to our context of theory explained elsewhere [for ex. Oli91], where the reader will find the full details on the formal derivation of such transition and internal events rules.

3.1 Internal events

Let D be a database, U an update and D^n the "new" updated database. We say that U induces a transition from D (the current state) to D^n (the new, updated state). We assume that U consists of an unspecified set of base facts to be inserted and/or deleted.

Due to the deductive rules, U may induce other updates on some derived predicates. Let P be a (derived) predicate in D, and let P^n denote the same predicate evaluated in D^n. Formally, we associate to each base, derived or inconsistency predicate P an insertion internal events predicate ιP and a deletion internal events predicate δP, defined as:

(1) $\forall \mathbf{x}(\iota P(\mathbf{x}) \leftrightarrow P^n(\mathbf{x}) \wedge \neg P(\mathbf{x}))$

(2) $\forall \mathbf{x}(\delta P(\mathbf{x}) \leftrightarrow P(\mathbf{x}) \wedge \neg P^n(\mathbf{x}))$

where $\mathbf{x}$ is a vector of variables. From (1) and (2) we have:

(3) $\forall \mathbf{x}(P^n(\mathbf{x}) \leftrightarrow (P(\mathbf{x}) \wedge \neg \delta P(\mathbf{x})) \vee \iota P(\mathbf{x}))$

(4) $\forall \mathbf{x}(\neg P^n(\mathbf{x}) \leftrightarrow (\neg P(\mathbf{x}) \wedge \neg \iota P(\mathbf{x})) \vee \delta P(\mathbf{x}))$

If P is a base predicate, then ιP facts and δP facts respectively represent insertions and deletions of base facts, i.e. base updates. They will represent derived or view-updates if P is a derived predicate.

If P (i.e. Ic) is an inconsistency predicate, then ιIc facts that occur during the transition will correspond to violations of its corresponding IC. Note that, for an inconsistency predicate Ic, δIc facts cannot happen in any transition, since we assume that the database is consistent before the update and, thus, Ic is always false. Two special-purpose system events are also used, $\iota Abort$ and $\iota Exit$, but their meaning will be clear with the examples of section 4.

3.2 Transition rules

Let us take a base, derived or inconsistency predicate P of the database. The definition of P consists of the rules in the database schema having P in the conclusion. Assume, in general, that there are m ($m \geq 1$) such rules. For our purposes, we require to rename the predicate symbol in the conclusions of the m rules by $P_1, \ldots, P_m$ and add the set of clauses:

$$P(\mathbf{x}) \leftarrow P_i(\mathbf{x}) \qquad i = 1,\ldots, m$$

Consider now one of the rules $P_i(\mathbf{x}) \leftrightarrow L_1 \wedge \ldots \wedge L_q$. When the rule is to be evaluated in the updated state its form is $P^n_i(\mathbf{x}) \leftrightarrow L^n_1 \wedge \ldots \wedge L^n_q$. Now if we replace each literal in the body by its equivalent definition, given in (3) and (4), in terms of the current state (before update) and the internal events, we get a new rule, called a *transition rule*, which defines predicate P^n_i (new state) in terms of current state predicates and of internal events.

It will be convenient to refer to the resulting rules by the formula:

$$(5) \qquad P^n_{i,j}(\mathbf{x}) \leftrightarrow \bigwedge_{r=1}^{r=q} [O(L_r) \mid T(L_r)] \qquad \text{for } j = 1,\ldots, 2^q$$

where q is the number of literals in the P_i rule, and where $O(L_j)$ and $T(L_j)$ are

$$O(L_j) = (Q_j(\mathbf{x}_j) \wedge \neg \delta Q_j(\mathbf{x}_j)) \qquad \text{if } L_j = Q_j(\mathbf{x}_j)$$
$$= (\neg Q_j(\mathbf{x}_j) \wedge \neg \iota Q_j(\mathbf{x}_j)) \qquad \text{if } L_j = \neg Q_j(\mathbf{x}_j)$$

and
$$T(L_j) = \iota Q_j(\mathbf{x}_j) \qquad \text{if } L_j = Q_j(\mathbf{x}_j)$$
$$= \delta Q_j(\mathbf{x}_j) \qquad \text{if } L_j = \neg Q_j(\mathbf{x}_j)$$

That is, $O(L_j)$ defines the part of L_j not changing from the "Old" state, while $T(L_j)$ specifies the part of L_j that changes during the "Transition".

In order to isolate when P remains true because it has not been changed during the transition, it will be useful to assume that in the above set of rules (5) the rule corresponding to j = 1 is:

$$P^n_{i,1}(\mathbf{x}) \leftrightarrow O(L_1) \wedge \ldots \wedge O(L_q)$$

and to refer to it through the rule:

$$P^{nO}_i(\mathbf{x}) \leftarrow P^n_{i,1}(\mathbf{x})$$

Then, for the m rules defining P we may further have:

$$P^{nO}(\mathbf{x}) \leftarrow P^{nO}_i(\mathbf{x}) \qquad i = 1,\ldots, m$$

Similarly, it is also useful to group those rules (5) with $j = 2,\ldots, 2^q$, since they indicate, for definition P_i, all possible ways for P to become true in the new state due to some internal events occurred within the Transition. The grouping rule will be:

$$P^{nT}_i(\mathbf{x}) \leftarrow P^n_{i,j}(\mathbf{x}) \qquad j = 1,\ldots, 2^q$$

Again, considering all m rules defining P we get:

$$P^{nT}(\mathbf{x}) \leftarrow P^{nT}_i(\mathbf{x}) \qquad i = 1,\ldots, m$$

Finally, we may now refer to both P^{nO} and P^{nT} through:

$$P^n(\mathbf{x}) \leftarrow P^{nO}(\mathbf{x})$$
$$P^n(\mathbf{x}) \leftarrow P^{nT}(\mathbf{x})$$

We also consider the above intermediate rules, i.e rules with conclusions P^n, P^{nT} and P^{nO}, *transition rules* for predicate P. Observe that these rules ultimately serve to define predicate P^n (new state) in terms of old state predicates and of internal events predicates.

The transition rules corresponding to the database example are shown in Fig. 3-1. Transition rules for derived predicates (TR.1 to TR.20) and ICs (TR.21 to TR.32) are listed before those for base predicate Emp (TR.33 to TR.36). Transition rules for base predicates App, Eco, Int, Cit, Ra and Cr are similar to those of Emp, and thus have not been presented. Note that the rules take the form of the above formulas, except for the omission of the intermediate predicates $P^n_{i,j}$, which are in fact auxiliary and were only used for presentation purposes. Neither are necessary P^{nO}_i and P^{nT}_i for integrity and base predicates. Also, the meaning of $\delta Rr^{Ic}(x)$ in TR.24 will be clarified in section 3.4.

Fig. 3-1

Code	Transition rule
TR.1	$Rr^n(x) \leftarrow Rr^{nO}(x)$
TR.2	$Rr^n(x) \leftarrow Rr^{nT}(x)$
TR.3	$Rr^{nT}(x) \leftarrow Rr^{nT}_1(x)$
TR.4	$Rr^{nT}(x) \leftarrow Rr^{nT}_2(x)$
TR.5	$Rr^{nO}(x) \leftarrow Rr^{nO}_1(x)$
TR.6	$Rr^{nO}(x) \leftarrow Rr^{nO}_2(x)$
TR.7	$Rr^{nO}_1(x) \leftarrow Ra(x) \wedge \neg \delta Ra(x) \wedge \neg Cr(x) \wedge \neg \iota Cr(x)$
TR.8	$Rr^{nT}_1(x) \leftarrow Ra(x) \wedge \neg \delta Ra(x) \wedge \delta Cr(x)$
TR.9	$Rr^{nT}_1(x) \leftarrow \iota Ra(x) \wedge \neg Cr(x) \wedge \neg \iota Cr(x)$
TR.10	$Rr^{nT}_1(x) \leftarrow \iota Ra(x) \wedge \delta Cr(x)$
TR.11	$Rr^{nO}_2(x) \leftarrow Cit(x) \wedge \neg \delta Cit(x)$
TR.12	$Rr^{nT}_2(x) \leftarrow \iota Cit(x)$
TR.13	$Cand^n(x) \leftarrow Cand^{nO}(x)$
TR.14	$Cand^n(x) \leftarrow Cand^{nT}(x)$
TR.15	$Cand^{nT}(x) \leftarrow Cand^{nT}_1(x)$
TR.16	$Cand^{nO}(x) \leftarrow Cand^{nO}_1(x)$
TR.17	$Cand^{nO}_1(x) \leftarrow Int(x,y) \wedge \neg \delta Int(x,y) \wedge Eco(y) \wedge \neg \delta Eco(y)$
TR.18	$Cand^{nT}_1(x) \leftarrow Int(x,y) \wedge \neg \delta Int(x,y) \wedge \iota Eco(y)$
TR.19	$Cand^{nT}_1(x) \leftarrow \iota Int(x,y) \wedge Eco(y) \wedge \neg \delta Eco(y)$
TR.20	$Cand^{nT}_1(x) \leftarrow \iota Int(x,y) \wedge \iota Eco(y)$
TR.21	$Ic1^{nO}(x) \leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \neg Rr(x) \wedge \neg \iota Rr(x)$
TR.22	$Ic1^{nT}(x) \leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \delta Rr(x)$
TR.23	$Ic1^{nT}(x) \leftarrow \iota Emp(x) \wedge \neg Rr(x) \wedge \neg \iota Rr(x)$
TR.24	$Ic1^{nT}(x) \leftarrow \iota Emp(x) \wedge \delta Rr^{Ic}(x)$
TR.25	$Ic2^{nO}(x) \leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge App(x) \wedge \neg \delta App(x)$
TR.26	$Ic2^{nT}(x) \leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \iota App(x)$
TR.27	$Ic2^{nT}(x) \leftarrow \iota Emp(x) \wedge App(x) \wedge \neg \delta App(x)$
TR.28	$Ic2^{nT}(x) \leftarrow \iota Emp(x) \wedge \iota App(x)$
TR.29	$Ic3^{nO}(x) \leftarrow Cand(x) \wedge \neg \delta Cand(x) \wedge \neg App(x) \wedge \neg \iota App(x)$
TR.30	$Ic3^{nT}(x) \leftarrow Cand(x) \wedge \neg \delta Cand(x) \wedge \delta App(x)$
TR.31	$Ic3^{nT}(x) \leftarrow \iota Cand(x) \wedge \neg App(x) \wedge \neg \iota App(x)$
TR.32	$Ic3^{nT}(x) \leftarrow \iota Cand(x) \wedge \delta App(x)$
TR.33	$Emp^n(x) \leftarrow Emp^{nO}(x)$
TR.34	$Emp^n(x) \leftarrow Emp^{nT}(x)$
TR.35	$Emp^{nO}(x) \leftarrow Emp(x) \wedge \neg \delta Emp(x)$
TR.36	$Emp^{nT}(x) \leftarrow \iota Emp(x)$
TR....	$App^n(x) \leftarrow ...$; $Eco^n(y) \leftarrow ...$; $Int^n(x,y) \leftarrow ...$ $Cit^n(x) \leftarrow ...$; $Ra^n(x) \leftarrow ...$; $Cr^n(x) \leftarrow ...$

Each of the above rules has a clear intuitive meaning. Thus, for example, TR.7 states that 'x' has the right of residence in the new state ($Rr^{nO}{}_1(x)$), if s/he was a registered alien in the old state (Ra(x)), and this fact has not been deleted in the transition ($\neg\delta Ra(x)$), and 'x' did not have a criminal record in the old state ($\neg Cr(x)$), and a criminal record for 'x' has not been inserted during the transition ($\neg\iota Cr(x)$). Similarly, TR.18 states that 'x' is a candidate in the new state ($Cand^{nO}{}_1(x)$), if s/he had a programmed interview with 'y' in the old state (Int(x,y)) that has not been cancelled in the transition ($\neg\delta Int(x,y)$), and 'y' has been inserted as employer company during the transition ($\iota Eco(y)$).

3.3 Insertion internal events rules

Let P be a derived or inconsistency predicate. Once P^n has been formally stated, from formula (1) we get:

(6) $\quad \iota P(\mathbf{x}) \leftarrow P^n(\mathbf{x}) \wedge \neg P(\mathbf{x})$

which is called the *insertion internal events rule* of predicate P, and allows us to deduce which ιP facts (induced insertions) happen in a transition. However, this rule can be simplified in the following ways.

It is easy to prove that no ιP facts can be produced through P^{nO}, since in this case $P^{nO}(\mathbf{x}) \rightarrow P(\mathbf{x})$. We can then simplify (6) to:

$$\iota P(\mathbf{x}) \leftarrow P^{nT}(\mathbf{x}) \wedge \neg P(\mathbf{x})$$

If P is an inconsistency predicate we can further remove the literal $\neg P(\mathbf{x})$ since we assume that $P(\mathbf{x})$ is false, for all **x**, in the old state. For this case we further define general database inconsistency with the standard auxiliary rules

$$\iota Ic \leftarrow \iota Ic_k \qquad k = 1..r$$

where r is the number of ICs in the database and each ιIc_k has its corresponding arguments.

Fig. 3-2 shows the insertion internal events rules for the example, respectively corresponding to derived predicates Rr and Cand, to inconsistency predicates Ic1, Ic2 and Ic3, and to database inconsistency.

Fig. 3-2

Code	Internal events rule
IR.1	$\iota Rr(x) \leftarrow Rr^{nT}(x) \wedge \neg Rr(x)$
IR.2	$\iota Cand(x) \leftarrow Cand^{nT}(x) \wedge \neg Cand(x)$
IR.3	$\iota Ic1(x) \leftarrow Ic1^{nT}(x)$
IR.4	$\iota Ic2(x) \leftarrow Ic2^{nT}(x)$
IR.5	$\iota Ic3(x) \leftarrow Ic3^{nT}(x)$
IR.6	$\iota Ic \leftarrow \iota Ic1(x)$
IR.7	$\iota Ic \leftarrow \iota Ic2(x)$
IR.8	$\iota Ic \leftarrow \iota Ic3(x)$

3.4 Deletion internal events rules

Let P be a derived predicate. We can use definition (2) for a deletion internal event to generate its corresponding deletion internal events rules. In fact, we find convenient to draw various versions of such rules, versions with similar meanings, but different uses. From (2) we get:

$$\delta P(\mathbf{x}) \leftarrow P(\mathbf{x}) \wedge \neg P^n(\mathbf{x})$$

This is the *deletion internal events rule* of predicate P. This rule is used as such, without further transformations, in our process of transaction synthesis, particularly in the translation of views or derived predicates. However, we also need to use a more specialised version, although in this case for drawing compile-time repairs from ICs for transgressing base updates. We will now show the general pattern and intuitive meaning of such version, as well as its application to our example schema. We qualify this version of deletion rules with super-index 'Ic' according to their use.

There is a specialised deletion internal event rule for each definition P_i of P and for each literal L_j in such definition. Its general pattern is:

$\delta P^{Ic}(\mathbf{x}) \leftarrow L_1 \wedge ... \wedge L_{j-1} \wedge [\delta Q_j(\mathbf{x}_j) \mid \iota Q_j(\mathbf{x}_j)] \wedge L_{j+1} \wedge ... \wedge L_q \wedge \alpha$

where the first option is taken if $L_j = Q_j(\mathbf{x}_j)$ is positive and the second if negative.

Its tail α is defined as follows:

$\alpha = [\ \neg P^{nO}{}_1(\mathbf{x}) \wedge \neg P^{nT}{}_1(\mathbf{x}) \wedge ... \wedge \neg P^{nO}{}_{i-1}(\mathbf{x}) \wedge \neg P^{nT}{}_{i-1}(\mathbf{x}) \wedge$
$\wedge \neg P^{nO}{}_{i+1}(\mathbf{x}) \wedge \neg P^{nT}{}_{i+1}(\mathbf{x}) \wedge ... \wedge \neg P^{nO}{}_m(\mathbf{x}) \wedge \neg P^{nT}{}_m(\mathbf{x})$
$\mid \neg P^n(\mathbf{x})\ ]$

where the first option is taken when P_i does not have any existential variable and the second otherwise (i.e. for existential rule P_i).

In summary, the deletion internal events rules for our example database schema are the ones shown in Fig. 3-3.

Fig. 3-3

Code	Deletion internal events rule
DR.1	$\delta Rr(x) \leftarrow Rr(x) \wedge \neg Rr^n(x)$
DR.2	$\delta Rr^{Ic}(x) \leftarrow \delta Ra(x) \wedge \neg Cr(x) \wedge \neg Rr^{nO}{}_2(x) \wedge \neg Rr^{nT}{}_2(x)$
DR.3	$\delta Rr^{Ic}(x) \leftarrow Ra(x) \wedge \iota Cr(x) \wedge \neg Rr^{nO}{}_2(x) \wedge \neg Rr^{nT}{}_2(x)$
DR.4	$\delta Rr^{Ic}(x) \leftarrow \delta Cit(x) \wedge \neg Rr^{nO}{}_1(x) \wedge \neg Rr^{nT}{}_1(x)$
DR.5	$\delta Cand(x) \leftarrow Cand(x) \wedge \neg Cand^n(x)$
DR.6	$\delta Cand^{Ic}(x) \leftarrow \delta Int(x,y) \wedge Eco(y) \wedge \neg Cand^n(x)$
DR.7	$\delta Cand^{Ic}(x) \leftarrow Int(x,y) \wedge \delta Eco(y) \wedge \neg Cand^n(x)$

Note again the intuitive meaning of rules in Fig. 3-3. For example, DR.2 defines that the right of residence of 'x' is deleted during a transition ($\delta Rr(x)$) if 'x' is deleted as a registered alien ($\delta Ra(x)$), and 'x' did not have a criminal record in the old state ($\neg Cr(x)$), and 'x' does not have that right in the new state according to $Rr^n{}_2$ (given by "$\neg Rr^{nO}{}_2(x) \wedge \neg Rr^{nT}{}_2(x)$"). Similarly, DR.7 states that 'x' is deleted as a job candidate if s/he had an interview with company 'y' in the old state (Int(x,y)), and 'y' is deleted as employer company during the transition ($\delta Eco(y)$), and in no other way 'x' remains as candidate in the new state ($\neg Cand^n(x)$).

3.5 The augmented database schema

Let DBS be a database schema. We call *augmented database schema*, or A(DBS), the database schema consisting of DBS, its transition rules and its internal events rules. In the next section we will discuss the important role of A(DBS) in our method for update transaction synthesis. The augmented database schema for our example would be the union of the contents of the above Figs. 2-1, 2-2, 2-3, 3-1, 3-2 and 3-3. It is easy to show that, because DBS is allowed, then A(DBS) is also allowed.

4. SYNTHESIS OF TRANSACTIONS

We envision a transaction-design-support-system that builds minimal and meaningful database update transactions, from the corresponding design-time parameterised *transaction requests* (Tr). A Tr represents the designer's intents about the effects of the transaction, i.e. the transaction post-conditions.

After formally defining transaction requests, we will address our approach to the synthesis of consistency-preserving database update transactions. For presentation purposes, we prefer to postpone the full formalisation of our method to the Appendix, while providing within the main body of the report the description of the method together with the detailed explanation of several representative examples.

4.1 Transaction requests

Formally, a parameterised update transaction request Tr consists basically of either [P^n(**p**)] or [not P^n(**p**)], where P can be a base or a derived predicate, and **p** is a vector of terms. Usually, terms will mostly be parameters (i.e. 'Per', 'Age') but some could also be constants (i.e. 'john', '32'). However, it should be clear that these constants are initially provided by the designer at "transaction-design-time" because they are meaningful for his particular transaction request. At "transaction-processing-time", actual values for parameters will be given by the end-user when using the transaction.

As examples, two of the transaction requests that we will later elaborate on are [Emp^n(Per)] and [$Cand^n$(Per)]. With the first one the designer wants a consistency-preserving transaction such that after executed for a concrete person, to be provided in 'Per', we can guaranty that s/he is an employee; in other words, that inserts the person as employee, if necessary. In the case of [$Cand^n$(Per)], our method will synthesise a transaction for assigning the job candidate status to a particular person if s/he did not hold it, so that after its execution the person will be in the Cand view. Of course, this view updating must also preserve the consistency of the database.

Transaction requests may vary from the above pattern for special purposes. For example, in order to preserve database consistency [$\neg$ιIc] is used as a special "consistency-requirement" request. Also, a transaction request could include further negative base or derived events, such as in [Rr^n(Per),$\neg$ιCit(Per)], where their role is that of "selective-requirements"; in the given case the designer wants a transaction to include someone as right resident but not through the granting of citizenship.

4.2 Our approach

We now focus on the problem of the automatic generation at design-time of consistency-preserving transactions from transaction requests in the context of the deductive databases described in section 2.

Stated more precisely, the problem is: Given an initial transaction request, which reflects the transaction designer's updating intents, and considering the database schema, obtain a minimal and meaningful transaction able to perform, at run-time, those updating intents without violating database

consistency. In order to realise this purpose, we have designed and implemented a method that can be briefly described and exemplified as follows.

The transaction request posed by the designer together with the A(DBS) implicitly configure a generic search space that we explore through two types of compile-time derivations: *Translate* and *Repair* derivations. From the interleaving of those derivations we draw an interim tree, the Trek_tree. This tree sometimes needs to be optimised in various ways; redundant nodes and unuseful or unsuccessful branches must be pruned away. Using the resulting enhanced tree, the designer may further choose, out of all the valid updating alternatives considered in it, those options most interesting for his/her application. However, s/he can also rely on the run-time transaction processing system or the end-user to take some or all of these decisions.

Then, a simple in-order search of the remaining tree is the base for the layout of the final transaction text, or Trek_text, in whatever appropriate transaction language syntax we choose. For the moment we have English and Catalan pseudo-code, as well as directly executable Prolog code, but other languages can easily be added. The labels in the nodes of the tree are interpreted and treated according to their implied semantics and the language chosen; this guides the inclusion of the appropriate keywords in the text, as well as the correct composition of condition conjunctions and disjunctions. For ease of comprehension, an indentation mechanism presents the Trek_text as will be shown in the following examples.

From the above description, note that the starting step and core phase of our method is that represented by Translate and Repair derivations. We give here only an intuitive idea of such derivations. Their formal definitions are included in the Appendix, right after the References. Translate and Repair derivations traverse in single steps the generic search space implicitly defined by the Tr and the A(DBS). A Translate derivation is used to obtain a "translation" from the original Tr, that is, a transaction that will accomplish the designer's intents. However, for such translation to be consistency-preserving, consistency needs to be enforced with regard to some conditions, such as the schema ICs and other particular transaction requirements either initially given by the designer or drawn from the A(DBS) while doing the Translate derivation.

Repair derivations are in charge of enforcing such external and internal consistency conditions. A Repair derivation represents a subsidiary derivation spawning from a Translate derivation. Repair derivations maintain, check and use the "Consistency conditions set" C, an internally maintained set of conditions that we do not want any transaction to possibly imply at update-time, i.e that we want any transaction to avoid. C is the source of all possible repairs or branch invalidations in our interim tree. Repair derivations can further call other translate derivations in order to obtain the translations for their found compensating actions.

With regard to our assumed run-time environment, we consider delayed-update semantics for transaction-processing-time. That is, conditions within transaction programs always refer to the old database state, while proposed base updates are to be collected and finally committed as a whole and all at once to the database. Thus, there is no need to keep and/or to query any intermediate state.

4.3 Consistency conditions for our example database schema

Let's see what consistency conditions should be initially included in the set C for our example database schema. Those conditions are the ones emerging from the schema ICs, which will have to be checked for any transaction update that we generate. This initialisation of C is attained in our method through a Translate derivation from the single request [¬ιIc]. This derivation immediately calls for a Repair derivation, which guaranties failure for any possibly successful derivation in the search space implicitly defined by {←ιIc} and the A(DBS). This will not generate any transaction text, but will include into C any checking and enforcement conditions coming from the ICs.

The Repair derivation from {←ιIc} keeps resolving its goals until they include at least one base event. This will later on permit to select the conditions affected by a base update in order to extract from them either repairs or branch invalidations.

To see how this realises in our example database schema, recall from Fig. 3-1 and Fig. 3-2 the transition and internal events rules for the three ICs of our example. We may resolve these rules with the ones of δRr^{Ic} (DR.2, DR.3 and DR.4) and of ιCand (IR.2) in order to obtain the initial set C_0 of our particular example, which is shown in Fig. 4-2. C.1 to C.5 correspond to Ic1, C.6 to C.8 come directly from Ic2, and C.9 to C.13 correspond to Ic3. However, note that all conditions consider literal "¬ιAbort". For example, the intuitive meaning of consistency condition C.4 is that the database will eventually go into an inconsistent state if we insert as employee someone without right of residence, pretend not to legalise such person, and if the overall action is not "aborted" by the transaction processing system.

Fig. 4-2

Code	Consistency condition
C.1	$\leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \delta Ra(x) \wedge \neg Cr(x) \wedge \neg Rr^{nO}{}_2(x) \wedge \neg Rr^{nT}{}_2(x) \wedge \neg \iota Abort$
C.2	$\leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge Ra(x) \wedge \iota Cr(x) \wedge \neg Rr^{nO}{}_2(x) \wedge \neg Rr^{nT}{}_2(x) \wedge \neg \iota Abort$
C.3	$\leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \delta Cit(x) \wedge \neg Rr^{nO}{}_1(x) \wedge \neg Rr^{nT}{}_1(x) \wedge \neg \iota Abort$
C.4	$\leftarrow \iota Emp(x) \wedge \neg Rr(x) \wedge \neg \iota Rr(x) \wedge \neg \iota Abort$
C.5	$\leftarrow \iota Emp(x) \wedge \delta Rr^{Ic}(x) \wedge \neg \iota Abort$
C.6	$\leftarrow Emp(x) \wedge \neg \delta Emp(x) \wedge \iota App(x) \wedge \neg \iota Abort$
C.7	$\leftarrow \iota Emp(x) \wedge App(x) \wedge \neg \delta App(x) \wedge \neg \iota Abort$
C.8	$\leftarrow \iota Emp(x) \wedge \iota App(x) \wedge \neg \iota Abort$
C.9	$\leftarrow Cand(x) \wedge \neg \delta Cand(x) \wedge \delta App(x) \wedge \neg \iota Abort$
C.10	$\leftarrow Int(x,y) \wedge \neg \delta Int(x,y) \wedge \iota Eco(y) \wedge \neg Cand(x) \wedge \neg App(x) \wedge \neg \iota App(x) \wedge \neg \iota Abort$
C.11	$\leftarrow \iota Int(x,y) \wedge Eco(y) \wedge \neg \delta Eco(y) \wedge \neg Cand(x) \wedge \neg App(x) \wedge \neg \iota App(x) \wedge \neg \iota Abort$
C.12	$\leftarrow \iota Int(x,y) \wedge \iota Eco(y) \wedge \neg Cand(x) \wedge \neg App(x) \wedge \neg \iota App(x) \wedge \neg \iota Abort$
C.13	$\leftarrow \iota Cand(x) \wedge \delta App(x) \wedge \neg \iota Abort$

4.4 Transaction synthesis examples

The following pages include several synthesis examples illustrating the method. In order to facilitate the comprehension of each example, we prefer to first show and comment on the resulting synthesised transaction program, and then to present the detailed explanation of the reasoning process behind such synthesis. We will show the synthesis (output and process) from transaction requests [Emp^n(Per)] and [$Cand^n$(Per)]. For incremental presentation purposes, we will make a selective use of the various ICs of our database example.

4.4.1 Example synthesis from [Emp^n(Per)]

We could synthesise directly from [Emp^n(Per)] while considering all of our ICs, but prefer to show the same result in a step-by-step fashion. In this way, [Emp^n(Per) wrt. Ic1] will permit us to concentrate on the synthesis output and process in the case of not having to deal with existential rules. On the other side, the synthesis from [Emp^n(Per) wrt. Ic2,Ic3] will show some of the problems and proposed solutions that appear when dealing with such existential rules. Other existential issues will appear when synthesising from [Cand^n(Per)].

4.4.1.1 Synthesis output from [Emp^n(Per) wrt. Ic1]

Considering only Ic1, if a designer requests a transaction for adding someone as employee in the database, our method generates the corresponding Trek_text contained in Fig. 4-1. Note the slightly different syntax used for the various predicate types, which comes directly from our implementation of the method in Prolog. The only differences are that base and derived predicates must begin with a lower-case letter, that the super-index '^{n}' qualifying new predicates is implemented with prefix 'n_', and that meta-level update operators 'ι' and 'δ' are also handled as prefixes 'i_' and 'd_', respectively. Horizontal and vertical lines have been added for ease of reading. This implementation syntax will be followed for all synthesised program text.

Fig. 4-1

```
    trek_text([n_emp(Per)],  % wrt. Ic1(x)

1   ------------ if emp(Per) then
2   -------------|--- i_exit
3   -------------|- else
4   -------------|--- i_emp(Per) ,
5   -------------|----- if not rr(Per) then
6   -------------|------|- either
7   -------------|------|--|- { i_rr(Per) }
8   -------------|------|--|--- either
9   -------------|------|--|----|--- if ra(Per) then
10  -------------|------|--|----|----|--- d_cr(Per)
11  -------------|------|--|----|----|---|- else
12  -------------|------|--|----|----|--- if not cr(Per) then
13  -------------|------|--|----|----|----|--- i_ra(Per)
14  -------------|------|--|----|----|----|- else
15  -------------|------|--|----|----|----|--- i_ra(Per) ,
16  -------------|------|--|----|----|----|--- d_cr(Per)
17  -------------|------|--|----|----|--- end_if
18  -------------|------|--|----|--- end_if
19  -------------|------|--|----|- or
20  -------------|------|--|----|--- i_cit(Per)
21  -------------|------|--|--- end_either
22  -------------|------|--|- or
23  -------------|------|--|--- i_abort
24  -------------|------|- end_either
25  -------------|----- end_if
26  ------------ end_if

        ). % end of trek text
```

Within Fig. 4-1, line 1 controls if the person to be employed is already an employee, in which case line 2 exits the transaction without any updating. In general, the special event "tExit" is used to exit its nesting compound instruction but keeping any update so far proposed; in this example, however, no update has been proposed before such instruction. If the person under consideration is not an employee, line 4 proposes to insert him/her as such. However, in this case, our integrity constraint Ic1 is directly affected by such base update, and a checking/enforcement preventive repair is needed. The repair notices that, if we want to insert someone as employee (line 4) without a right of residence status (line 5), then there are only two alternatives not to violate database consistency: either to grant right of residence to the person (lines 7 to 21) or to abort the whole transaction (line 23).

For the alternative of granting right of residence, we initially draw the proposal that tRr(x) should be pursued, which is shown in line 7 as a commented action preceding its further development. Later on, our method translates such view-update request into the needed base update instructions for granting right of residence (lines 8 to 21). The second alternative has been drawn from the special base event "tAbort" (line 23) which, during the execution of the transaction, will backout whatever updates have been proposed so far. Aborting the transaction, however conservative, might be a useful option in some cases.

In fact, lines 5 to 21 would be the main body of the transaction synthesised from the [Rr^n(Per)] request. Observe that for this view-update transaction, there are also two alternatives, corresponding to the two schema definitions for Rr(x). That is, we can give the right of residence to a person either by making him a non-criminal registered alien (lines 9 to 18) or by granting citizenship to such individual (line 20). For the first option, we proceed depending upon the old database state, which can be precisely determined using the run-time user-provided value for parameter 'Per', thus the conditional if-then-else instruction proposed.

Besides entering values for parameters, some further intervention will usually be expected from the user at run-time. For example, if the transaction in Fig. 4-1 were provided to him/her as it is (i.e. without any prior pruning decisions made by a transaction designer), then s/he would hold all choice responsibly when confronted with 'either' control instructions. However, there could be other alternative choice strategies, such as leaving some of these decisions for the transaction processing system to take. It could use priority criteria either provided by the transaction designer from application domain semantics or inferred from extensional database statistics, or a combination of both. So far, this is an interesting open problem that we have not addressed yet.

4.4.1.2 Synthesis process from [Emp^n(Per) wrt. Ic1]

Figures 4-2.1, 4-2.2 and 4-2.3 below show a detailed step-by-step explanation of the implicit reasoning behind the Translate and Repair derivations involved in this example. There you can see the use of the A(DBS) and of the consistency set C to draw the needed (view-)update translations and repairs. The "WHAT" column shows the goal under consideration, which comes either from the designer (the original request) or from a rule or condition of the A(DBS) or the set C, referenced in the "Where from" column. The treatment of a goal may lead to its further resolving through the rules indicated in "Where next", that also signals when some new internal consistency condition must be added to the set C (i.e. "$\rightarrow\Delta$C"). Finally, "Reasoning script" verbally explains the various steps

leading to the text written in the "HOW" column, where only the bold-typed text corresponds to the final transaction instructions.

Fig. 4-2.1 shows the reasoning needed to obtain the starting TRANSLATION_1 for our original transaction request. However, since its proposed update affects consistency conditions in C, it calls for a repair from those conditions. The reasoning leading to that REPAIR_1 is presented in Fig. 4-2.2, where we can see that only one of the two relevant conditions needs in fact to be avoided. Thus, we draw the repair from it, but find out that such repair calls for a further view-update TRANSLATION_2, which is shown in Fig. 4-2.3. This final translation ends the synthesis process, since none of its proposed base updates affects the so far maintained set C, i.e. they do not need to be further repaired.

Fig. 4-2.1 (TRANSLATION_1)

Where from	WHAT	Reasoning script	Where next	HOW
Designer	Give me a trck for employing a person, i.e. accomplish $Emp^{n}(Per)$	Just a moment!	TR.33 TR.34	TRANSLATION_1=
TR.33	$Emp^{nO}(Per)$	Exit if already in the old state, i.e.	TR.35	
TR.35	Emp(Per)∧ ¬δEmp(Per)	exit if Emp(Per)		**if emp(Per) then** **i_exit**
	¬δEmp(Per)	but remember not to delete Emp(Per)	→ΔC	
TR.34	$Emp^{nT}(Per)$	otherwise, we'll do it in the transition, i.e.	TR.36	**else**
TR.36	ιEmp(Per)	insert Emp(Per), but		**i_emp(Per)** , REPAIR_1? (See Fig. 4-2.2, later Fig. 4-4.1)
C?		see that it affects consistency, i.e. set C, and repair appropriately		
				end_if

Fig. 4-2.2 (REPAIR_1 wrt. Ic1)

Where from	WHAT	Reasoning script	Where next	HOW
C?	Avoid any Ic1-related consistency condition affected by ιEmp(Per), i.e.	Let's see! Oh, yes; there are two such conditions	C.4 C.5	REPAIR_1=
C.4	repair from ¬Rr(Per)∧ ¬ιRr(Per)∧ ¬ιAbort	for the first one, check if ¬Rr(Per) and, if so,		**if not rr(Per) then**
	¬ιRr(Per)∧ ¬ιAbort	alternatively accomplish the insertion of Rr(Per)		**either** **{ i_rr(Per) }** TRANSLATION_2? (See Fig. 4-2.3)
	¬ιAbort	or abort the whole transaction;		**or** **i_abort**
				end_either
				end_if
C.5	repair from δRr(Per)∧ ¬ιAbort	the second condition is in fact no problem since we have not proposed to delete Rr(per)		

Fig. 4-2.3 (TRANSLATION_2)

Where from	WHAT	Reasoning script	Where next	HOW
	Accomplish ιRr(Per)	Of course!	IR.1	TRANSLATION_2=
IR.1	Rr^{nT}(Per)∧ ¬Rr(Per)	We are already assuming that ¬Rr(Per), so we won't propose to check it again;		
	Rr^{nT}(Per)	in the transition you may insert Rr(Per) in two possible ways: W1 or W2	TR.3 TR.4	**either**
TR.3	Rr^{nT_1}(Per)	(W1) through its first definition, that is	TR.8 TR.9 TR.10	
TR.8	Ra(Per)∧ ¬δRa(Per)∧ δCr(Per)	depending upon the old state, check if Ra(Per) and, since thus Cr(Per), then		**if ra(Per) then**
	¬δRa(Per)∧ δCr(Per)	as long as you remember not to delete Ra(Per),	→ΔC	
	δCr(Per)	you can delete Cr(Per)		**d_cr(Per)**
C?		-it does not affect C-		REPAIR_2=∅
TR.9	ιRa(Per)∧ ¬Cr(Per) ∧¬ιCr(Per)	otherwise, if ¬Cr(Per), and thus ¬Ra(Per), then		**else if not cr(Per) then**
	ιRa(Per)∧ ¬ιCr(Per)	remember not to insert Cr(Per), and	→ΔC	
	ιRa(Per)	you can insert Ra(Per)		**i_ra(Per)**
C?		-it neither affects C-		REPAIR_3=∅
TR.10	ιRa(Per)∧ δCr(Per)	otherwise, i.e. when Ra(Per) and Cr(Per)		**else**
	ιRa(Per)∧ δCr(Per)	then insert Ra(Per)		**i_ra(Per) ,**
C?		-not affecting C-		REPAIR_4=∅
	δCr(Per)	and then delete Cr(Per)		**d_cr(Per)**
C?		-not relevant to C, neither alone nor with the previous update-		REPAIR_5=∅
				end_if
				end_if
TR.4	Rr^{nT_2}(Per)	(W2) but we can also insert Rr(Per) through its second definition, that is	TR.12	**or**
TR.12	ιCit(Per)	since for sure ¬Cit(Per), we may insert Cit(Per)		**i_cit(Per)**
C?		-which does not affect C-		REPAIR_6=∅
		All right, we made it!		**end_either**

The previous example already shows how we synthesise transaction text from a base update request, which in turn requires a corresponding integrity checking/enforcement repair, which itself further needs some view-update translation code. That is, it exemplifies how we address in an integrative way the main problems presented in the introduction of this report. However, it does not show neither the intermediate trees used to do the synthesis nor the formal rules used to build and trim such derivation trees. See [Pas94a] and the Appendix for more details.

4.4.1.3 Synthesis output from [Emp^n(Per) wrt. Ic2,Ic3]

If a designer issues the same [Emp^n(Per)] request to our system, but this time considering only Ic2 and Ic3, the method will generate the Trek_text contained in Fig. 4-3. Lines 1 through 4 work as in the previous example. This time the insertion of employee directly affects Ic2, so a checking/enforcement preventive repair is needed. The repair notices that, if we want to insert as employee (line 4) some applicant (line 5), then database consistency must be preserved either by deleting such person as applicant (line 7) or by aborting the transaction (line 23). The deletion of applicant would furtherly affect Ic3, thus reclaiming the corresponding repair. That is, in case that such not-to-be-applicant were also a candidate (line 8) either it should be deleted as such (lines 10 to 17) or an abort should be proposed (line 19).

Fig. 4-3

```
    trek_text([n_emp(Per)],  % wrt. Ic2(x) and Ic3(x)

1   ---------- if emp(Per) then
2   ----------|--- i_exit
3   ----------|- else
4   ----------|--- i_emp(Per) ,
5   ----------|----- if app(Per) then
6   ----------|-----|- either
7   ----------|-----|--|--- d_app(Per) ,
8   ----------|-----|--|------ if cand(Per) then
9   ----------|-----|--|-------|--- either
10  ----------|-----|--|-------|----|--{ d_cand(Per) }
11  ----------|-----|--|-------|----|----foreach [_Comp] in int(Per, _Comp) and eco(_Comp) do
12  ----------|-----|--|-------|----|----|--- either
13  ----------|-----|--|-------|----|----|--- |--- d_int(Per, _Comp)
14  ----------|-----|--|-------|----|----|--- |- or
15  ----------|-----|--|-------|----|----|--- |--- d_eco(_Comp)
16  ----------|-----|--|-------|----|----|--- end_either
17  ----------|-----|--|-------|----|----end_foreach
18  ----------|-----|--|-------|----|- or
19  ----------|-----|--|-------|----|--- i_abort
20  ----------|-----|--|-------|--- end_either
21  ----------|-----|--|------ end_if
22  ----------|-----|--|- or
23  ----------|-----|--|--- i_abort
24  ----------|-----|- end_either
25  ----------|----- end_if
26  ---------- end_if

        ). % end of trek text
```

For the alternative of deleting the person as candidate, we initially draw the proposal that δCand(x) should be pursued, shown in line 10 as a commented action preceding its unfolding. Later on, our method translates such view-update request into the needed base update instructions (lines 11 to 17).

Line 8 together with lines 10 to 17 in fact correspond to the main body of the transaction that would be synthesised from the [¬Cand^n(Per)] request. This is a view-update transaction request for deleting the extension of an existentially defined view predicate. To accomplish such objective, we should eliminate any existing way in which the contents of the database support our view extension, for which we will now need to take into account the values taken by the existential variables in the

definition(s) of the view predicate. In our example, this is obtained with the 'foreach-in-do' instruction of lines 11 to 17. For this instruction we automatically synthesise the needed meaningful Skolem variable names (i.e. '_Comp'), depending upon the existential variables under consideration. Line 11 walks through the set of all employer companies with whom the person in 'Per' has an arranged job interview, thus setting the cursor variable '_Comp' appropriately. For each such company, lines 12 to 16 offer to either delete the pending interview or delete the employer status for the company. In this way, 'Per' will no longer remain a job candidate since s/he will not have any more interviews with employer companies, although s/he could still keep some interviews with non-employers.

Regarding run-time user interaction in this case, nothing new has appeared with respect to the previous example. The user must enter values for parameters, and take any remaining either-choice decision. He needs not provide or select any values for the Skolem variables of the 'foreach-in-do' instruction, since such values are precisely obtained from the database contents.

Let us note before going on with the synthesis process for this example, that the complete output that would have been synthesised from [Emp^n(Per)], taking into account all ICs at once, can be obtained by inserting lines 5 to 25 of Fig. 4-3 between lines 25 and 26 of Fig. 4-1.

4.4.1.4 Synthesis process from [Emp^n(Per) wrt. Ic2,Ic3]

Note that this example shares with the previous one the first part of the synthesis process, i.e. TRANSLATION_1 included in Fig. 4-2.1, to which we refer the reader. The new figures 4-4.1, 4-4.2, 4-4.3 and 4-4.4 below explain the implicit reasoning behind the remaining Repair and Translate derivations involved in the synthesis of this example.

(TRANSLATE_1 wrt. Ic2,Ic3)

Same	**A S**	TRANSLATE_1	wrt.	**Ic1** (See Fig. 4-2.1)

Fig. 4-4.1 (REPAIR_1 wrt. Ic2,Ic3)

Where from	**WHAT**	Reasoning script	Where next	**HOW**
C?	Avoid any Ic2 or Ic3-related consistency condition affected by tEmp(Per), i.e.	Let's see now! Humm, yes; here we have one such condition, drawn from Ic2	C.7	REPAIR_1=
C.7	repair from App(Per)∧ ¬δApp(Per)∧ ¬tAbort	check if App(Per) and, if so,		**if app(Per) then**
	¬δApp(Per)∧ ¬tAbort	alternatively accomplish the deletion of App(Per),		**either** **d_app(Per)** ,
C?		but see that it bothers consistency from C, so repair appropriately		REPAIR_2'? (See Fig. 4-4.2)
	¬tAbort	or abort the whole transaction;		**or** **i_abort** **end_either**
				end_if

Fig. 4-4.2 (REPAIR_2')

Where from	**WHAT**	Reasoning script	Where next	**HOW**
C?	Avoid any Ic2 or Ic3-related consistency condition affected by δApp(Per), i.e.	Let's see again! Right here: only one such condition, related to Ic3	C.9	REPAIR_2'=
C.9	repair from Cand(Per)∧ ¬δCand(Per)∧ ¬ιAbort	check if Cand(Per) and, if so,		**if cand(Per) then**
	¬δCand(Per)∧ ¬ιAbort	alternatively accomplish the deletion of Cand(Per),		**either** { **d_cand(Per)** } TRANSLATION_2'? (See Fig. 4-4.3)
	¬ιAbort	or abort the whole transaction;		**or** **i_abort**
				end_either
				end_if

Fig. 4-4.3 (TRANSLATION_2')

Where from	**WHAT**	Reasoning script	Where next	**HOW**
	Accomplish δCand(Per)	Right away!	DR.5	TRANSLATION_2'=
DR.5	Cand(Per)∧ ¬Cand^{n}(Per)	We are already assuming that Cand(Per), so won't propose to check it again,		
	¬Cand^{n}(Per)	and we must avoid Cand^{n}(Per)		REPAIR_3'? (See Fig. 4-4.4)

Fig. 4-4.4 (REPAIR_3')

Where from	WHAT	Reasoning script	Where next	HOW
	Avoid $Cand^{n}$(Per), i.e.	This is different! I will do it by avoiding all ways supporting Cand(per) in the new state, which requires A1 and A2	TR.14 TR.13	REPAIR_3'=
TR.14	first avoid $Cand^{nT}$(Per),	(A1): not to insert Cand(Per) in the transition, i.e. ... resolve a little bit...	TR.15	
TR.15	i.e. avoid $Cand^{nT}{}_{1}$(Per),	... and a bit more ... ... to find out that:	TR.18 TR.19 TR.20	
TR.18	i.e. repair from Int(Per,y)∧ ¬δInt(Per,y)∧ tEco(y)	nobody has issued an tEco(y), so this condition is already false, but we keep the whole goal, just in case someone tries later;	→ΔC	
TR.19	and repair from tInt(Per,y)∧ Eco(y)∧ ¬δEco(y)	neither tInt(Per,y) has been issued, so we remember the goal for future checks;	→ΔC	
TR.20	and repair from tInt(Per,y)∧ tEco(y)	It has been said ! so we keep remembering for later use	→ΔC	
TR.13	and then avoid $Cand^{nO}$(Per),	(A2): to delete all means by which Cand(Per) holds in the old state, i.e. ... resolve again ...	TR.16	
TR.16	i.e. avoid $Cand^{nO}{}_{1}$(Per),	... and once more ... ... to see that	TR.17	
TR.17	i.e. repair from Int(Per,y)∧ Eco(y)∧ ¬δInt(Per,y)∧ ¬δEco(y)	we need to find out the values of 'y' -call it "_Comp"- that make 'Per' a candidate, and for each of them		**foreach [_Comp] in int(Per,_Comp) and eco(_Comp) do**
	i.e. repair from ¬δInt(Per, _Comp) ∧ ¬δEco(_Comp)	avoid its support, i.e. either delete the interview,		**either d_int(Per,_Comp)** REPAIR_4'=∅ **or d_eco(_Comp)** REPAIR_5'=∅ **end_either**
C?		-which not affects C-		
	¬δEco(_Comp)	or remove the found employer as such;		
		-neither affects C-		
		Yes, yes! ...		
		We finally got it tied!!		**end_foreach**

The example has shown the synthesis of transaction text from a base update request, which in turn requires a corresponding integrity checking/enforcement text. This repair text again needs another consistency checking/enforcement repair, which finally requires some view-update translation code, this time generated from a combination of two translation and repair processes.

4.4.2 Example synthesis from [Cand^n(Per)]

This example deals with a view-update request to make some person 'Per' candidate after the execution of the resulting transaction. We will first show the synthesis output and process for [Cand^n(Per)] without considering any of our ICs. That is, in [Cand^n(Per) without ICs] we will concentrate on some more issues when synthesising from existential rules. Finally, the complete output for [Cand^n(Per) with ICs] will be given and commented; for space considerations, no synthesis process will be provided in this case.

4.4.2.1 Synthesis output from [Cand^n(Per) without ICs]

Fig. 4-5 contains the Trek_text for this view-update request. When 'Per' already has some interview with some employer (line 1), i.e. s/he is already a job candidate, line 2 exits the transaction. Otherwise, three alternatives exist: namely, to consider as employers some (at least one) of the companies with whom 'Per' has interviews, if any (lines 5 to 7); or to arrange an interview between 'Per' and some (one or more) of our already considered employer companies, if any (lines 9 to 11); or to ask the user for some (one at least) yet unknown companies in order to make them employers with interviews with 'Per' (lines 13 to 18).

Fig. 4-5

```
    trek_text([n_cand(Per)], % without ICs

1   ---------- if int(Per, _Comp) and eco(_Comp) then
2   ----------|--- i_exit
3   ----------|- else
4   ----------|--- either
5   ----------|---|--- forsome [_Comp] in int(Per, _Comp) do
6   ----------|---|---|- i_eco(_Comp)
7   ----------|---|--- end_forsome
8   ----------|---|- or
9   ----------|---|--- forsome [_Comp] in eco(_Comp) do
10  ----------|---|---|- i_int(Per,_Comp)
11  ----------|---|--- end_forsome
12  ----------|---|- or
13  ----------|---|--- forsome new [_Comp] such that
14  ----------|---|---|--- not int(Per, _Comp) and not eco(_Comp)
15  ----------|---|---|- do
16  ----------|---|---|--- i_int(Per, _Comp) ,
17  ----------|---|---|--- i_eco(_Comp)
18  ----------|---|--- end_forsome
19  ----------|--- end_either
20  ---------- end_if

          ). % end of trek text
```

The synthesised condition within line 14 can be used to help the user look for the right companies, or to help the system check for wrong user elections. Similarly, the conditions in lines 5 and 9 could be used to present the respectively satisfying companies to the user for him/her to select some.

The above combination of either-or with forsome-in-do's is highly non-deterministic. With the transaction as it is (i.e. without prior designer intervention), at run-time the user should also choose one or more either alternatives out of the relevant ones. While the last alternative may always be

relevant, the other two depend on the existence of values in the database satisfying their conditions. Note that the (three) relevant alternatives could be freely combined within one transaction execution, thus making 'Per' a candidate through various non-conflicting ways. A run-time update solution involving these multiple ways might not be minimal, but it could be meaningful, and thus useful. The lack of conflicts is given by the delayed-update semantics; recall that it guaranties that forsome-in-do and forsome-new-suchthat conditions are only affected by the old database state, and not by the proposed base updates, applied as a whole at transaction-finish.

The flexibility implied by the above instructions, which by the way will require a sophisticated run-time user-interaction-system, contrasts with the strict determinism imposed by the if-then-else structure used in the synthesis of [Rr^n(Per)] (Fig. 4-1, lines 9 to 18). Such big difference is due to the existential Skolem variables.

However, in the general case, such flexible user-interaction framework could sometimes prove too demanding for some types of user, or even inadequate for some types of applications (i.e. user-less applications, with update requests issued programmatically). For situations like these, our transactions could be synthesised under the selective guidance of a transaction-designer. S/he could use domain knowledge to purge alternatives and/or assign them priorities to be used by the transaction-processing-system. Evaluation cost-estimates could be used at design-time, such as the length or complexity of either-or alternatives, or types of forsome-in-do conditions (i.e. base vs. derived, simple vs. compound); as well as at run-time, such as database population statistics. The transaction-processing system, on its side, could also incorporate mechanisms to automatically select or invent condition values. There is plenty of further work along this line.

4.4.2.2 Synthesis process from [$Cand^n$(Per) without ICs]

Fig. 4-6 on next page explains the implicit reasoning to synthesise this example. Before going on with the final example synthesis from [$Cand^n$(Per) with ICs], let us make some comments on an interesting use of the consistency set C that will show up there.

Observe that following the synthesis of each forsome-in-do we include into the consistency set C the negation of the event predicate that would contradict its condition. This guaranties that any further synthesised code within each forsome-in-do is also checked for this "internal" consistency condition. Later on it will be shown how these two consistency conditions help specialise the respective repairs. While the first one ({←δInt(Per,_Comp)}) will restrict one repair alternative, the second one ({←δEco(_Comp)}) will fully remove another repair alternative.

There is no such internal consistency problem affecting our third translate alternative, i.e. the forsome-new-suchthat instruction. In fact, the semantics of such construct imply the inverse objective. That is, right after the forsome-new-suchthat condition is layed out, we want to synthesise precisely those updates that will contradict such condition, thus undoing the state of things described by it.

Fig. 4-6 (TRANSLATION_1)

Where from	WHAT	Reasoning script	Where next	HOW
Designer	Accomplish $Cand^{n}(Per)$	Yes sir/maam!	TR.13 TR.14	TRANSLATION_1=
TR.13	$Cand^{nO}(Per)$	See when it already holds,	TR.16	
TR.16	$Cand^{nO}{}_{1}(Per)$	which is	TR.17	
TR.17	i.e. repair from Int(Per,y)∧ Eco(y)∧ ¬δInt(Per,y)∧ ¬δEco(y)	if there is some value of 'y' -call it "_Comp"- that already makes 'Per' a candidate;		**if int(Per,_Comp) and eco(_Comp) then**
	¬δInt(Per,_Comp) ∧¬δEco(_Comp)	and since none of these base updates have been issued, we may propose to exit the transaction.		**i_exit**
TR.14	$Cand^{nT}(Per)$	Otherwise, we'll do it during the transition.	TR.15	**else**
TR.15	$Cand^{nT}{}_{1}(Per)$	Since we are dealing with an existential view-definition, we can use three combinable ways: W1 or W2 or W3	TR.18 TR.19 TR.20	**either**
TR.18	Int(Per,y)∧ ¬δInt(Per,y)∧ ιEco(y)	(W1): for some of 'Per's' interviewing companies in 'y' -call it "_Comp"- ,		**forsome [_Comp] in int(Per,_Comp) do**
	¬δInt(Per,_Comp) ∧ιEco(_Comp)	as long as we remember not to delete it/them later,	→ΔC	
	ιEco(_Comp)	make it/them employers		**i_eco(_Comp)**
C?		-C not affected-		REPAIR_1=∅ (R1)
				end_forsome
TR.19	ιInt(Per,y)∧ Eco(y)∧ ¬δEco(y)	(W2): for some of our employers in 'y' -call it "_Comp"- ,		**or forsome [_Comp] in eco(_Comp) do**
	ιInt(Per,_Comp)∧ ¬δEco(_Comp)	as long as we remember not to remove it/them,	→ΔC	
	ιInt(Per,_Comp)	set an interview with 'Per'		**i_int(Per,_Comp)**
C?		-C not triggered-		REPAIR_2=∅ (R2)
				end_forsome
TR.20	ιInt(Per,y)∧ ιEco(y)	(W1): for some newly-provided companies in 'y' -call it "_Comp"- such		**or forsome new [_Comp] such that**
		that all preconditions of the considered updates hold		**not int(Per,_Comp) and not eco(_Comp) do**
	ιInt(Per,_Comp)∧ ιEco(_Comp)	arrange an interview between 'Per' ...		**i_int(Per,_Comp) ,**
C?		-which does not affect C-		REPAIR_3=∅
	ιEco(_Comp)	... and the newly inserted ... employer.		**i_eco(_Comp)**
C?		-C not bothered neither by this last update alone, nor by both the last two ones, taken together-		REPAIR_4=∅ (R4)
		O.K., That's		**end_forsome**
		all		**end_either**
		folks!!!		**end_if**

4.4.2.3 Synthesis output from [Cand^n(Per) with ICs]

In the previous Fig. 4-6, the various repairs are empty because ICs were not considered. If we take all the ICs into account, the complete synthesised output corresponds to that shown across Fig. 4-7.1 and Fig. 4-7.2, with the above repairs R1, R2 and R4 indicated.

Fig. 4-7.1

```
trek_text([n_cand(Per)],  % with ICs
  '
   if int(Per, _Comp) and eco(_Comp) then
      i_exit
   else
     either
        forsome [_Comp] in int(Per, _Comp) do
         i_eco(_Comp).
|---------------
|            foreach [_Person] in int(_Person, _Comp) and
|                                        not app(_Person) do
|           either
|              d_int(_Person, _Comp) ,
|                if [_Person]=[Per] then
|                 i_abort
|                end_if
|           or
|              i_app(_Person) ,
|                if emp(_Person) then
R1               either
|                  d_emp(_Person)
|                or
|                  i_abort
|                end_either
|               end_if
|           or
|             i_abort
|           end_either
|         end_foreach
|---------------
       end_forsome
    or
       forsome [_Comp] in eco(_Comp) do
        i_int(Per, _Comp) ,
|---------------
|           if not app(Per) then
|            either
|              i_app(Per) ,
|                 if emp(Per) then
|                  either
|                   d_emp(Per)
R2                or
|                   i_abort
|                  end_either
|                end_if
|            or
|              i_abort
|            end_either
|          end_if
|---------------
       end_forsome
    or
       {continues on the right ->}
```

Fig. 4-7.2

```
{-> continued from the left}
    or
       forsome new [_Comp] such that
           not int(Per, _Comp) and not eco(_Comp)
        do
          i_int(Per, _Comp) ,
          i_eco(_Comp) ,
|---------------
|            foreach [_Person] in int(_Person, _Comp) and
|                                        not app(_Person) do
|             either
|               d_int(_Person, _Comp)
|             or
|               i_app(_Person) ,
|                 if emp(_Person) then
|                  either
|                    d_emp(_Person)
|                  or
|                    i_abort
|                  end_either
|                end_if
|             or
|               i_abort
|             end_either
|           end_foreach ,
R4
|           if not app(Per) then
|             either
|               i_app(Per) ,
|                 if emp(Per) then
|                  either
|                    d_emp(Per)
|                  or
|                    i_abort
|                  end_either
|                end_if
|             or
|               i_abort
|             end_either
|           end_if
|---------------
       end_forsome
    end_either
  end_if
  '
 ). % end of trek text
```

The three consistency-preserving repairs, all drawn from the same ICs, namely Ic2 and Ic3 (Ic1 is not affected by this request), have been automatically specialised for their respective situations.

REPAIR_1 (R1) follows the case where we convert to employers some companies with whom 'Per' had interviews. But for each one of these employers-to-be (in '_Comp'), there could very well be other arranged interviews with people, including 'Per'. At this point, as a side-effect, we are making these persons labour candidates all at once. This could affect Ic3 for those not previously being job applicants. R1, thus, offers three alternative ways to avoid this conflicting situation for each transgressing person ('_Person').

First, we may remove his/her interviews. Second, we can add him/her as applicant. And third, we could abort the whole transaction. The first alternative must be further conditioned, since such compensating action is valid for any one but precisely 'Per'. We are not interested in contradicting a condition (the one in the forsome-in-do) that we are taking for granted in the search for our initial view-update request. So if a user insists in removing our 'Per's' interview, we will abort instead, since such action is forbidden by our consistency condition set C. The second alternative (i.e. making applicants) also has to be extended with the appropriate repair drawn from Ic2, for the case of those persons that were already employees. Note that this could very well be the case of our initial 'Per', if s/he were not applicant; then, his/her name would also be a value for '_Person' and could be removed as employee. For this case, the second alternative does not pose any restrictions, as did the first.

We find a less complex situation in the case of REPAIR_2 (R2). Since it has been drawn for a new interview for 'Per' with an old employer, no side-effects appear. The only problematic candidate-to-be would be 'Per', if s/he was not applicant (and employee). However, it must be pointed out that our synthesis method has eliminated from the alternatives of R2 the one consisting in removing the companies generated by the previous forsome-in-do. Such alternatives would unconditionally contradict our original view-update request goal, because it would contradict its nesting forsome-in-do condition. Again, this is controlled through set C.

REPAIR_3 from Fig. 4-6 is really empty, so no trace of it remains in Fig. 4-7. No compensating action has been synthesised in this case because none of the conditions in C is relevant to the previous base update solely, i.e. the insertion of an interview to a non-employer company. However, such update does affect one such condition when issued together with the insertion of employer of such company. Thus, we obtain the corresponding REPAIR_4.

REPAIR_4 (R4) is in fact composed of two consecutive sub-repairs, sort of a combination of simpler versions of R1 and R2. Since the user/system is now providing completely new non-employer companies with whom 'Per' could not have had a pre-arranged interview, then 'Per' will not be contained in the set of persons generated by the foreach-in-do, so its first alternative does not need to be conditioned. For that same reason, however, its second alternative cannot take care of a conflicting 'Per', so such compensation must be done apart, as a different sub-repair. This last sub-repair coincides in output with R2 but its synthesis process is simpler, since it must not deal with the elimination of the request-contradicting alternative; at this point it would be nonsense to propose the deletion of a company as employer when we know that it is not. In fact, both repairs have been drawn from different instances of the consistency conditions set C: namely, R2 from C.11, and the last sub-repair from C.12. This has been our last example in this report (see [Pas94a] for other). Obvious space considerations prevent us from presenting its synthesis process.

5. ADDITIONAL FEATURES OF OUR METHOD

[Pas94a] also provides examples of additional features of our method that have not been shown in this report. These features include the representation of transition ICs and their use in transaction synthesis. Specialising transaction synthesis to accommodate particular selective requirements is possible, thus being able to implement some interesting update policies such as the prevention of side-effects to selected base or derived predicates. Also considered is the implicit handling of the modification operation through deletions and insertions. Evaluable predicates can also be used in our database schemes. The method can also handle compound transaction requests relating post-conditions affecting more than one base and/or derived predicate.

Our approach has proved useful not only in analysing and avoiding update inconsistencies but also in rectifying previously inconsistent databases, as introduced in [ML91]. For doing so, we drop the assumption on database consistency before an update and then request the deletion of database inconsistency through δIc, which makes use of TR.21, TR.25 and TR.29 in Fig. 3-1. From them we draw the appropriate consistency-repairing Treks.

We believe that it will not be very difficult to extend the method to deal with initial set-oriented qualified-update requests (for ex. remove as employee those citizens with criminal records, or convert to employers all companies with whom our applicants have interviews). This will surely be eased by the treatment of existential rules introduced in this report.

6. RELATIONSHIP WITH PREVIOUS METHODS

In this section, we briefly compare our method with some previous transaction-design-time methods.

Stemple and Sheard [SS89] offer a set of tools based on a Boyer-Moore theorem prover to support a transaction designer in coming up with non-violating transactions. A model of relational theory with recursive functions and additional convenient lemmata provide the basis for the deduction process within the prover. With this knowledge in hand, it tries to prove without accessing facts that a given transaction will not violate consistency. When that is not possible, those parts of the transaction and schema that caused the failure are identified and the system generates feedback to the transaction designer by suggesting new run-time consistency checks, additional updates that would make the transaction safe and post-conditions that reflect the designer's intent. As the reader may infer from this description, our intentions and procedure are similar to those of Stemple et al. However, we aim at the fully automatic synthesis of transactions in the more general case of deductive databases. In our approach, the designer does not have to hand out a (possibly erroneous) transaction to the system, but only give a request for it and, eventually, adapt the resulting transaction to his/her particular application requirements. Our transaction-design-time approach is preventive instead of corrective.

Ceri and Widom [CW90] present a semi-automatic approach with an SQL-based language for defining ICs and a framework for translating them into consistency-maintaining ECA-like production rules. Their method is semi-automatic because some parts of the translation require the designer intervention. S/he must specify for each IC, i.e. for each consistency-maintaining rule, a set of repair actions. Such actions may introduce cyclic, infinite rule execution and inefficiencies, which requires

additional rule analysis and optimisation steps. They plan to extend their facility to allow multiple rules enforcing one IC and the possibility of automatically or semi-automatically deriving compensating actions. It is interesting to observe that this method happens to run into the view-update problem when trying to translate ICs with "table expressions", a restricted form of views, but its solution is not explained in detail. Since general views are essential to deductive databases, we address them from the beginning. Note also that, due to the nature of our internal events rules, we get for free the multiple rules enforcing a single constraint and we can also automatically generate compensating actions (our preventive repairs), both goals aimed at Ceri and Widom's future extensions.

Some of these extensions have already been addressed in [CF+92, FPT92], but in restricted or adapted cases. [CF+92] extends the previous method by including some views and automatically generates repairs under some circumstances. For example, view predicates cannot appear in the body of ICs, and its ECA rules are prepared to handle only base updates. In [FPT92] the authors depart towards a more logical framework, showing how to automatically generate production rules from ICs specified in a restricted Domain Relational Calculus. Apart from the replacement operation, that we handle indirectly through deletion and insertion, our method deals so far with more general databases and with more general kinds of updates, i.e. view-updates, base updates with possibly additional transaction requirements, and compound updates. [CW90, CF+92, FPT92] share the common aim of an Integrity Maintenance System within an Active DBMS architecture, with its rule triggering system taking care of the application of their consistency-repairing ECA-rules or production rules. We take a more general approach, since our resulting transactions could be used (once adapted) in a varied set of DBMS architectures. With regard to execution semantics, they let violations occur and then try to restore consistency while accomplishing the initial user update; this sometimes leads to the undoing of some previous work, specially when aborting. Recall that our delayed-update semantics save us from such complex run-time situations. Our transaction-processing approach is also preventive instead of curative.

Still within the relational context, the results of Qian [Qia90,Qia94] in the area of automatic programming of database transactions are also of much interest, but only from the IC enforcement side. They aim at the synthesis of database transactions from the designer's updating intent and the ICs. For that purpose, the Manna-Waldinger deductive-tableau system is extended with additional inference rules for the extraction of valid transactions from proofs. Observe that our approach tries to do the same but handling also general views, not present in the work of Qian. For the common case, we share some language constructs, such as if-then-else and foreach-in-do. Qian's method lays out consistency-preserving code just before the possibly transgressing update, while we do the other way around. Delayed-update semantics make both textual layouts equivalent from the processing point of view.

[Wal91] investigates, in the context of deductive databases, the automatic compilation of ICs checking into update procedures written in a procedural update language, whose semantics is defined in detail. Through the application of partial evaluation and logical optimisation, a set of conditions are imposed as preconditions on the corresponding update procedure. However, it does not deal with view updates and/or ICs consistency-preserving actions.

6. CONCLUSIONS AND FURTHER WORK

The methods developed so far for View updating and ICs checking and enforcement in relational and deductive databases are a big step towards a practical solution to the overall problem of providing true knowledge independence within databases. They have open plenty of room within this research area.

In this report we have presented a new method for the generation of consistency-preserving transaction programs from (view-)updates in the context of deductive databases with existential rules. The method is based on the transition and internal events rules, which explicitly define the dynamic behaviour of the database when updated. Using these rules, a formal method allows us to automatically synthesise a legal transaction program from an initial (view-)update transaction request. The integrative way in which the method already deals at compile-time with the problems of View updating, IC checking and IC enforcement can be considered as its most important asset.

At its current stage, the method has been fully prototyped using meta-programming techniques in Prolog. With regard to transaction-processing-time capabilities, we can generate Trek_text which is directly a Prolog program used to simulate the updating of the database within the dynamic main-memory Prolog database. The program is essentially the transaction unit, once appropriately declared within the corresponding transaction boundaries. In the future, this will be the base for the update processing of real disk-based databases either within a tight-coupling architecture, i.e. on top of a deductive DBMS, for which case we do not expect big changes; or through a loose-coupling setting, where further interaction instructions will be needed. In this last respect, we find that some of our results may proof useful in the emerging contexts of extended relational databases, particularly in SQL-3, and of active databases, whose advanced ECA rules mechanisms could also be the target of our transaction synthesis. As it has been commented through the examples, the potential transaction-processing-system capabilities are so varied that they open a whole new line of future work.

We plan to extend the method for the case of deductive database schemes with aggregate functions within rules, where the appropriate formalisation and implementation will be needed. Also, the case of dealing with recursive rules must be studied in more detail. Another interesting improvement would be the treatment of initial qualified-update requests. We may also consider the explicit treatment of the modification as a database operation of its own, along the line taken by [UO92,MT93] for similar run-time problems.

ACKNOWLEDGEMENTS

The author would like to thank Antoni Olivé, who encouraged pursuance of this work, as well as the rest of the Odissea Group (Dolors Costal, Carme Martín, Enric Mayol, Carme Quer, Maria R. Sancho, Jaume Sistac, Ernest Teniente and Toni Urpí) for many useful comments and discussions on the theme of this report.

This work has been partially supported by the CICYT PRONTIC project TIC 680.

REFERENCES

[BMM90] Bry,F.;Manthey,R.;Martens,B. "Integrity verification in knowledge bases". ECRC report D.2.1.a, Munich, April 1990.

[CW90] Ceri, S.; Widom,J. "Deriving Production Rules for Constraint Maintenance". Proc. of 16th VLDB, Brisbane, Australia, 1990, pp. 566-577.

[CF+92] Ceri, S.; Fraternali, P.; Paraboschi, S.; Tanca, L. "Integrity Maintenance Systems: An Architecture". 3rd. Int. Workshop on the Deductive Approach to Information Systems and Databases, Roses, Catalonia 1992, pp. 327-344.

[FPT92] Fraternali, P.; Paraboschi, S.; Tanca, L. "Automatic Rule Generation for Constraint Enforcement in Active Databases". In *Modelling Database Dynamics* , Eds. U.W.Lipeck, B.Thalheim, Volkse, 1992, pp. 327-344.

[KSS87] Kowalsky, R.; Sadri, F.; Soper, P. "Integrity Checking in Deductive Databases". Proc. of the 13th VLDB, Brighton, 1987, pp. 61-70.

[ML91] Moerkotte,G.; Lockemann,P.C. "Reactive Consistency Control in Deductive Databases", in ACM TODS, Vol. 16, No. 4, December 1991, pp. 670-702.

[MT93] Mayol,E.; Teniente,E. "Incorporating Modification Requests in Updating Consistent Knowledge Databases", 4th. Int. Workshop on the Deductive Approach to Information Systems and Databases, Lloret, Catalonia 1993, pp. 335-360.

[Oli89] Olivé,A. "The internal events method for integrity checking in deductive databases". Internal research report, Dept. LSI, UPC, Barcelona, 1989.

[Oli91] Olivé, A. "Integrity constraints checking in deductive databases". Proc. of the 17th VLDB, Barcelona, 1991, pp. 513-523.

[OP90] Olivé,A.; Pastor, J.A. "The internal events method for integrity constraint enforcement in deductive databases", 2nd. Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, September 1990, pp. 139-168.

[Pas92] Pastor, J.A. "Deriving Consistency-preserving Transaction Specifications for (View-) Updates in Relational Databases", 3rd Int. Workshop on the Deductive Approach to Information Systems and Databases, Roses, Catalonia 1992, pp. 275-300.

[Pas94a] Pastor, J.A. "A Formal Method for the Synthesis of Update Transactions in Deductive Databases without Existential Rules", Internal research report LSI-94-22-R, Dept. LSI, UPC, Barcelona, 1994.

[Pas94b] Pastor, J.A. "Extending the Synthesis of Update Transaction Programs to handle Existential Rules in Deductive Databases", to appear in Proc. of the 5th Int. Workshop on the Deductive Approach to Information Systems and Databases, Catalonia 1994.

[PO94] Pastor, J.A.; Olivé,A. "An Approach to the Synthesis of Update Transactions in Deductive Databases", to appear in Proc. of the CISMOD'94 (Fifth International Conference on Information Systems and Management of Data), Madras, India, October 1994.

[Qia90] Quian,X. "Synthesising Database Transactions". Proc. of the 16th VLDB, Brisbane, Australia 1990, pp. 552-564.

[Qia93] Quian,X. "The Deductive Synthesis of Database Transactions". in ACM TODS, Vol. 18, No. 4, December 1993, pp. 626-677.

[SS89] Sheard,T.;Stemple,D. "Automatic Verification of Database Transaction Safety", in ACM TODS, Vol. 14, No. 3, September 1989, pp. 322-368.

[TO92] Teniente,E.; Olivé,A. "The Events Method for View Updating in Deductive Databases", Proc. of EDBT 92, Vienna, March 1992, pp. 245-260.

[Ten92] Teniente,E. "El mètode dels esdeveniments per a l'actualització de vistes en bases de dades deductives", PhD Thesis, Dept. of LSI, Universitat Politècnica de Catalunya, Barcelona, June 1992 (written in Catalan).

[UO92] Urpí, T.; Olivé, A. "A Method for Change Computation in Deductive Databases". Proc. of the 18th VLDB, Vancouver, 1992, pp. 225-237.

[Wal91] Wallace, M. "Compiling Integrity Checking into Update Procedures", 12th Int. Conf. on Artificial Intelligence, Sydney, Australia, August 1991, Vol. 2, pp. 24-30.

APPENDIX: Formalisation of *Tree

In this Appendix, we give a formal definition of *Tree, the first and core step of our method, whose application has been illustrated in the various examples of this report.

Translate and Repair derivations explore the generic search space implicitly defined by the posed (view-) update transaction request Tr together with the A(BDS), while building at the same time a special-purpose representation of such space in the form of a general n-ary tree, the Trek_tree. Each node in this tree has one label and zero, one or more children sub-trees. The different label types in the nodes show our interpretation of the various kinds of rules and literals found during the search, while the sub-trees spawning from a node represent alternative ways of satisfying the updating intents.

Before the description and formalisation of Translate and Repair derivations, we discuss the different kinds of predicates used in the A(DBS), now in view of their meaning and usefulness for the purpose of deriving Trek_trees. Predicates in the augmented database schema A(DBS) are of one of the types shown in the following Fig. A-1, which also includes examples for the predicate types and the resulting Trek_tree labels.

Fig. A-1

A(DBS) predicate type	Examples (F.O.L.)	Result.Trek-tree label ex.(Prolog)
old (state) base predicate	App(p),Int(p,c),Eco(c) Cit(p),Ra(p),Cr(p),...	precond(app(P)), fsome_cond(int(P,_C),[_C]), fsnew_cond(not eco(_C),[_C]), feach_cond(eco(_C),[_C]),...
old (state) derived predicate	Rr(p),Cand(p),Ic1(p),...	precond(rr(P)), fsome_cond(rr(_P),[_P]), fsnew_cond(not cand(_P),[_P]), feach_cond(cand(_P,[_P])),...
evaluable predicate	p1=John, age<=32,...	precond(P1=john),...
base event predicate	ιRa(p),δApp(p),ιInt(p,c),...	action(i_ra(P)), action(d_int(_P,C))
derived event predicate	ιRr(p),δCand(p),ιIc,...	comment(i_rr(P)) comment(d_cand(_P)),...
new (state) predicate	$Ra^{n}(p)$,$Rr^{n}(p)$,$Cand^{n}(p)$,...	
- new (from Old state)	$Ra^{nO}(p)$,$Cand^{nO}(p)$,...	
- exit-level	$Ra^{nO}{}_{1}(p)$,$Cand^{nO}{}_{1}(p)$,...	exit_struct
- sequence-level	$Rr^{nO}(p)$,...	sequence_struct
- new (from Transition)	$Ra^{nT}(p)$,$Rr^{nT}(p)$,...	
- either-level	$Rr^{nT}(p)$,$Cand^{nT}{}_{1}(p)$,...	either_struct
- conditional-level	$Rr^{nT}{}_{1}(p)$,...	cond_struct

Old base and old derived predicates represent "(pre-)conditions" that must be checked against the old state of the database prior to the application of accompanying base or derived event predicates. When they are not ground wrt. compile-time semantics (see later), they become the starting conditions for forsome-in-do, forsome-new-suchthat-do or foreach-in-do instructions. Base events represent the update "actions" that can be performed on base predicates of the database. Derived events represent the induced updates on derived predicates (implicitly) requested by the transaction designer and that

must be translated into further preconditions and actions. A derived event explicitly appears in the Trek_tree in form of "comment" preceding its translation.

New predicates conceptually represent the evaluation of their corresponding base or derived predicate on the updated database, that is they represent transaction "post-conditions". Besides their use as the designer intents in the initial update transaction request, they are only further used within the analysis made by translate and repair derivations. Although they do not come out directly in the Trek_tree, their implied meaning is included in the form of various labels. For example, some new predicates represent that their corresponding schema predicate evaluates to true because it remains untouched from the old state; these become "exit_struct" branches in the Trek_tree. Sometimes, one of this predicates indicates that their resolvents should be analysed as if they formed a sequence of transaction pieces, thus becoming "sequence_struct" labels. Other new predicates say how their corresponding schema predicate could evaluate to true after being updated during the state transition. This can be done through various alternative ways or conditionally depending on the old database state; thus, these new predicates appear in the nodes of a Trek_tree as "either_struct" and "cond_struct" labels with their possibly many alternatives becoming children sub-trees of the node. Some may become "either_struct" or "cond_struct" depending upon their defining rule type, i.e. existential or not.

When writing out the final transaction text from the tree, once enhanced, the above node labels are interpreted and treated according to their implied semantics and the language chosen, as shown in the examples of this report. Hopefully, the formal meaning of these and other special-purpose node types appearing in a Trek_tree will be clearer after the following description and formalisation of Translate and Repair derivations.

Translate and Repair derivations traverse in single steps the generic search space implicitly defined by the Tr and the A(DBS). Each step starts from input vertex (G_i C_i N_i) and finishes with output vertex (G_{i+1} C_{i+1} N_{i+1}). G_i is the goal under translation (repair), which once resolved by the step becomes G_{i+1}. C_i is a the set of special conditions used to repair or invalidate derivation sequences, and that (in repairs) can also be augmented to C_{i+1} by certain derivation steps. Finally, N_j represents a node of the Trek_tree under construction. Each derivation step extends the input node N_i with a further output node N_{i+1}. Each such Trek_tree node has a label and zero, one or more children sub-trees, to be built by future derivation steps.

A *Tanslate derivation* is a sequence from ("←Tr,¬tlc" { } TN_0) to either ([] C_n^m [Void, []]) or (G_n^m C_n^m [dead_end, []]), where the former case ends the analysis of one of possibly many alternative ways of translating the user request, and the later invalidates the branch as a translation source. The set of translate derivations starting with the same request partially overlap and thus can be merged into a *translate-derivations-tree*. Both this tree and single translate derivations are formally described below.

For a transaction program to be consistency-preserving, consistency needs to be enforced with regard to ICs and with regard to particular transaction requirements either initially given by the designer or drawn from the A(DBS) while doing the derivation. This is accomplished through *Repair derivations*, which represent subsidiary derivations spawning from translate derivations.

Each *Repair derivation* starts with (F_0 C_0 RN_0) and ends with either ({} C_n [Void, []]) or (F_n C_n [dead_end, []]). While the first case shows that one or more repairs have been drawn for a potentially transgressing update action, if necessary; the last case says that no possible (clean) repair could be found and, thus, indirectly invalidates (conditions) the original translate derivation as an alternative translation source. For doing all this, repair derivations maintain, check and use the "conditions to avoid" set C_i. This set is the source of all possible repairs or branch (conditioned) invalidations, and for that purpose it contains any condition drawn from the ICs and from particular external or internally implied transaction requirements. Repair derivations can further call other translate derivations in order to obtain the translations for their found repairs. Repair derivations are also formally described in the following paragraphs.

As can be understood from the above descriptions, it is when building and analysing the set of translate and repair derivations that the mentioned Trek_tree is built. Its nodes are identified by the above TN_is and RN_is. TN_is identify nodes originated in a translate derivation and RN_is identify those nodes drawn from repair derivations. All nodes take the general form of "[Node_label, Node_children]". While Node_children is a (possibly empty) list of other node identifiers, Node_label takes one out of the various existing labels, including "Void" when no particular label has been added, or "dead_end" for (conditionally) invalidated branches.

For the formal definition of translate-derivations-trees and repair derivations, we use a few conventions that hopefully will be self-explanatory. However, the following meta-predicates must be described:

tg(L) qualifies literal L as compile-time "trek-ground", that is having as terms either input parameters (i.e. 'Per'), input constants (i.e. 'john'), or internally generated Skolem constants (i.e. '_Comp'). The last ones are to be interpreted as run-time "Skolem variables", name with which we have referred to them in the examples. Otherwise, when literal L has some variable (i.e. 'y'), it is considered as non-trek-ground. This terminology should not be confused with the subsequently necessary grounding of transaction parameters and Skolem variables with the run-time actual values provided by the user or the database extension. Input constants are purposely given by the designer to be considered in our compile-time synthesis, i.e. to specialise a particular synthesis to concrete values. Evaluable predicates will always be trek-ground, by the "allowed" property of our schemes and our literal selection-function default order.

sk(L,G,SL,SG,SV) respectively returns in SL and SG the result of skolemising goal {L∧G} wrt. its existentially quantified variables, whose corresponding compile-time "Skolem variables" are also returned in SV. Instead of inventing non-sense names for these, we draw meaningful names (i.e. '_Comp', '_Person') from internal domain meta-clauses.

hpoc(E,PoC,SV) returns for event E the lists of its parameters or constants (PoC) and corresponding Skolem variables (SV), if it has; otherwise it returns false.

pc(E) returns the pre-condition corresponding to event E, thus pc(ιP)=¬P and pc("δP")=P.

Translate-derivations-tree and translate derivation descriptions

A *translate-derivations-tree* is a tree with root, intermediate nodes and ending leaves of the form (G_i C_i TN_i). Such tree subsumes a set of (partially overlapped) linear derivations built via a safe selection rule ST with priority. Each such *translate derivation* is a sequence:
(G_1 C_1 TN_1), (G_2 C_2 TN_2), ..., (G_n C_n TN_n)
such that for each $i \geq 1$, G_i has the form $\{\leftarrow L_1, ..., L_k\}$, $ST(G_i)=L_j$ selects literal L_j from goal G_i according to the order of appearance of the following rules T1 to T5, and (G_{i+1} C_{i+1} TN_{i+1}) is obtained through the application of the chosen rule.

T1) If L_j is a positive or negative old base, old derived or evaluable predicate then $C_{i+1}=C_i$, and then if
T1.1) $tg(L_j)$ then $G_{i+1}=G_i \backslash L_j$, $TN_{i+1}=[precond(L_j),[TN_{i+2}]]$,
T1.2) otherwise if $G_i=\{L_j \wedge rG_i\}$ then $sk(L_j,rG_i,SL_j,G_{i+1},SV)$ and $TN_{i+1}=[fsome\_cond(SL_j,SV),[TN_{i+2}]]$.

T2) If L_j is a negative base or derived event or negative new predicate then $G_{i+1}=G_i \backslash L_j$, $TN_{i+1}=[Void, [RN^{()}]]$, where $RN^{()}$ is obtained by the *repair derivation* from $(\{\neg L_j\}\ C_i\ RN^{()})$ to either $(\{\}\ C'\ [Void, []])$ or $(\{\}\ C'\ [dead\_end, []])$, and then $C_{i+1}=C'$.

T3) If L_j is a positive new predicate then $G_{i+1}=G_i \backslash L_j$, $C_{i+1}=C_i$, and if
T3.0) $\{\leftarrow L_j\} \in C_i$, then $TN_{i+1}=[Void, Void]$;
else if $S=\{R_1,R_2,...R_m\}$ is the set of all resolvents of clauses in the A(DBS) with G_i on L_j with corresponding *translate-derivation-trees* respectively rooted at
$(\{\leftarrow R_1\}\ C_i\ TN_{i+2,1})$, ..., $(\{\leftarrow R_m\}\ C_i\ TN_{i+2,m})$, then if
T3.1) $exit\_level(L_j)$ then $TN_{i+1}=[exit\_struct, [TN_{i+2,1}...TN_{i+2,m}]]$, else if
T3.2) $either\_level(L_j)$ then $TN_{i+1}=[either\_struct, [TN_{i+2,1}...TN_{i+2,m}]]$, else if
T3.3) $conditional\_level(L_j)$ then $TN_{i+1}=[cond\_struct, [TN_{i+2,1}...TN_{i+2,m}]]$,
T3.4) otherwise $TN_{i+1}=[Void, [TN_{i+2,1}]]$, since m=1 by the A(DBS).

T4) If L_j is a positive base event then if
T4.1) $tg(L_j)$ then $G_{i+1}=G_i \backslash L_j$, $TN_{i+1}=[action(L_j), [repair\_struct, [RN^{()}]]]$,
T4.2) otherwise if $G_i=\{L_j \wedge rG_i\}$ then $sk(L_j,rG_i,SL_j,G_{i+1},SV)$ and $TN_{i+1}=[fsnew\_cond(pe(SL_j),SV), [action(L_j), [repair\_struct, [RN^{()}]]]]$;
where $RN^{()}$ is obtained by the *repair derivation* from $(RC_i\ C_i\ RN^{()})$ to either $(\{\}\ C'\ [Void, []])$ or $(\{\}\ C'\ [dead\_end, []])$, and then $C_{i+1}=C'$. RC_i is the subset of C_i relevant to L_j in the sense that each of its members includes L_j.

T5) If L_j is a positive derived event such that R_1 is the resolvent of the corresponding unique clause in the A(DBS) with G_i on L_j, then $G_{i+1}=R_1$, $C_{i+1}=C_i$, $TN_{i+1}=[comment(L_j), [TN_{i+2}]]$.

Repair derivation description

A *repair derivation* from (F_1 C_1 RN_1) to (F_n C_n RN_n) via a safe selection rule SR with priority, is a sequence:
(F_1 C_1 RN_1), (F_2 C_2 RN_2), ..., (F_n C_n RN_n)
such that for each $i \geq 1$, F_i has the form $\{\leftarrow L_1, ..., L_k\} \cup F'_i$, $SR(F_i)=L_j$ selects literal L_j from goal F_i according to the order of appearance of the following rules R1 to R5, and (F_{i+1} C_{i+1} RN_{i+1}) is obtained through the application of the chosen rule.

R1) If L_j is a positive base event such as if

R1.1) [action(L_j), _] $\in$ ancestors(RN_i) (repair needed) then if

R1.1.1) k=1 and $\neg$hpoc(L_j,_,_) -no repair - then $F_{i+1}=\{\}$, RN_{i+1}=[dead_end, []]
R1.1.2) k=1 and hpoc(L_j,PoC,SV) -conditioned dead_end - then
$F_{i+1}=\{\}$, RN_{i+1}=[cond_struct,[precond([SV]=[PoC]),[dead_end,[]]]]
R1.1.3) k>1 -repair plausible- then
$F_{i+1}=\{\leftarrow L_1, ..., L_k\} \setminus L_j \cup F'_i$, $C_{i+1}=C_i$, RN_{i+1}=[Void, [RN_{i+2}]].

R1.2) [action(L_j), _] $\notin$ ancestors(N_i) then
$F_{i+1}=F'_i$, $C_{i+1}=C_i \cup \{\leftarrow L_1, ..., L_k\}$, RN_{i+1}=[Void, [RN_{i+2}]].

R2) If L_j is a positive derived event, then
$F_{i+1}=S' \cup F'_i$, $C_{i+1}=C_i$, RN_{i+1}=[Void, [RN_{i+2}]], where S' is the set of all resolvents of clauses of the A(DBS) with $\{\leftarrow L_1, ..., L_k\}$ on L_j.

R3) If L_j is a positive new predicate, then if
$S'=\{R_1, R_2, ... R_m\}$ is the set of all resolvents of clauses in the A(DBS) with $\{\leftarrow L_1, ..., L_k\}$ on L_j with corresponding *repair derivations* respectively rooted at ($\{\leftarrow R_1\}$ C_i $RN_{i+2,1}$), ..., ($\{\leftarrow R_m\}$ C_i $RN_{i+2,m}$), then if
R3.1) sequence_level(L_j) then $F_{i+1}=F'_i$, $C_{i+1}=C_i$, RN_{i+1}=[sequence_struct, [$RN_{i+2,1}$...$RN_{i+2,m}$]],
R3.2) otherwise $F_{i+1}=S' \cup F'_i$, $C_{i+1}=C_i \cup \{\leftarrow L_j\}$, RN_{i+1}=[Void, [RN_{i+2}]].

R4) If L_j is a positive or negative old base, old derived or evaluable predicate then
$C_{i+1}=C_i$, and then if
R4.1) tg(L_j) then $F_{i+1}=\{\leftarrow L_1, ..., L_k\} \setminus L_j \cup F'_i$, RN_{i+1}=[precond(L_j), [RN_{i+2}]],
R4.2) otherwise if $F_i=\{L_j \wedge rF_i\}$ then sk(L_j,rF_i,SL_j,F_{i+1},SV) and
RN_{i+1}=[feach_cond(SL_j,SV),[RN_{i+2}]].

R5) If L_j is an negative exit-level(L_j) new predicate with corresponding *translate-derivations-tree* rooted at ($\{\leftarrow \neg L_j\}$ C_i TN_{i+2}^j), then
$F_{i+1}=F'_i$, $C_{i+1}=C_i$, RN_{i+1}=[cond_struct, [TN_{i+2}^j]|[RN_{i+2}]].

R6) When $\{\leftarrow L_1, ..., L_k\}$ is solely composed of other negative new predicates or negative base or derived events with corresponding *translate-derivations-trees* respectively rooted at ($\{\leftarrow \neg L_1\}$ C_i TN_{i+2}^1), ..., ($\{\leftarrow \neg L_k\}$ C_i TN_{i+2}^k), then
$F_{i+1}=F'_i$, $C_{i+1}=C_i$, RN_{i+1}=[either_struct, [TN_{i+2}^1...TN_{i+2}^k]].

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Research Reports – 1994

LSI–94–1–R "Logspace and logtime leaf languages", Birgit Jenner, Pierre McKenzie, and Denis Thérien.

LSI–94–2–R "Degrees and reducibilities of easy tally sets", Montserrat Hermo.

LSI–94–3–R "Isothetic polyhedra and monotone boolean formulae", Robert Juan-Arinyo.

LSI–94–4–R "Una modelización de la incompletitud en los programas" (written in Spanish), Javier Pérez Campo.

LSI–94–5–R "A multiple shooting vectorial algorithm for progressive radiosity", Blanca Garcia and Xavier Pueyo.

LSI–94–6–R "Construction of the Face Octree model", Núria Pla-Garcia.

LSI–94–7–R "On the expected depth of boolean circuits", Josep Díaz, María J. Serna, Paul Spirakis, and Jacobo Torán.

LSI–94–8–R "A transformation scheme for double recursion", José L. Balcázar.

LSI–94–9–R "On architectures for federated DB systems", Fèlix Saltor, Benet Campderrich, and Manuel García-Solaco.

LSI–94–10–R "Relative knowledge and belief: SKL preferred model frames", Matías Alvarado.

LSI–94–11–R "A top-down design of a parallel dictionary using skip lists", Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer.

LSI–94–12–R "Analysis of an optimized search algorithm for skip lists", Peter Kirschenhofer, Conrado Martínez, and Helmut Prodinger.

LSI–94–13–R "Bases de dades bitemporals" (written in Catalan), Carme Martín and Jaume Sistac.

LSI–94–14–R "A volume visualization algorithm using a coherent extended weight matrix", Daniela Tost, Anna Puig, and Isabel Navazo.

LSI–94–15–R "Deriving transaction specifications from deductive conceptual models of information systems", María Ribera Sancho and Antoni Olivé.

LSI–94–16–R "Some remarks on the approximability of graph layout problems", Josep Díaz, María J. Serna, and Paul Spirakis.

LSI–94–17–R "SAREL: An assistance system for writing software specifications in natural language", Núria Castell and Àngels Hernández.

LSI–94–18–R "Medición del factor modificabilidad en el proyecto LESD" (written in Spanish), Núria Castell and Olga Slávkova

LSI–94–19–R "Algorismes paral·lels SIMD d'extracció de models de fronteres a partir d'arbres octals no exactes" (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI–94–20–R "Una paral·lelització SIMD de la conversió d'objectes codificats segons el model de fronteres al mdel d'octrees clàssics" (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI–94–21–R "Clausal proof nets and discontinuity", Glyn Morrill.

LSI–94–22–R "A formal method for the synthesis of update transactions in deductive databases without existential rules", Joan A. Pastor.

LSI–94–23–R "On the sparse set conjecture for sets with low density", Harry Buhrman and Montserrat Hermo.

LSI–94–24–R "On infinite sequences (almost) as easy as π", José L. Balcázar, Ricard Gavaldà, and Montserrat Hermo.

LSI–94–25–R "Updating knowledge bases while maintaining their consistency", Ernest Teniente and Antoni Olivé.

LSI–94–26–R "Structural facilitation and structural inhibition", Glyn Morrill.

LSI–94–27–R "Formalising existential rule treatment in the automatic synthesis of update transactions in deductive databases", Joan A. Pastor.

Copies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es