Supporting Transaction Design in
Conceptual Modelling of Information Systems

Joan A. Pastor-Collado
Antoni Olivé

Report LSI-95-11-R

UPC
Facultat d'Informàtica
de Barcelona - Biblioteca
13 Jun. 1995
Supporting Transaction Design in Conceptual Modelling of Information Systems
(Extended Version)

Joan A. Pastor-Collado
Antoni Olivé
Universitat Politècnica de Catalunya
Dept. de LSI - Facultat d'Informàtica
Pau Gargallo, 5
08028 Barcelona, Catalonia
{patorlolive}@lsi.upc.es

ABSTRACT

A method and a tool for supporting transaction design in conceptual modelling of information systems is presented. The method derives automatically a transaction specification that integrates in a uniform manner the updating of base and derived information and the checking and maintenance of integrity within an information base conceptual schema. Transaction specifications thus obtained achieve their intended purpose and guarantee that information base consistency will be preserved. When there are several possible solutions, the method derives all of them. The designer may then intervene in various ways in order to select the most appropriate ones. From this choice on, the transaction processing system and the end-user can also play an interesting role in the final application of the transaction specification, for this one can be directly executable. Using a declarative, logic-base approach, the method is general, and can be adapted easily to most conceptual modelling methodologies.
1. INTRODUCTION AND PREVIOUS WORK

We present here a method and a tool that we have developed for supporting transaction design in conceptual modelling of information systems.

Transaction design is one of the key activities in most current information systems development methodologies. In essence, transaction design has as input the conceptual schema of the information base, including a set of integrity constraints (ICs) that must be satisfied, and the expected result (or intended effect) of a given transaction. From this input, the designer's job consists in specifying, in some language, a set of preconditions and a sequence of operations such that, if the preconditions are satisfied, the sequence of operations will produce the expected result, while leaving the information base consistent [CA+94].

It is not difficult to see that in presence of a complex conceptual schema, possibly considering deduced as well as base information, and a large set of ICs, transaction design may be an error-prone activity. On the other hand, transaction specifications are very sensitive with regard to schema changes in deductive laws and integrity constraints: addition, removal or modification of a deductive law or a constraint may invalidate a given transaction specification.

Despite its importance and difficulty, transaction design support has not received the same level of attention as other activities in conceptual modelling. In most methodologies, the task of deriving the preconditions from the ICs is entirely manual, without a supporting tool. The same happens to the task of deriving the appropriate sequence of operations. As an example, [CFT91] presents an information system design expert tool that enforces a modularisation methodology where the designer is confronted with questions relevant to the preservation of consistency when defining update operations, but the designer must somehow ensure manually that transaction execution preserves consistency. Also, in [SO94] we presented, in the context of temporal deductive conceptual models, a method for deriving transactions that included consistency checking preconditions. These were derived from a single base ground update, integrity maintenance was not addressed, and updating derived information did not make sense in such context.

The situation is somewhat different in the database field. For example, [SS89] offers a set of tools based on theorem proving to support a transaction designer in coming up with non-violating transactions. With knowledge from the database schema in hand, they try 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 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. Along the same line, [Qia93] presents a method for the automatic synthesis of database transactions from the designer's updating intent and the ICs. For that purpose, a deductive-tableau theorem proving system is extended with additional inference rules for the extraction of valid transactions from proofs. Finally, [Wal91] investigates, in the context of deductive databases, the automatic compilation of ICs checking into update procedures written in a procedural update language. Through the application of partial evaluation and logical optimisation, a set of conditions are imposed as preconditions on the corresponding update procedure. These methods, however, are automatic up to an integrity checking
point. They neither consider automation when updating derived information nor when maintaining database integrity through additional consistency preserving actions.

In this paper, we describe a method that can be used to derive automatically a transaction specification, or *Trek* (Transaction enforcing -view and integrity- knowledge), that integrates in a uniform manner the updating of base and derived information and the checking and maintenance of integrity within an information base conceptual schema. The method is an extension and an adaptation of our previous work in the context of transaction synthesis for relational and deductive databases [Pas92,PO94]. We now propose new transaction constructs in order to deal with more powerful schemes, and regard the output of our synthesis more as transaction specifications to be further refined by a transaction designer. The method is general, and can be adapted easily to most conceptual modelling methodologies. We use a declarative, logic-base language for the definition of conceptual schemas, in the manner of [CHF92]. Transaction specifications obtained with our method achieve their intended purpose and guarantee that information base consistency will be preserved. Sometimes, there are several possible solutions and the method derives all of them. However, the designer may intervene in various ways in order to select the most appropriate ones. From this choice on, the transaction processing system and the end-user can also play an interesting role in the final application of the transaction specification, for this one can be directly executable. To our fair knowledge, no previous approach within the conceptual modelling area offers our level of transaction support.

The paper is organised as follows. Next section defines the information base schemes currently accepted by our method and introduces the example that will be used throughout the paper. Section 3 reviews the components of the augmented information base schema, a key concept for the method. Section 4 illustrates our method for generating consistency-preserving transaction specifications through a detailed example and gives its formalisation. In section 5 we comment on how the method can be used to furtherly support transaction design with some additional examples. Additional features of the method are sketched in section 6. Finally, in section 7 we present our conclusions and comment on future work.

2. INFORMATION BASE CONCEPTUAL SCHEMES

We define here the kind of information base schemes treated in this paper. We want to be general, and therefore we use a simple formalism, easily adaptable to any conceptual modelling language. An information base (conceptual) schema IBS consists of three finite sets: a set $B$ of base predicates, a set $D$ of derived predicates with their deductive rules, and a set $I$ of integrity constraints (ICs). Base predicates are the schemes of the facts explicitly stored in the information base, which form the so-called extensional information base. Derived predicates are schemes representing information that is not stored in the information base but can be derived using deductive rules. ICs are used to specify unwanted information base states and forbidden state transitions.

Before providing more formal definitions for some of the previous concepts, let us introduce the base predicate schemas corresponding to the information base example that we will be using throughout the paper. They are shown in Fig. 2-1 on next page, together with their intended meaning. Our example, inspired upon the one in [Qia93], is an information base for an "Employment Office" that
arranges labour interviews between its registered job applicants and some employer companies collaborating with it. For the people administered by the office, it also keeps tract of those already employed.

<table>
<thead>
<tr>
<th>Base predicate</th>
<th>Base predicate meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>App(x)</td>
<td>'x' is a job applicant</td>
</tr>
<tr>
<td>Eco(y)</td>
<td>'y' is an employer company</td>
</tr>
<tr>
<td>Int(x,y)</td>
<td>'x' has an interview with 'y'</td>
</tr>
<tr>
<td>Emp(x)</td>
<td>'x' is an employee</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Derived predicate + deductive rules</th>
<th>Derived predicate meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cand(x) ← Int(x,y) ∧ Eco(y)</td>
<td>'x' is considered a job candidate when s/he has an interview with an employer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Integrity rule</th>
<th>Integrity constraint meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ic1 ← Emp(x) ∧ App(x)</td>
<td>Nobody can be both employee and applicant</td>
</tr>
<tr>
<td>Ic2 ← Cand(x) ∧ ¬ App(x)</td>
<td>Candidates must be applicants</td>
</tr>
</tbody>
</table>

2.1 Deductive rules

Formally, a deductive rule is a formula of the form: $A \leftarrow L_1 \land \ldots \land L_n$ with $n \geq 1$
where $A$ is an atom denoting the conclusion or derived predicate, and the $L_1, \ldots, L_n$ 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 information base. Any variables in $A$, $L_1, \ldots, L_n$ 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. Variables in the body of a rule not appearing in its head are called the "local variables" of such rule. As usual, we require that the schema is allowed, that is any variable that occurs in a deductive rule has an occurrence in at least one of its positive conditions. For the purposes of our presentation, we need just to include in our example a single derived predicate for defining job candidates, as shown in Fig. 2-2. However, there could be many different derived predicates, and more than one rule per derived predicate, with or without local variables.

2.2 Integrity constraints

Integrity constraints (ICs) are conditions that the information base is required to satisfy at all times. ICs are either state (or static), when they must be satisfied in any state of the information base, or dynamic, when they involve the evolution between two or more of its states. Dynamic ICs compelling only one transition between two successive states are further called transition ICs. Our method works with state and transition ICs. More general dynamic ICs require more sophisticated representation languages that we do not provide here. However, examples of such constraints may be found in our previous work on the related area of deductive conceptual models of information systems [Oli89,SO94].
Formally, an IC is a closed first-order formula that the information base is required to satisfy. We deal with constraints that have the form of a denial:

\[ \leftarrow L_1 \land \ldots \land L_n \quad \text{with } n \geq 1 \]

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 $\text{Icn}$, thus taking the same form as deductive rules. We call them integrity rules. We will use in our example the two state ICs shown in Fig. 2-3 above. The set of employees is disjoint with the set of applicants (Ic1), which is a superset of candidates (Ic2). Note that Ic2 is furtherly defined in terms of Cand, which is a derived predicate.

3. THE AUGMENTED INFORMATION BASE 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 information base schema in order to later on synthesise transaction specifications from them.

Conceptually, internal events, transition rules and internal events rules are meta-level constructs describing the dynamic behaviour of a information base 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 information base extension. However, the resulting rules depend only on the information base schema. They are independent from the base facts stored, and from any particular update. Their implied dynamic update behaviour is not represented by the schema solely.

In section 4, we will discuss the use of transition and internal events rules 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 $\text{IB}$ be a information base, $U$ an update and $\text{IB}^n$ the "new" updated information base. We say that $U$ induces a transition from $\text{IB}$ (the current state) to $\text{IB}^n$ (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 $\text{IB}^n$. Formally, we associate to each base, derived or inconsistency predicate $P$ an insertion internal events predicate $tP$ and a deletion internal events predicate $\delta P$, defined as:

1. \[ \forall x(tP(x) \leftrightarrow P^n(x) \land \neg P(x)) \]
2. \[ \forall x(\delta P(x) \leftrightarrow P(x) \land \neg P^n(x)) \]

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

3. \[ \forall x(P^n(x) \leftrightarrow \neg P(x) \land \neg tP(x)) \]
4. \[ \forall x(\neg P^n(x) \leftrightarrow \neg P(x) \land \neg tP(x)) \]
If \( P \) is a base predicate, then \( \tau P \) facts and \( \delta P \) facts respectively represent insertions and deletions of base facts, i.e. base updates. They will represent derived updates if \( P \) is a derived predicate.

If \( P \) is an inconsistency predicate (i.e. \( \lambda c \)), then \( \lambda c \) facts that occur during the transition will correspond to violations of its corresponding IC. Note that, for an inconsistency predicate \( IC \), \( \delta IC \) facts cannot happen in any transition, since we assume that the information base is consistent before the update and, thus, \( IC \) is always false. Two special-purpose system events are also used, '\( \lambda \)Abort' and '\( \lambda \)Exit', but their meaning will be clear with the examples of sections 4 and 5.

3.2 Transition rules

Let us take a base, derived or inconsistency predicate \( P \) of the information base. 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:

\[
P(x) \leftarrow P_i(x) \quad i = 1, \ldots, m
\]

Consider now one of the rules \( P_i(x) \leftarrow L_1 \land \ldots \land L_q \). When the rule is to be evaluated in the updated state its form is \( P^n_i(x) \leftarrow L^n_1 \land \ldots \land L^n_q \). 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, 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:

\[
(5) \quad P^n_{i,j}(x) \leftarrow \bigwedge_{r=1}^{q} [O(L_{r,j}) \lor T(L_{r,j})] \quad \text{for } j = 1, \ldots, 2^q
\]

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

\[
O(L_j) = \left( Q_j(x_j) \land \neg \delta Q_j(x_j) \right) \quad \text{if } L_j = Q_j(x_j)
\]

\[
= \left( \neg Q_j(x_j) \land \neg \tau Q_j(x_j) \right) \quad \text{if } L_j = \neg Q_j(x_j)
\]

and

\[
T(L_j) = \left( \tau Q_j(x_j) \right) \quad \text{if } L_j = Q_j(x_j)
\]

\[
= \left( \delta Q_j(x_j) \right) \quad \text{if } L_j = \neg Q_j(x_j)
\]

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:

\[
P^n_{i,1}(x) \leftarrow O(L_1) \land \ldots \land O(L_q)
\]

and to refer to it through the rule:

\[
P^{O}_{n,i}(x) \leftarrow P^n_{i,1}(x)
\]

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

\[
P^n_{O}(x) \leftarrow P^{O}_{n,i}(x) \quad i = 1, \ldots, m
\]

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:

\[
P^n_{T,i}(x) \leftarrow P^n_{i,j}(x) \quad j = 1, \ldots, 2^q
\]
Again, considering all \( m \) rules defining \( P \) we get:

\[
P^n_i(x) \leftarrow P^{nT}_i(x) \quad i = 1, \ldots, m
\]

Finally, we may now refer to both \( P^nO \) and \( P^nT \) through:

\[
P^n_i(x) \leftarrow P^{nO}_i(x) \quad
\]

\[
P^n_i(x) \leftarrow P^{nT}_i(x)
\]

We call the above rules, i.e. rules with (possibly subindexed) 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 information base example are shown in Fig. 3-1.

<table>
<thead>
<tr>
<th>Code</th>
<th>Transition rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR.1</td>
<td>( \text{Cand}^{nO}_i(x) \leftarrow \text{Cand}^{nT}_i(x) )</td>
</tr>
<tr>
<td>TR.2</td>
<td>( \text{Cand}^{nT}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.3</td>
<td>( \text{Cand}^{nO}_i(x) \leftarrow \text{Cand}^{nT}_i(x) )</td>
</tr>
<tr>
<td>TR.4</td>
<td>( \text{Cand}^{nO}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.5</td>
<td>( \text{Cand}^{nT}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.6</td>
<td>( \text{Cand}^{nT}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.7</td>
<td>( \text{Cand}^{nT}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.8</td>
<td>( \text{Cand}^{nT}_i(x) \leftarrow \text{Int}(x,y) \wedge \neg \text{Int}(x,y) \wedge \text{Eco}(y) \wedge \neg \text{Eco}(y) )</td>
</tr>
<tr>
<td>TR.9</td>
<td>( \text{Emp}(x) \wedge \neg \text{Emp}(x) \wedge \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.10</td>
<td>( \text{Emp}(x) \wedge \neg \text{Emp}(x) \wedge \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.11</td>
<td>( \text{Emp}(x) \wedge \neg \text{Emp}(x) \wedge \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.12</td>
<td>( \text{Emp}(x) \wedge \neg \text{Emp}(x) \wedge \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.13</td>
<td>( \text{App}(x) \leftarrow \text{App}^{nO}(x) )</td>
</tr>
<tr>
<td>TR.14</td>
<td>( \text{App}(x) \leftarrow \text{App}^{nT}(x) )</td>
</tr>
<tr>
<td>TR.15</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.16</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.17</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.18</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.19</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.20</td>
<td>( \text{App}(x) \leftarrow \text{App}(x) \wedge \neg \text{App}(x) )</td>
</tr>
<tr>
<td>TR.21</td>
<td>( \text{Eco}(x) \leftarrow \ldots \text{Int}(x,y) \leftarrow \ldots \text{Emp}(x) )</td>
</tr>
</tbody>
</table>

Transition rules for derived predicate \( \text{Cand} \) (TR.1 to TR.8) and for ICs (TR.9 to TR.16) are listed before those for base predicates \( \text{App} \) (TR.17 to TR.20). Transition rules for base predicates \( \text{Eco}, \text{Int} \) and \( \text{Emp} \) are similar to those of \( \text{App} \), 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 \( P^{nT}_i \), 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.

Each of the rules in Fig. 3-1 has a clear intuitive meaning. Thus, for example, TR.6 states that 'x' is a candidate in the new state \( \text{Cand}^{nT}_i(x) \), if s/he had a programmed interview with 'y' in the old state \( \text{Int}(x,y) \) that has not been cancelled in the transition \( \neg \text{Int}(x,y) \), and 'y' has been inserted as employer company during the transition \( \text{Eco}(y) \).
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:

\[
(6) \quad tP(x) \leftarrow P^n(x) \land \neg P(x)
\]

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

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

\[
tP(x) \leftarrow P^nT(x) \land \neg P(x)
\]

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

\[
tlc \leftarrow tlc k \quad k = 1..r
\]

where \( r \) is the number of ICs in the database.

Fig. 3-2 shows the insertion internal events rules for the example, respectively corresponding to derived predicate Cand, to inconsistency predicates lC1 and lC2, and to database inconsistency.

<table>
<thead>
<tr>
<th>Code</th>
<th>Internal events rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR.1</td>
<td>tCand(x) \leftarrow Cand^nT(x) \land \neg Cand(x)</td>
</tr>
<tr>
<td>IR.2</td>
<td>tlc1 \leftarrow lC1^nT</td>
</tr>
<tr>
<td>IR.3</td>
<td>tlc2 \leftarrow lC2^nT</td>
</tr>
<tr>
<td>IR.4</td>
<td>tlc \leftarrow tlc1</td>
</tr>
<tr>
<td>IR.5</td>
<td>tlc \leftarrow tlc2</td>
</tr>
</tbody>
</table>

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. From (2) we get:

\[
\delta P(x) \leftarrow P(x) \land \neg P^n(x)
\]

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 derived predicates. A more specialised version of the above rule is sometimes needed for drawing compile-time repairs from ICs for transgressing base updates. However, since it is not needed for the next examples, we ommit its development here and refer the reader to [Pas94b]. In summary, the deletion internal events rule for our example schema is the one shown in Fig. 3-3.

<table>
<thead>
<tr>
<th>Code</th>
<th>Deletion internal events rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR.1</td>
<td>\delta Cand(x) \leftarrow Cand(x) \land \neg Cand^n(x)</td>
</tr>
</tbody>
</table>

Note again the intuitive meaning of the rule in Fig. 3-3. It states that 'x' is deleted as a job candidate if s/he was a candidate in the old state, but does not remain as such in the new state.
3.5 The augmented information base schema

Let IBS be a information base schema. We call *augmented information base schema*, or A(ibs), the schema consisting of IBS, its transition rules and its internal events rules. In the next section we will discuss the important role of A(ibs) in our method for update transaction specification synthesis. The augmented information base 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 IBS is allowed, then A(ibs) is also allowed.

4. SYNTHESIS OF TRANSACTION SPECIFICATIONS

We envision a transaction-design-support-system that builds minimal and meaningful information base update transactions specifications, from the corresponding design-time parameterised transaction requests.

After formally defining transaction requests, we will address our approach to the synthesis of consistency-preserving update transaction specifications. The various steps involved in such synthesis process will be described and illustrated with a small but complete example. We will then provide a full formalisation of the first and core step of the method. In Section 5 we will show through further longer and more complex examples how our synthesis method can be used to support transaction design in conceptual modelling.

4.1 Transaction requests

A transaction (specification) request (Tr) basically includes those transaction "postconditions requirements" posed by the designer, i.e. his/her intents about the effect of the expected transaction. Formally, a parameterised update transaction request Tr consists of either $[P^n(p)]$ or $[\text{not } P^n(p)]$ at least, where $P$ can be a base, a derived or an auxiliar predicate, and $p$ is a vector of terms. Usually, terms will mostly be parameters (i.e. 'Per', 'Comp') but some could also be constants (i.e. Joan, UPC). However, it should be clear that these constants are initially provided by the designer at transaction-design-time because they are meaningful for his/her particular transaction request. They are not not be confused with the actual values for parameters to be later on provided by the end-users.

The simplest case is that of $P$ being a base or a derived predicate. Then $Tr$ is a postcondition expressed in terms of one of the base or derived predicates of the information base schema. As examples, two of the transaction requests that we will later elaborate on are $[\text{App}^n(\text{Per})]$ and $[\neg \text{Cand}^n(\text{Per})]$. With the first one the designer wants a transaction specification to insert the person 'Per' as applicant. In the case of $[\neg \text{Cand}^n(\text{Per})]$, our method will synthesise a transaction specification for removing the job candidate status of a particular person if s/he had it. Note that this means a deletion from a derived predicate.

More complex is the case where Tr represents a compound postcondition affecting more than one base and/or derived predicate. For doing so, the designer must temporarily use an auxiliar (derived) predicate (i.e. $P$), different from any other in the information base schema, whose definition expresses the intended postcondition. The (auxiliar) augmented schema corresponding to the rules of
such predicate is generated on the fly, to be used in the synthesis of the pursued transaction specification, after which auxiliary predicate and schema can be forgotten. For example, \([\text{Aux1}^n(\text{Per})]\) with \(\text{Aux1}(x) \leftarrow \text{Emp}(x) \land \neg \text{App}(x)\) can be used to synthesise a transaction specification for doing whatever is needed so that 'Per' is an employee but not an applicant.

An interesting subcase of the above one occurs when the auxiliary predicate in the request is defined using local variables. With such a request the designer is in fact after a "set-oriented" update transaction specification. As happens with example \((\neg \text{Aux2}^n(\text{Per}))\), with \(\text{Aux2}(x) \leftarrow \text{Int}(x,y)\), where the expected effect of the resulting transaction is the deletion of every job interview planned for 'Per'. An auxiliary predicate could also be defined in terms of other auxiliary predicates so as to express even more complex postconditions.

Transaction requests may vary from the general pattern for special (customisation) purposes. For example, transactions can be specialised to particular initial states by including "precondition requirements", as in \([\text{Emp}^n(\text{Per}), \neg \text{Cand}(\text{Per})]\), where the designer asks for a transaction specification to insert as employee a non-candidate. Also, a transaction request could include further negative events that play the role of "transition-selective requirements", as in \([\text{Emp}^n(\text{Per}), \neg \delta \text{App}(\text{Per})]\), where the designer wants a transaction specification to include someone as employee as long as s/he is not deleted as applicant. These two examples will be addressed again in section 5.

In fact, any combination of the above types of requests is possible. Finally, in order to preserve database consistency \([\neg \text{tlc}]\) is used as a special "consistency requirement" request implicitly appended to every other transaction request.

### 4.2 Our approach

We now focus on the problem of the automatic generation at design-time of consistency-preserving transaction specifications from transaction requests in the context of the information bases 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 information base schema, obtain a minimal and meaningful transaction capable of performing those intends without violating consistency. In order to realise this purpose, we have designed and implemented a method that can be briefly described and exemplified as follows. To facilitate the comprehension of the method, we prefer to show first the result of its application to one example transaction request. We will then describe in more depth the steps involved in the generation of such result, while informally describing the synthesis process. And finally, a detailed formalisation of its kernel step will be provided.

#### 4.2.1 Synthesis output from \([\text{App}^n(\text{Per})]\)

Assume that a designer poses the request \([\text{App}^n(\text{Per})]\) in search of a transaction specification for adding someone as a job applicant. From this request and our example (augmented) information base schema, our method ultimately generates the corresponding transaction text (i.e. "trek_text") contained in Fig. 4-1 on next page. 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 't' and 'δ' are
also handled as prefixes 'i_' and 'd_', respectively. Horizontal and vertical lines have been added for ease of reading. This layout format will be also followed for the other example outputs in section 5.

| trek_text((n_app(Per)), |
| 1          if app(Per) then |
| 2          i_exit            |
| 3          else              |
| 4          i_app(Per)        |
| 5          if emp(Per) then  |
| 6          either            |
| 7          d_emp(Per)       |
| 8          or                |
| 9          i_abort           |
| 10         end_either        |
| 11         end_if            |
| 12         % end of trek text |

With regard to our assumed run-time environment in this and any other examples, we consider delayed-update semantics for transaction-processing-time. That is, conditions within transactions 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.

Within Fig. 4-1, line 1 controls if the person is already an applicant, in which case line 2 proposes to exit the transaction without any updating. In general, the special event 'i_exit' 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 applicant, 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/maintenance preventive repair can be offered. The repair notices that, if we want to insert as applicant (line 4) some employee (line 5), then there are only two alternatives not to violate database consistency: either to delete the person as employee (line 7) or to abort the whole transaction (line 9).

4.2.2 Synthesis process from [App^a(Per)]

The above used transaction request [App^a(Per)], together with the implicit consistency requirement [-1lc] and the A(IBS), implicitly configure a generic search space that we conveniently explore through two types of design-time derivations: Translate and Repair derivations. From the interleaving of those derivations we draw an interim tree, the trek_tree. The process is independent of any particular value that parameter 'Per' could take, thus the "generic" nature of the search space under exploration. For the case of our example, Fig. 4-2 on next page shows the generic search space of interest, in plain text type, together with the resulting trek_tree, in bold type.

A translate derivation is used to obtain a "translation" from the original transaction request. Box T1 in Fig. 4-2 includes the starting translate derivation rooted at the original request. Single translate
steps explore and resolve their input goals until none is left. Intuitively, App^n(Per) will succeed if it was already true in the old state (step 1, left branch), that is if App(Per) holds (step 3, left) and is not deleted during the transition (to be controlled in box \( R_1 \)). Alternatively, it will also succeed if added in the transition (step 1, right branch), i.e. if App(Per) is inserted (step 3, right). On their way, translate steps add new nodes to the trek_tree under construction, depending upon the semantics of their input goal and selected literal within such goal. Note how various new predicates in the example have resulted in different node types in the trek_tree (steps 1, 2 left, 2 right). Their concrete semantics as well as the formal meaning of circle-coded steps are left for section 5.

However, for the translation above 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(IBM) 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 representing situations that we want any transaction to avoid. C is the source of all possible repairs or branch invalidations in our interim tree. For efficiency considerations, C is initially filled with all consistency conditions implied by the special consistency request \([-\text{IC}]\), which is implicitly appended to every other transaction request. That way, any subsequent translation may use them without the need to contruct them anew. For a deeper discussion on consistency set C, we refer the reader to section 4.3, where the full initial contents of such set are shown and commented. For our current example, only condition C.1 is used from Fig. 4-4 there.

Back to our example, box \( R_1 \) in Fig. 4-2 includes the appropriate repair derivation for ensuring that App(Per) has not been deleted and, more important, that it will not be deleted later on; this is accomplished by including such internal consistency condition in set C. On the other hand, repair derivation in box \( R_2 \) follows the right branch in \( T_1 \), where the insertion of App(Per) was considered. This insertion affects one of our ICs, i.e. lcl, in the way shown in \( R_2 \). There, one of the consistency conditions previously drawn from the implicit request \([-\text{IC}]\) is relevant to the proposed insertion (step 1), particularly if 'Per' was already employee in the old state (step 2). Since we do not want such potential inconsistency to succeed, we may force its failure in either two ways (step 3): by deleting 'Per' as employee, or by aborting the whole transaction. This last option is always a possible one for any consistency condition such as C.1 and, since it is implicit, its originating literal is shown in brackets in box \( R_2 \). Both alternatives are respectively considered by the two translate derivations in boxes \( T_2 \) and \( T_3 \). This ends the derivation process, for ICs are not further affected.

In this way, repair derivations call other translate derivations in order to translate their found redressing actions. These actions may include base updates, such as \( \delta E m p(Per) \) in \( T_2 \), or the special 'tabort' event, like in \( T_3 \), cases where the translation is straightforward. But they may also include derived events, for which an appropriate translation in terms of base events must be found through the further exploration of the search space implied by their internal events rules from A(IBM).

Trek_trees such as the one in Fig. 4-2 usually need to be optimised in various ways; redundant and empty nodes as well as useless or unsuccessful branches must be pruned away. In our example, empty nodes and empty repair structures as well as a single-branch either structure can been eliminated without lose of usefulness. This results in the tree depicted in Fig. 4-3 starting next page.
Finally, a simple in-order search of the remaining tree is the base for the layout of the final transaction specification text, or trek_text, in whatever appropriate transaction language syntax we choose. The labels in the nodes of the trimmed trek_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. Back to the beginning, Fig. 4-1 portrays the trek_text resulting from the above trimmed tree, using an English pseudo-code language. We also have Catalan pseudo-code, as well as directly executable Prolog code, but other languages can easily be added. For ease of comprehension, an indentation mechanism presents the trek_text as shown in Fig. 4-1.

This ends the explanation of the synthesis process leading to the output shown in the previous section for transaction request \([\text{App}^n(\text{Per})]\). The next two sections respectively describe the complete generation of consistency set \(C\), and the formal derivation rules used to start our synthesis processes.

4.3 Consistency conditions for our example schema

Let's see what consistency conditions should be initially included in the set \(C\) for our example information base schema. They are the ones emerging from the schema ICs. This initialisation of \(C\) is attained in our method through a translate derivation from the single request \([-t1c]\). This derivation immediately calls for a repair derivation, which guarantees failure for any possibly successful derivation in the search space implicitly defined by \((\leftarrow t1c)\) and the A(ibs). It does so by resolving its goals until they include at least one base event, then including such goals into set \(C\).

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 one of \(t\text{Cand} (\text{IR.1})\) in order to obtain the initial set \(C_0\) of our particular example, which is shown
in Fig. 4-4. C.1 to C.3 come directly from Ic1, and C.4 to C.8 correspond to Ic2. For example, the intuitive meaning of consistency condition C.4 is that the database will eventually go into an inconsistent state if we deleted as applicant some candidate, and pretend not to delete such person as candidate. Note, however, that for every consistency condition there is always an implicit preserving action, i.e. that of aborting whatever updates had been proposed so far, as shown in the use of C.1 in Fig. 4-2. Section 5 will provide more example syntheses that make use of other consistency conditions from Fig. 4-4.

### Fig. 4-4

<table>
<thead>
<tr>
<th>Code</th>
<th>Consistency condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>( \leftarrow \text{Emp}(x) \land \delta\text{Emp}(x) \land t\text{App}(x) )</td>
</tr>
<tr>
<td>C.2</td>
<td>( \leftarrow t\text{Emp}(x) \land \text{App}(x) \land \neg \delta\text{App}(x) )</td>
</tr>
<tr>
<td>C.3</td>
<td>( \leftarrow t\text{Emp}(x) \land t\text{App}(x) )</td>
</tr>
<tr>
<td>C.4</td>
<td>( \leftarrow \text{Cand}(x) \land \neg \delta\text{Cand}(x) \land \delta\text{App}(x) )</td>
</tr>
<tr>
<td>C.5</td>
<td>( \leftarrow \text{Int}(x,y) \land \neg \delta\text{Int}(x,y) \land t\text{Eco}(y) \land \neg \text{Cand}(x) \land \neg \text{App}(x) \land \neg t\text{App}(x) )</td>
</tr>
<tr>
<td>C.6</td>
<td>( \leftarrow t\text{Int}(x,y) \land \text{Eco}(y) \land \neg \delta\text{Eco}(y) \land \neg \text{Cand}(x) \land \neg \text{App}(x) \land \neg t\text{App}(x) )</td>
</tr>
<tr>
<td>C.7</td>
<td>( \leftarrow t\text{Int}(x,y) \land t\text{Eco}(y) \land \neg \text{Cand}(x) \land \neg \text{App}(x) \land \neg t\text{App}(x) )</td>
</tr>
<tr>
<td>C.8</td>
<td>( \leftarrow \text{Cand}(x) \land \delta\text{App}(x) )</td>
</tr>
</tbody>
</table>

### 4.4 Formalisation of Trek_tree generation

We now provide a formal definition of the first and core step of our method, whose application has been illustrated in the example of Section 4.2. Translate and repair derivations explore the generic search space implicitly defined by the posed transaction request Tr together with \([-tIc]\) and the A(IDS), while conveniently building 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. Fig. 4-5 below includes examples and related trek_tree labels for the various predicate types in the A(IBS).

### Fig. 4-5

<table>
<thead>
<tr>
<th>A(IBS) predicate type</th>
<th>Examples (F.O.L.)</th>
<th>Result.Trek-tree label ex.(Prolog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>old (state) base predicate</td>
<td>( \text{App}(p),\text{Int}(p,c),\text{Eco}(c),\text{Emp}(p) )</td>
<td>( \text{precond(app(P)}, \text{fsome_cond(int(P,_C),[C]), fnew_cond(not eco(_C),[C]), feachCond(emp(_P),[_P]),...} )</td>
</tr>
<tr>
<td>old (state) derived predicate</td>
<td>( \text{Cand}(p),... )</td>
<td>( \text{precond(cand(P))}, \text{fnome}_\text{cond(cand(}_P,[_P]), fnew_cond(not cand(_P),[_P]), feach_cond(cand(_P),[_P]),...} )</td>
</tr>
<tr>
<td>evaluative predicate</td>
<td>( p_1=\text{John}, _\text{age}&lt;=32,_... )</td>
<td>( \text{precond(P1=\text{john}),...} )</td>
</tr>
<tr>
<td>base event predicate</td>
<td>( t\text{App}(p),\delta\text{Int}(p,c),... )</td>
<td>( \text{action(t_app(P))}, \text{action(d_int(P,C))} )</td>
</tr>
<tr>
<td>derived event predicate</td>
<td>( t\text{Cand}(p),\delta\text{Cand}(p),... )</td>
<td>( \text{commented_action(t_cand(P))}, \text{commented_action(d_cand(P))} )</td>
</tr>
<tr>
<td>new (state) predicate</td>
<td>( \text{App}^n(p),\text{Cand}^n(p),... )</td>
<td>( \text{cond_struct} )</td>
</tr>
<tr>
<td>- new (from Old state)</td>
<td>( \text{App}^{n_0}(p),\text{Cand}^{n_0}(p),... )</td>
<td>( \text{exit_struct} )</td>
</tr>
<tr>
<td>- sequence-level</td>
<td>( \text{Cand}^{n}(p),... )</td>
<td>( \text{sequence_struct} )</td>
</tr>
<tr>
<td>- new (from Transition)</td>
<td>( \text{App}^{n_1}(p),\text{Cand}^{n_1}(p),... )</td>
<td>( \text{either_struct, cond_struct} )</td>
</tr>
</tbody>
</table>
Old state predicates represent "preconditions" that must be checked against the old state of the information base prior to the application of any accompanying base or derived event predicate. When they are not trek-ground (see later), then they become the starting conditions for forsome, forsome-new or foreach instructions. Base events represent the update "actions" that can be performed on base predicates of the information base. 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 "commented action" preceding its translation. New predicates conceptually represent the evaluation of their corresponding base or derived predicate on the updated information base, that is they represent transaction postconditions. Besides their use as the designer intents in the initial update transaction request, they are also 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 holds in the new state because it remains untouched from the old state, thus making unnecessary any further actions ("exit_struct"). Other new predicates indicate that their resolvents should be analysed as if they formed a sequence of transaction pieces ("sequence_struct"). Others 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 ("either_struct") or conditionally depending on the old database state ("cond_struct"). In all previous cases, the possibly many alternatives become then children sub-trees of the corresponding node.

In the step-by-step traversal of the relevant search space made by translate (and repair) derivations, each step starts from input vertex \((G_i \ C_i \ N_j)\) and finishes with output vertex \((G_{i+1} \ C_{i+1} \ N_{j+1})\). \(G_i\) is the goal under translation (resp. repair), which once resolved by the step becomes \(G_{i+1}\). \(C_i\) is the the set of consistency 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 \(N_j\) with a further output node \(N_{j+1}\). Each such trek_tree node has a label and zero, one or more children sub-trees, to be built by future derivation steps.

A translate derivation is a sequence from ("← Tr,→Ic" \(\{\) TN0\(\})\) to either (\(\{\) Cn\(m\) [ValidSubTree]\)) or (Gn\(m\) Cn\(m\) [DeadEndedSubTree]), where the former case ends the analysis of one of possibly many alternative ways of translating the user request, and the later invalidates (part of) 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. A repair derivation starts with (F0 C0 RN0) and ends with either (\(\{\) Cn [ValidSubTree]\)) or (Fn Cn [DeadEndedSubTree]). 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, (conditionally) invalidates the original translate derivation as an alternative translation source. For doing all this, repair derivations maintain, check and use the "Consistency conditions set" \(C\), explained above. In the description below, TNj is identify nodes originated in a translate derivation and RNj identifies 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. Besides some hopefully self-explanatory conventions, below we use a few functions that are defined at the end of this section.
Translate-derivations-tree and translate derivation descriptions

A translate-derivations-tree is a tree with root, intermediate nodes and ending leaves of the form \((G_i \land C_i \land TN_i)\). 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: \((G_1 \land C_1 \land TN_1), (G_2 \land C_2 \land TN_2), \ldots, (G_n \land C_n \land TN_n)\) such that for each \(i \geq 1\), \(G_i\) has the form \(\{\leftarrow L_1, \ldots, L_k\}\), \(ST(G_i) = L_j\) selects literal \(L_j\) from goal \(G_i\) according to the order of appearance of the following rules T1 to T5, and \((G_{i+1} \land C_{i+1} \land TN_{i+1})\) 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 \(G_{i+1} = G_i \land L_j\), \(TN_{i+1} = \{\text{precond}(L_j), [TN_{i+2}]\}\),
\[ T1.2\) otherwise if \(G_i = \{L_j \land rG_i\}\) then sk(L_j, rG_i, SL_j, G_{i+1}, SV) and \(TN_{i+1} = \{\text{some cond}(SL_j, SV), [TN_{i+2}]\}\).

**T2** If \(L_j\) is a negative base or derived event or negative new predicate then
\(G_{i+1} = G_i \land L_j\), \(TN_{i+1} = \{\text{Void}, [RN^0]\}\), where \(RN^0\) is obtained by the repair derivation from \(\{\leftarrow L_j\} C_i \land RN^0\) to either \(\{\} C' \land \{\text{Void}, [\]\) or \(\{\} C' \land \{\text{dead_end}, [\}\) and then \(C_{i+1} = C'\).

**T3** If \(L_j\) is a positive new predicate then
\(G_{i+1} = G_i \land L_j\), \(C_{i+1} = C_i\), and if
\[ T3.0\) \(\{\leftarrow R_1\} C_i \land TN_{i+2}, 1\), \ldots, \(\{\leftarrow R_m\} C_i \land TN_{i+2}, m\), then if
\[ T3.1\) exit_level(L_j) then \(TN_{i+1} = \{\text{exit_struct}, [TN_{i+2}, 1] \ldots TN_{i+2}, m\}\), else if
\[ T3.2\) either_level(L_j) then \(TN_{i+1} = \{\text{either_struct}, [TN_{i+2}, 1] \ldots TN_{i+2}, m\}\), else if
\[ T3.3\) conditional_level(L_j) then \(TN_{i+1} = \{\text{cond_struct}, [TN_{i+2}, 1] \ldots TN_{i+2}, m\}\),
\[ T3.4\) otherwise \(TN_{i+1} = \{\text{Void}, [TN_{i+2}, 1]\}\), since \(m = 1\) by the A(DBS).

**T4** If \(L_j\) is a positive base event then if
\[ T4.1\) tg(L_j) then \(G_{i+1} = G_i \land L_j\), \(TN_{i+1} = \{\text{action}(L_j), [\text{repair_struct}, [RN^0]]\}\),
\[ T4.2\) otherwise if \(G_i = \{L_j \land rG_i\}\) then sk(L_j, rG_i, SL_j, G_{i+1}, SV) and \(TN_{i+1} = \{\text{fnew.cond}(pc(SL_j), SV), [\text{action}(L_j), [\text{repair_struct}, [RN^0]]\}\); where \(RN^0\) is obtained by the repair derivation from \((RC_j \land C_i \land RN^0)\) to either \(\{\} C' \land \{\text{Void}, [\}\) or \(\{\} C' \land \{\text{dead_end}, [\}\) and then \(C_{i+1} = C'\). \(RC_j\) 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
\(G_{i+1} = R_1\), \(C_{i+1} = C_i\), \(TN_{i+1} = \{\text{commented_action}(L_j), [TN_{i+2}]\}\).

Rules T1.x above deal with those precondition label types relevant to translate derivations. Rule T2 calls for consistency maintaining repair derivations from predicates that are not to hold, or otherwise to be avoided. Rules T3.x include various structure labels in the trek tree depending upon the implied semantics of their considered new state predicate. Rules T4.x add base action labels and spawn consistency maintaining repair derivations relevant to their proposed action. Finally, rule T5 puts a (commented) derived action and starts its further translation.
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: \((F_1 \ C_1 \ RN_1), (F_2 \ C_2 \ RN_2), \ldots, (F_n \ C_n \ RN_n)\) such that for each \(i \geq 1\), \(F_i\) has the form \((\leftarrow L_1, \ldots, L_k) \cup F_i\). \(SR(F_i)=L_j\) 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** \(\text{[action}(L_j), \_\] \in \text{ancestors}(RN_i)\) (repair needed) then if

**R1.1.1** \(k=1\) and \(-\text{hpoc}(L_j, \_\) - no repair - then \(F_{i+1}={}\), \(RN_{i+1}=[\text{dead\_end}, []]\)

**R1.1.2** \(k=1\) and \(\text{hpoc}(L_j, \text{PoC}, SV)\) - conditioned dead\_end - then

\(F_{i+1}={}\), \(RN_{i+1}=[\text{cond\_struct}, [\text{precond}(SV)=[\text{PoC}]], [\text{dead\_end}, []]]\)

**R1.1.3** \(k>1\) -repair plausible- then

\(F_{i+1}={}\{\leftarrow L_1, \ldots, L_k \}\cup F_i\), \(C_{i+1}=C_i\), \(RN_{i+1}=[\text{Void}, [RN_{i+2}]]\).

**R1.2** \(\text{[action}(L_j), \_\] \in \text{ancestors}(N_i)\) then

\(F_{i+1}=F_i\), \(C_{i+1}=C_i \cup \{\leftarrow L_1, \ldots, L_k\}\), \(RN_{i+1}=[\text{Void}, [RN_{i+2}]]\).

**R2** If \(L_j\) is a positive derived event, then

\(F_{i+1}=S' \cup F_i\), \(C_{i+1}=C_i\), \(RN_{i+1}=[\text{Void}, [RN_{i+2}]]\), where \(S'\) is the set of all resolvents of clauses of the A(DFS) with \(\{\leftarrow L_1, \ldots, L_k\}\) on \(L_j\).

**R3** If \(L_j\) is a positive new predicate, then if

\(S=[R_1, R_2, \ldots, R_m]\) is the set of all resolvents of clauses in the A(DFS) with \(\{\leftarrow L_1, \ldots, L_k\}\) on \(L_j\) with corresponding repair derivations respectively rooted at \((\leftarrow R_1) C_i RN_{i+2,1}, \ldots, (\leftarrow R_m) C_i RN_{i+2,m}\), then if

**R3.1** sequence\_level\(L_j\) then \(F_{i+1}=F_i\), \(C_{i+1}=C_i\), \(RN_{i+1}=[\text{sequence\_struct}, [RN_{i+2,1} \ldots, RN_{i+2,m}]]\).

**R3.2** otherwise \(F_{i+1}=S' \cup F_i\), \(C_{i+1}=C_i \cup \{\leftarrow L_j\}\), \(RN_{i+1}=[\text{Void}, [RN_{i+2}]]\).

**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 \(F_{i+1}={}\{\leftarrow L_1, \ldots, L_k\}\cup L_j \cup F_i\), \(RN_{i+1}=[\text{precond}(L_j), [RN_{i+2}]]\).

**R4.2** otherwise if \(F_i={}\{L_j \land F_j\}\) then \(sk(L_j, rF_j, SL_j, F_{i+1}, SV)\) and \(RN_{i+1}=[\text{reach\_cond}(SL_j, SV), [RN_{i+2}]]\).

**R5** If \(L_j\) is an negative exit\_level\(L_j\) new predicate with corresponding translate\_derivations\_tree rooted at \((\leftarrow L_j) C_i TN_{i+2}^l\), then

\(F_{i+1}=F_i\), \(C_{i+1}=C_i\), \(RN_{i+1}=[\text{cond\_struct}, [TN_{i+2}^l[RN_{i+2}]]]\).

**R6** When \(\{\leftarrow L_1, \ldots, L_k\}\) is solely composed of other negative new predicates or negative base or derived events with corresponding translate\_derivations\_trees respectively rooted at \((\leftarrow L_1) C_i TN_{i+2}^l, \ldots, (\leftarrow L_k) C_i TN_{i+2}^k\), then

\(F_{i+1}=F_i\), \(C_{i+1}=C_i\), \(RN_{i+1}=[\text{either\_struct}, [TN_{i+2}^l \ldots, TN_{i+2}^k]]\).

Rules R1.x above check for consistency of a base event. If the event was already proposed within the trek\_tree branch under construction, then rules R.1.1.x either start a possible repair or mark the
branch as (conditionally) dead-ended, when no repair is available. Otherwise, rule R1.2 saves in set C the goal containing the event, for further consideration. Rule 2 simply unfolds a derived event using its corresponding definition in A(DBS). Rules R3.x do the same for new state predicates, that in repair derivations may require the addition of a "sequence_struct" label to the trek_tree or their inclusion in C as further conditions to be avoided. Rules R4.x deal with the two precondition label types appealing to repair derivations. Rule R5 spawns a conditioned exit structure for cases when it will not be necessary to propose any further redressing action. Rule R6 appends an "either_struct" label with as many children as redressing actions are in its goal, for which it calls the appropriate translate derivations.

In order to better understand the above formal descriptions, the meaning of some main functions used follows:

\[ \text{tg}(L) \] qualifies literal L as compile-time "trek-ground", that is having as terms either input parameters ('Per'), input constants (toni), or internally generated skolem variables ('_Comp'). Otherwise, when it has some variable ('y'), it is said to be non-trek-ground.

\[ \text{sk}(L,G,SL,SG,SV) \] respectively returns in SL and SG the result of skolemising goal \([L\land G]\) wrt. its existentially quantified variables, whose corresponding compile-time "skolem variables" are also returned in SV.

\[ \text{hpoc}(E,POC,SV) \] if event E has parameters or constants, they are returned in POC with corresponding skolem variables in SV; otherwise it returns false.

\[ \text{pc}(E) \] returns the precondition corresponding to event E, thus \(pc(tP)=-P\) and \(pc("\delta P")=P\).

5. SUPPORTING TRANSACTION DESIGN

In general, a transaction specification synthesised with our method may include every possible way in which its request could be accomplished. This may embrace several alternative ways for preserving consistency, translating an update to a derived predicate, or selecting relevant tuples for any of those. In our transaction specifications, all such alternative options may be presented under the premises of special ad-hoc control instructions, such as 'either' in Fig. 4-1. However, there are cases where a designer is not necessarily interested in the fullblown transaction specification but in a (still consistency-preserving) version of it. Such refinement may result from specialising the synthesis to particular design requirements, and/or from the appropriate handling of the synthesis (interim) outputs. We next comment on these possibilities through several examples.

For the simplest one, recall from Fig. 4-1 the two alternative ways of preserving consistency included within the 'either' control instruction. That was our first example of non-determinism within a transaction specification. In our transaction specifications, non-determinism may appear within consistency repairs, and in the context of translating updates to derived predicates. Since, in general, translate and repair transaction pieces may interleave, the resulting transaction specifications can be highly non-deterministic. However, we regard such non-determinism both as a good specification knowledge source for further transaction design, as well as the basis for an advanced transaction processing system and a sophisticated user-interaction system.

The trek_tree in Fig. 4-3 includes all consistency-preserving alternatives relevant to its original request. We used them all in the trek_text of Fig. 4-1. However, we could have searched such
trek_tree in a more specialised way in order to come up with different (customised) trek_texts. For example, a designer could be interested in considering just consistency checking for a particular transaction, thus only aborting any potential integrity violation. This would leave our example trek_text without lines 6, 7, 8 and 10. For some other transaction, s/he could be after integrity maintenance alone, i.e. not to consider aborts as long as there are possible compensating actions. Under this requirement, our example trek_text would not show lines 6, 8, 9 and 10. There are also interesting intermediate situations, where consistency checking might be used for some constraints while for some other constraints integrity maintenance is preferred. Such synthesis modes can be applied in a per transaction manner, or set as default preferences for an information base schema.

From a trimmed trek_tree, a designer could further choose, out of all the valid updating alternatives considered in it, those options most interesting for his/her application. This would not require to undo every non-deterministic situation within the tree. On the other hand, s/he can also rely on the run-time transaction processing system or the end-user to take some or all of the (remaining) decisions. The next two examples show more complex non-deterministic situations amenable to further design refinement and advanced use.

5.1 Synthesis output from [Emp^n(Per)]

If the designer issues the [Emp^n(Per)] request to our system, the method will generate the trek_text contained in Fig. 5-1 on next page. Within this figure, line 1 controls if the person to be employed is already an employee, in which case line 2 exits the transaction without any updating. If the person under consideration is not an employee, line 4 proposes to insert him/her as such. Such insertion of employee directly affects lC1, so a checking/maintenance preventive repair is drawn from consistency condition C.2 in Fig. 4-4. 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 lC2, thus reclaiming the corresponding repair, which can be obtained from consistency condition C.4 in Fig. 4-4. 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).

For the alternative of deleting the person as candidate, we initially draw the proposal that δCand(Per) 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 derived-update transaction request for deleting the extension of a derived predicate defined using a local variable. To accomplish such objective, we should eliminate any existing way in which the contents of the information base support the fact Cand(Per), for which we will now need to take into account the values taken by the local variables in the definition(s) of the view predicate. In our example, this is obtained with the 'foreach' instruction of lines 11 to 17. For this instruction we automatically synthesise the needed meaningful Skolem variable names (i.e. '!Comp'), depending upon the local 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.

Fig. 5-1

```
trek_text([n_emp(Per)],
1         if emp(Per) then
2         ... i_exit
3         else
4         ... i_emp(Per)
5         if app(Per) then
6         ... either
7         ... end either
8         if cand(Per) then
9         ... either
10        ... end either
11        ... { d_cand(Per) }
12        ... foreach [ _Comp ] in int(Per, _Comp) and eco(_Comp) do
13        ... either
14        ... end either
15        ... end foreach
16        ... or
17        ... i_abort
18        ... end either
19        ... end if
20        ... end if
21        ... or
22        ... i_abort
23        ... end either
24        ... end if
25        ... end if
26        ... end if
). % end of trek text
```

This example has shown the result of synthesising transaction text from a base update request, which in turn requires a corresponding integrity checking/maintenance repair, which itself further needs some derived-update translation code. That is, it exemplifies how we address in an integrative way the problems of base/derived updating, integrity checking and integrity maintenance within our transaction specification synthesis approach.

Again, Fig. 5.1 includes the transaction obtained directly from a trek_tree that includes all possible consistency-preserving and derived-update alternatives. But, as was said before, the designer could intervene in order to customise the resulting transaction to particular application-domain semantics or to personal requirements. Integrity checking alone, or integrity maintenance alone, or both adequately mixed would result in various versions of the above transaction in Fig. 5-1. S/he could also manipulate the original trek_tree in order to, for example, undo the derived-update non-deterministic situation. But such situations can also be resolved or even exploited for a more advanced used in later states, as will be commented in the following example.
5.2 Synthesis output from [Cand^n(Per)]

This example deals with a derived-update request for a transaction specification to make some person 'Per' candidate. For space limitations, we will only show the synthesis output for [Cand^n(Per)] without considering ICs. The complete commented output with ICs can be found in [Pas94b].

Fig. 5-2 below contains the trek_text for this 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).

<table>
<thead>
<tr>
<th>Fig. 5-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>trek_text((n_cand(Per)), % without ICs</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

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' with 'forsomes' is highly non-deterministic. Of course, the designer could purge some 'either' options. She could also restrict some 'forsome' instructions to their "forone" counterpart, which asks the user (resp. system) for just one (resp. the first) Skolem-variable value satisfying the condition. Out of the remaining alternatives, at run-time the user should choose one or more relevant 'either' options and guide the selection of (or provide) 'forsome' values. While the last 'either' option 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 guarantees that 'forsome' and 'forsome-new' 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 will require a sophisticated run-time user interaction system that we have not yet developed. 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). It is for situations like these that our transactions should better be synthesised under the selective guidance of a designer. In this case, s/he could also use application-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' options, or types of 'forsome' 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.

5.3 Specialising transaction synthesis

Let us finish with an interesting feature of our method that may be used along the customisation purposes of the examples above. To begin with a particular synthesis, the designer could also pose transaction requests including not only the expected transaction effects (as in previous examples) but also further literals which customise the generation of the trek_tree itself.

For example, recall that transactions can be specialised to particular initial states through the inclusion of "precondition requirements" within the transaction request, as in [Emp^n(Per),¬Cand(Per)], where the designer asks for a transaction to insert an employee, once it is known that s/he is not a candidate. The result for this request would be similar to that of Fig. 5-1, except for that the transaction would start with an 'if not cand(Per) then' guard and lines 8 to 21 would not be present, since Ic2 would not be affected.

Also, remember that a transaction request could include further negative base or derived events, such as in [Emp^n(Per),¬δApp(Per)], where their role is that of "transition-selective requirements"; in the given case the designer wants a transaction to include someone as employee as long as s/he is not deleted as applicant. Using this last feature, we are able to implement some interesting update policies such as the prevention of side-effects to selected base or derived predicates. This request would result in a transaction specification as the one in Fig. 5-1 except for lines 6, 8 to 22 and 24. That is, it would only offer an 'abort' for potential Ic1 violations. Thus, Ic2 cannot be affected by the (unwanted) repair action for Ic1, which does not appear anymore. For this particular example, the resulting situation coincides with that of desiring consistency checking only.
6. ADDITIONAL FEATURES OF OUR METHOD

[Pas94a] provides examples of additional features of our method in the context of database transaction design that have not been shown in this paper. These features include the representation of transition ICs and their use in transaction specification synthesis. The implicit handling of the modification operation through deletions and insertions is also considered there. Evaluable predicates are used too in our information base schemes and transaction requests. Also, the method is applied to transaction requests relating compound post-conditions affecting more than one base and/or derived predicate.

7. CONCLUSIONS AND FURTHER WORK

Transaction design is one of the key activities in conceptual modelling of information systems, but its support has not yet received enough attention by the research community. However, there are clear opportunities to improve this difficult and error-prone task.

In this paper we have presented a new method for the generation of consistency-preserving transaction specifications in the context of conceptual modelling of information systems. The method is based on the transition and internal events rules, which explicitly define the dynamic behaviour of the information base when updated. Using these rules, a formal method allows us to automatically synthesise a legal transaction specification from an initial update transaction request. The integrative way in which the method deals with the problems of base and derived updating, integrity checking and integrity maintenance can be considered as its most important asset. However, the results are also useful as the basis for more advanced transaction design support and more sophisticated transaction processing and user-interaction systems.

The method presented extends and adapts a previous one for transaction synthesis in deductive databases [PO94]. We may now deal with much more general schemes where rules may have local variables, for which we have researched appropriate transaction specification instructions. Also, we now regard such basic synthesis method as the kernel of an advanced transaction-desing-support-system in the context of conceptual modelling of information systems.

At its current stage, the synthesis part of the method has been fully prototyped using metaprogramming techniques in Prolog. We can also generate directly executable transaction specifications in Prolog in order to simulate information base updating within the dynamic main-memory Prolog database.

We plan to extend this work along several lines. The case of information base conceptual schemes with recursive rules and rules with aggregate functions must be appropriately studied, formalised and implemented. We may also consider the explicit treatment of the modification as an operation of its own. We believe that it will not be very difficult to extend our method to deal with initial "qualified" set-oriented update transaction requests (for ex. convert to employers all companies with whom our applicants have interviews). Last, there is plenty of further implementation work along the advanced transaction design support, processing and utilisation introduced in this paper.
ACKNOWLEDGEMENTS

The authors would like to thank Hendrick Decker and the rest of the Odissea Group (T.Urpi, E.Teniente, J.Sistac, M.R.Sancho, C.Quer, E.Mayol, D.Costal, C.Martín) for many useful comments and discussions on the theme of this paper.

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

REFERENCES


[Oli89] Olivé, A. "On the design and implementation of information systems from deductive conceptual models", Proc. of the 15th Int. Conf. on VLDB, Amsterdam, 1989, pp. 3-11.


LSI-95-1-R "Octree simplification of polyhedral solids", Dolors Ayala and Pere Brunet.


LSI-95-3-R "The complexity of searching implicit graphs", José L. Balcázar.

LSI-95-4-R "Design quality metrics for object-oriented software development", Alonso Peralta, Joan Serras, and Olga Slavkova.

LSI-95-5-R "Extension orderings", Albert Rubio.

LSI-95-6-R "Triangles, ruler, and compass", R. Juan-Arinyo.

LSI-95-7-R "The modifiability factor in the LESD project: definition and practical results", Nuria Castell and Olga Slavkova.

LSI-95-8-R "Learnability of Kolmogorov-easy circuit expressions via queries", José L. Balcázar, Harry Buhrman, and Montserrat Hermo.

LSI-95-9-R "A case study on prototyping with specifications and multiple implementations", Xavier Franch.

LSI-95-10-R "Evidence of a noise induced transition in fluid neural networks", Jordi Delgado and Ricard V. Solé.


Hardcopies of reports can be ordered from:

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

See also the Department WWW pages, http://www-lsi.upc.es/www/