Supporting Transaction Design in Conceptual Modelling of Information Systems

Joan A. Pastor-Collado, Antoni Olivé

Dept. de LSI - Facultat d'Informàtica
Universitat Politècnica de Catalunya
Pau Gargallo, 5, 08028 Barcelona, Catalonia
{pastor | olive} @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 a role in the final application of the transaction specification, for this one can be directly executable. Using a declarative, logic-based 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 derived 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. 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. There has also been some related work in the database field [CW90,Qia93,SS89,Wal91]. See [PO94,PO95] for more comparative details.

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 [PO94]. We now 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-based 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. Often, 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.

The paper (see [PO95] for an extended version) is organised as follows. Next section defines our accepted information base schemes 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 synthesis method through a detailed example. In section 5 we comment on how the method can be used to furtherly support transaction design with some additional examples. Finally, in section 6 we present our conclusions.

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 schemes 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 track of those already employed.

Fig. 2-1

Base predicate	Base predicate meaning
App(x)	'x' is a job applicant
Eco(y)	'y' is an employer company
Int(x,y)	'x' has an interview with 'y'
Emp(x)	'x' is an employee

Fig. 2-2

Derived predicate with rule	Derived predicate meaning
$Cand(x) \leftarrow Int(x,y) \wedge Eco(y)$	'x' is considered a job candidate when s/he has an interview with an employer

Fig. 2-3

Integrity rule	Integrity constraint meaning
$Ic1 \leftarrow Emp(x) \wedge App(x)$	Nobody can be both employee and applicant
$Ic2 \leftarrow Cand(x) \wedge \neg App(x)$	Candidates must be applicants

Formally, a deductive rule is a formula of the form: $A \leftarrow L_1 \wedge ... \wedge L_n$ 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 information base. 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. 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. Fig. 2-2 has our single derived predicate for defining job candidates.

Integrity constraints (ICs) are conditions that the information base is required to satisfy at all times. 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: $\leftarrow L_1 \wedge ... \wedge L_n$ 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*, thus taking the same form as deductive rules. We call them integrity rules. We will use in our example the two 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 the derived predicate Cand.

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, key concepts in our method. Conceptually, internal events, transition rules and internal events rules are meta-level constructs describing the dynamic behaviour of an information base when confronted with updates. These rules depend only on the information base schema. They are independent from the base facts stored, and from any particular update. In section 4, we will discuss the use of transition and internal events rules for transaction specification synthesis. The following presentation is an overview of theory explained elsewhere [for ex. PO94], where the reader will find the full details on their formal derivation.

Let IB be a information base, U an update and IB^n the "new" updated information base. We say that U induces a transition from IB (current state) to IB^n (new, updated state). We assume that U consists of a 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 IB^n. Formally, we associate to each predicate P an *insertion internal events* predicate ιP and a *deletion internal events* predicate δP, defined as:

(1) $\forall x(\iota P(x) \leftrightarrow P^n(x) \land \neg P(x))$

(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) $\forall x(P^n(x) \leftrightarrow (P(x) \land \neg\delta P(x)) \lor \iota P(x))$

(4) $\forall x(\neg P^n(x) \leftrightarrow (\neg P(x) \land \neg\iota P(x)) \lor \delta P(x))$

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 updates if P is a derived predicate. If P is an inconsistency predicate (i.e. Ic), then ιIc facts that occur during the transition will correspond to violations of its corresponding IC and δIc facts cannot happen in any transition. Two special-purpose system events are also used, 'ιAbort' and 'ιExit'; their meaning will be clear with the examples of sections 4 and 5.

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. Consider now one of such rules, say rule 'i': $P_i(x) \leftrightarrow L_1 \land ... \land L_q$. When the rule is to be evaluated in the updated state its form is $P^n_i(x) \leftrightarrow L^n_1 \land ... \land L^n_q$. Now if we replace each literal in the body by its equivalent definition, given in (3) and (4), we get a new rule, which defines predicate P^n_i (new state) in terms of current state predicates and of internal events. When this is done for all deductive rules defining predicate P, we obtain a whole new rule set, where it is convenient to distinguish between two types of rules:

1) Rules '**nO**': They explain when P remains true in the new state because it has not been changed during the transition, thus remaining as in the Old state. They are headed with P^{nO}_i (x) when they apply to a single definition 'i' of P, and with P^{nO} (x) when applying to P as a whole.
2) Rules '**nT**': They indicate all possible ways for P to become true in the new state due to some internal events occurred within the Transition. They are headed with P^{nT}_i (x) when they apply to a single definition 'i' of P, and with P^{nT} (x) when applying to P as a whole.

Finally, we may now refer to both P^{nO} and P^{nT} through: $P^n(x) \leftarrow P^{nO}(x)$ and $P^n(x) \leftarrow P^{nT}(x)$. We call these rules, i.e. with (possibly subindexed) conclusions P^n, P^{nT} and P^{nO}, *transition rules* for predicate P. The transition rules corresponding to the information base example are shown in Fig. 3-1 with a clear intuitive meaning. Thus, for example, TR.6 states that 'x' is a candidate in the new state, if s/he had a programmed interview with 'y' in the old state that has not been cancelled in the transition, and 'y' has been inserted as employer company during the transition.

Fig. 3-1

Code	Transition rule
TR.1	$Cand^n(x) \leftarrow Cand^{nO}(x)$
TR.2	$Cand^n(x) \leftarrow Cand^{nT}(x)$
TR.3	$Cand^{nT}(x) \leftarrow Cand^{nT}_1(x)$
TR.4	$Cand^{nO}(x) \leftarrow Cand^{nO}_1(x)$
TR.5	$Cand^{nO}_1(x) \leftarrow Int(x,y) \wedge \neg\ \delta Int(x,y) \wedge Eco(y) \wedge \neg\ \delta Eco(y)$
TR.6	$Cand^{nT}_1(x) \leftarrow Int(x,y) \wedge \neg\ \delta Int(x,y) \wedge \iota Eco(y)$
TR.7	$Cand^{nT}_1(x) \leftarrow \iota Int(x,y) \wedge Eco(y) \wedge \neg\ \delta Eco(y)$
TR.8	$Cand^{nT}_1(x) \leftarrow \iota Int(x,y) \wedge \iota Eco(y)$
TR.9	$Ic1^{nO} \leftarrow Emp(x) \wedge \neg\ \delta Emp(x) \wedge App(x) \wedge \neg\ \delta App(x)$
TR.10	$Ic1^{nT} \leftarrow Emp(x) \wedge \neg\ \delta Emp(x) \wedge \iota App(x)$
TR.11	$Ic1^{nT} \leftarrow \iota Emp(x) \wedge App(x) \wedge \neg\ \delta App(x)$
TR.12	$Ic1^{nT} \leftarrow \iota Emp(x) \wedge \iota App(x)$
TR.13	$Ic2^{nO} \leftarrow Cand(x) \wedge \neg\ \delta Cand(x) \wedge \neg App(x) \wedge \neg\ \iota App(x)$
TR.14	$Ic2^{nT} \leftarrow Cand(x) \wedge \neg\ \delta Cand(x) \wedge \delta App(x)$
TR.15	$Ic2^{nT} \leftarrow \iota Cand(x) \wedge \neg\ App(x) \wedge \neg\ \iota App(x)$
TR.16	$Ic2^{nT} \leftarrow \iota Cand(x) \wedge \delta App(x)$
TR.17	$App^n(x) \leftarrow App^{nO}(x)$
TR.18	$App^n(x) \leftarrow App^{nT}(x)$
TR.19	$App^{nO}(x) \leftarrow App(x) \wedge \neg\ \delta App(x)$
TR.20	$App^{nT}(x) \leftarrow \iota App(x)$
TR....	$Eco^n(y) \leftarrow ...;\ Int^n(x,y) \leftarrow ...;\ Emp^n(x)$

Let P be a derived or inconsistency predicate. Once P^{nT} has been stated, from formula (1) we get: $\iota P(x) \leftarrow P^{nT}(x) \wedge \neg P(x)$ 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. If P is an inconsistency predicate we can remove the literal ¬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: $\iota Ic \leftarrow \iota Ick$ with $k = 1..r$, where r is the number of ICs in the database. Fig. 3-2 shows the insertion internal events rules for the example.

If P is a derived predicate, we can use definition (2) for a deletion internal event to generate its corresponding *deletion internal events rule* of predicate P: $\delta P(x) \leftarrow P(x) \wedge \neg P^n(x)$. Last row in Fig. 3-2 includes the deletion internal events rule for our example.

Fig. 3-2

Code	Insertion internal events rule
IR.1	$\iota Cand(x) \leftarrow Cand^{nT}(x) \wedge \neg Cand(x)$
IR.2	$\iota Ic1 \leftarrow Ic1^{nT}$
IR.3	$\iota Ic2 \leftarrow Ic2^{nT}$
IR.4	$\iota Ic \leftarrow \iota Ic1$
IR.5	$\iota Ic \leftarrow \iota Ic2$
	Deletion internal events rule
DR.1	$\delta Cand(x) \leftarrow Cand(x) \wedge \neg Cand^n(x)$

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 transaction specificacion synthesis. The augmented information base schema for our example would be the union of the contents of Figs. 2-1, 2-2, 2-3, 3-1 and 3-2. It is easy to show that, because IBS is allowed, then A(IBS) is also allowed.

4. Synthesis of Transaction Specifications

4.1 Transaction Requests

We envision a transaction-design-support-system that builds transactions specifications from the corresponding design-time parameterised *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 $[\neg P^n(p)]$ 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 ('joan','UPC').

The simplest case is that of Tr being 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 $[App^n(Per)]$ and $[\neg Cand^n(Per)]$. With the first one the designer wants a transaction specification to insert the person 'Per' as applicant. In the case of $[\neg Cand^n(Per)]$, 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. For example, $[Aux1^n(Per)]$ with $Aux1(x) \leftarrow Emp(x) \wedge \neg App(x)$ can be used to synthesise a transaction specification for doing whatever is needed so that 'Per' is a non-applicant employee.

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. 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 transaction capable of performing those intents 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. See [PO95] for a detailed formalisation of the method.

4.2.1 Synthesis Output from [App^n(Per)]

Assume that a designer poses the request [App^n(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. 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 layout format will be also followed for the other example outputs in section 5.

Fig. 4-1

```
     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_if
              ). % 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.

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. 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^n(Per)]

The above used transaction request [App^n(Per)], together with the implicit consistency requirement [¬ιIc] 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. For the case of our example, Fig. 4-2 on next page shows the generic search space of interest, together with the resulting trek_tree.

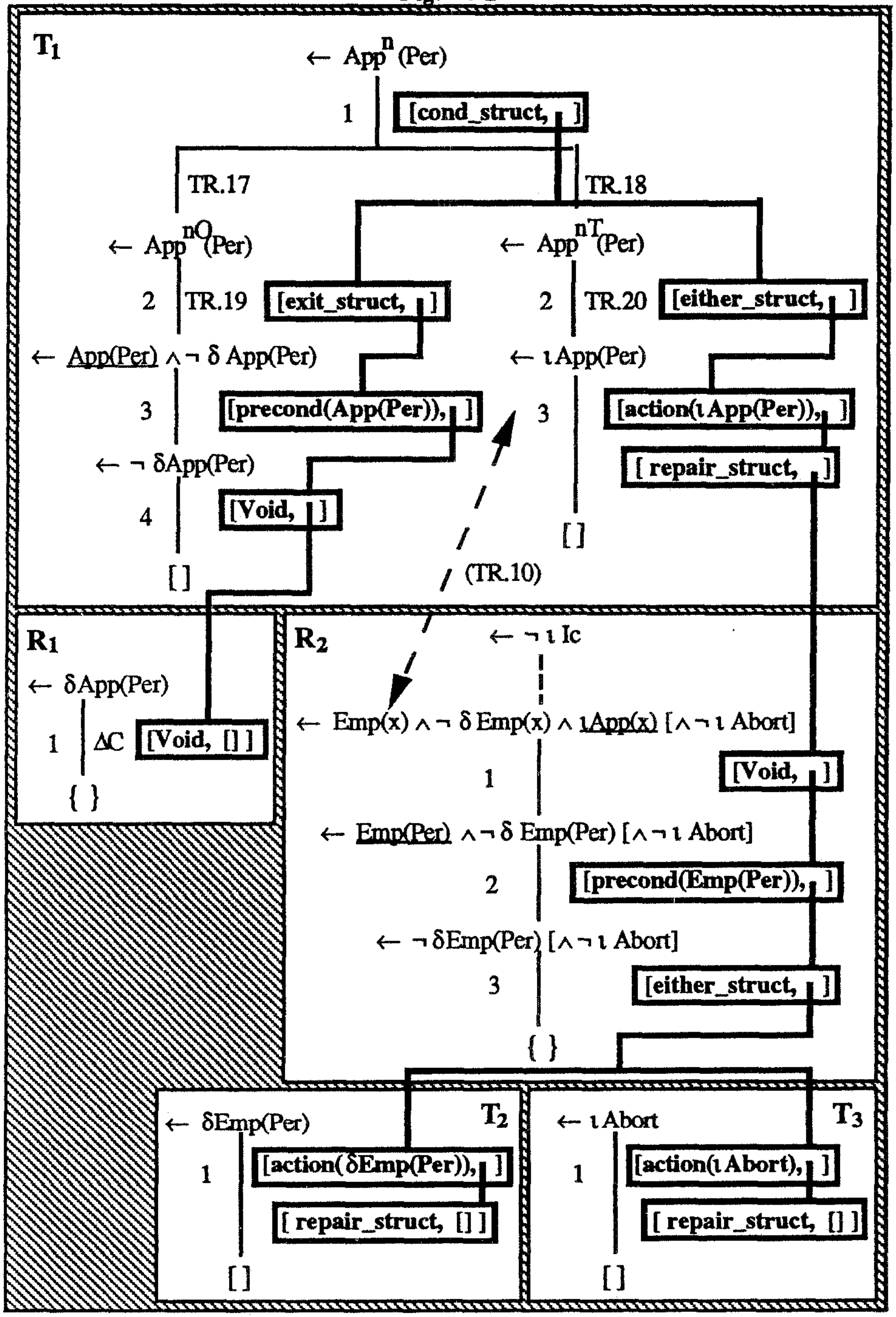
A *translate derivation* is used to obtain a "translation" from the original transaction request. Box T_1 in Fig. 4-2 includes the starting translate derivation rooted at the original request. Single translate steps explore and resolve their input goals until none is left. Intuitively, App^n(Per) will 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 can be found in [PO95].

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(IBS) 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 [¬ιIc], which is implicitely appended to every other transaction request. For our current example, only one such condition is used, which coincides with the body of rule TR.10 from Fig. 3-1. For this and other consistency conditions there is always an implicit preserving action, i.e. that of aborting whatever updates had been proposed so far, as shown in Fig. 4-2.

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. Ic1, in the way shown in R_2. There, the above mentioned consistency condition 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. 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 obtain the translations for their found redressing actions. These actions may include base updates, such as δEmp(Per) in T_2, or the special 'ιabort' 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(IBS).

Fig. 4-2


T1
← App^n (Per)
1
[cond_struct,]
TR.17
TR.18
← App^nO(Per)
← App^nT(Per)
2
TR.19
[exit_struct,]
2
TR.20
[either_struct,]
← App(Per) ∧ ¬ δ App(Per)
← ι App(Per)
3
[precond(App(Per)),]
3
[action(ι App(Per)),]
← ¬ δApp(Per)
[repair_struct,]
4
[Void,]
[]
[]
(TR.10)
R1
← δApp(Per)
1
ΔC
[Void, []]
{ }
R2
← ¬ ι Ic
← Emp(x) ∧ ¬ δ Emp(x) ∧ ιApp(x) [∧ ¬ ι Abort]
1
[Void,]
← Emp(Per) ∧ ¬ δ Emp(Per) [∧ ¬ ι Abort]
2
[precond(Emp(Per)),]
← ¬ δEmp(Per) [∧ ¬ ι Abort]
3
[either_struct,]
{ }
T2
← δEmp(Per)
1
[action(δEmp(Per)),]
[repair_struct, []]
[]
T3
← ι Abort
1
[action(ι Abort),]
[repair_struct, []]
[]

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. 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. Fig. 4-1 portraits the trek_text resulting from the above tree, once trimmed, using an English pseudo-code language.

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.

For a simple example, 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. 4-2 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. There are also interesting intermediate situations, where consistency checking might be used for some constraints while for some other constraints integrity maintenance is preferred.

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 [$\neg App^n(Per)$]

If the designer issues the [$\neg App^n(Per)$] request to our system, the method will generate the trek_text contained in Fig. 5-1. Within this figure, line 1 controls if the person to be employed is already an applicant, in which case line 2 proposes to delete him/her as such. Such deletion of applicant directly affects Ic2, so a checking/maintenance preventive repair is drawn from a consistency condition that coincides with the body of rule TR.14 in Fig. 3-1. That is, in case that such not-to-be-applicant were also a candidate (line 3) either it should be deleted as such (lines 5 to 12) or an abort should be proposed (line 14).

For the alternative of deleting the person as candidate, we initially draw the proposal that $\delta Cand(Per)$ should be pursued, shown in line 5 as a commented action preceding its unfolding. Later on, our method translates such derived-update request into the needed base update instructions (lines 5 to 12).

Line 3 together with lines 5 to 12 in fact correspond to the main body of the transaction that would be synthesised from the [$\neg Cand^n(Per)$] request. This is a request for deleting an instance 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 variable(s) in the definition(s) of the view predicate. In our example, this is obtained with the 'foreach' instruction of lines 6 to 12. For this instruction we automatically synthesise the needed meaningful Skolem variable names (i.e. '_Comp'). Line 6 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.

Fig. 5-1

```
     trek_text([not n_app(Per)],
 1   --------- if app(Per) then
 2   ----------|- d_app(Per) ,
 3   ----------|--- if cand(Per) then
 4   ----------|----|-- either
 5   ----------|----|---|- { d_cand(Per) }
 6   ----------|----|---|-- foreach [_Comp] in int(Per, _Comp) and eco(_Comp) do
 7   ----------|----|---|---|--- either
 8   ----------|----|---|---|--- |--- d_int(Per, _Comp)
 9   ----------|----|---|---|--- |- or
10   ----------|----|---|---|--- |--- d_eco(_Comp)
11   ----------|----|---|---|--- end_either
12   ----------|----|---|-- end_foreach
13   ----------|----|---|- or
14   ----------|----|---|--- i_abort
15   ----------|----|-- end_either
16   ----------|--- end_if
17   --------- end_if
                ). % end of trek text
```

For each such company, lines 7 to 11 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.

This example shows how we address in an integrative way the problems of base and 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.

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 only show in Fig. 5-2 the synthesis output for [Cand^n(Per)] without considering ICs.

Fig. 5-2

```
    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
```

Fig. 5-2 above 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).

The 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. S/he could also restrict some 'forsome' instructions to their "forone" counterpart, which asks the user (resp. system) for just one (resp. the first found) 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 nformation base 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, to be applied 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 variable values. Other additional features of our method are explained in [PO95].

6. Conclusions and Further Work

Transaction design is one of the key activities in conceptual modelling of information systems but its support has not received enough attention by the research community.

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.

At its current stage, the synthesis part of the method has been fully prototyped using meta-programming 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: formalisation and implementation of the case of schemes with recursive rules and rules with aggregate functions; explicit treatment of the modification operation; consideration of more complex initial transaction requests. 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 H. Decker and the rest of the Odissea Group (T.Urpí, E.Teniente, J.Sistac, M.R.Sancho, C.Quer, C.Martin, E.Mayol, D.Costal) for many useful comments and discussions on the theme of this paper. We also appreciate the valuable comments from the unknown referees.
This work has been partially supported by the CICYT PRONTIC project TIC 680.

References

[CA+94] Coleman,D.;Arnold,P.;Bodoff,S;Dollin,C;Gilchrist,H.;Hayes,F.;Jeremes,P. "Object-oriented Development. The Fusion Method". Prentice Hall Intl., 1994.

[CFT91] Casanova,M.A.;Furtado,A.L.;Tucherman,L. "A Software Tool for Modular Database Design". ACM TODS, Vol. 16, No. 2, June 1991, pp. 209-234.

[CHF92] Casanova,M.A.;Hemerly,A.S.;Furtado,A.L. "A Declarative Conceptual Modelling Language: Description and Example Application". Proc. of the 4th Int. Conf. CAiSE'92, Manchester, 1992, pp. 589-611.

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

[PO94] Pastor,J.A.; Olivé,A. "An Approach to the Synthesis of Update Transactions in Deductive Databases", Proc. of the 5th. Int. Conference on Information Systems and Management of Data (CISMOD'94), Madras, India, 1994.

[PO95] Pastor,J.A.; Olivé,A. "Supporting Transaction Design in Conceptual Modelling of Information Systems (Extended Version)", Internal research report LSI-95-11-R, Dept. LSI, UPC, Barcelona, 1995.

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

[SO94] Sancho,M.R.; Olivé,A. "Deriving Transaction Specifications from Deductive Conceptual Models of Information Systems". Proc. of the 6th Int. Conf. CAiSE'94, Utrech, The Netherlands, 1994, pp. 311-324.

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

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