The Odissea approach to the design of information systems from deductive conceptual models

Maria Ribera Sancho
Antoni Olivé

Report LSI–93–16–R

THE ODISSEA APPROACH TO THE DESIGN OF INFORMATION SYSTEMS FROM DEDUCTIVE CONCEPTUAL MODELS

Maria Ribera Sancho
Antoni Olivé

Facultat d'Informàtica
Pau Gargallo, 5
E 08028 Barcelona
Catalonia

e-mail:ribera@lsi.upc.es

January, 1992

ABSTRACT

This paper describes the general framework and some details of the ODISSEA approach to conceptual modelling and design of information systems. We use a logic-based language for the specification of conceptual models and apply extensively logic-based techniques for the generation of a system design from a conceptual model.

The main originality of our work is that our language is based on the "deductive" approach to conceptual modelling, instead of the usual "operational" approach. Deductive conceptual models (DCM) show a number of significant advantages, but are much more difficult to design and implement.

We deal with the problem of generating a design for a given DCM. We present a new, formal method for generating transactions from a DCM. The method uses the SLDNF proof procedure and can be implemented in Prolog environments.

A general trend in informations systems is the adoption of knowledge engineering techniques to enhance existing methodologies as well as to introduce new and more productive development paradigms, methods and supporting tools [Bub86]. This trend is based on the fact that information systems development is a knowledge intensive task [MBJ90] and, thus, it is not surprising that a lot of research has been directed toward providing knowledge-based tools to support the entire information systems development life cycle, from high-level design to low-level code generation [Fre85].

This paper describes the general framework and some details of the ODISSEA approach to conceptual modelling and design of information systems. Our approach uses a logic-based language for the specification of conceptual models, and applies logic-based techniques for the generation of a system design from a conceptual model.

The main originality of our work is that the ODISSEA language is based on the "deductive" approach to conceptual modelling, instead of the traditional, "operational" approach. Both approaches can provide a complete specification of the static and dynamic aspects of an information system, but they differ in the way the dynamic aspect is modelled.

In the operational approach, changes to the Information Base (IB), corresponding to changes in the Universe of Discourse (UoD), are defined by means of operations. The occurrence of a real-world, external event triggers the execution of an operation (transaction), which reflects the effect of the event on the IB. These effects consist usually of insertions, updates or deletions to the IB. On the other hand, operations, as well as queries and integrity constraints usually have only access to the current state of the IB.

In the deductive approach, the IB is defined only in terms of the external events, by means of deductive rules, and queries and integrity constraints are defined as if the complete history of the IB were available. A deductive conceptual model (DCM) is a specification of an IS in the deductive approach. Examples of conceptual modelling languages using the deductive approach are DADES [Oli82] and CIAM [GKB82].

A detailed comparison of the operational and deductive approaches can be found in [BuO86, Oli86]. The main conclusions are that DCMs provide more local definitions, are easier to change and to accomodate new requirements, and provide more design freedom. However, DCMs are much more difficult to implement than operational models. The reason is that an operational model already embeddes some architectural design decisions, which are not made in DCMs [Oli89].

The main design decisions required to implement an information system from a DCM are data base design and transaction design. Usually, many valid alternatives in data base design exist, and the designer must choose the alternative that he/she considers most appropiate. Complete automation of this decision is not possible, but there is a place for CASE tools that aid the designer by confirming the consistency of his/her decisions and by evaluating their impact in terms of performance.

In transaction design, the designer must decide which transactions will exist and, for each of them, when will be executed, its pre-conditions and the actions to be performed, including data base updates and output production.

Contrary to data base design, transaction design from a DCM can be completely automated. We can build a transaction for each external event, to be executed when that event occurs in the real world. Transaction pre-conditions can be determined from the integrity constraints defined in the DCM. Data base updates can be determined from an analysis of deductive rules, and output production can be determined from queries definition.

In this paper we present a new, formal method for generating transactions from a DCM. The method is based on the use of the SLDNF proof procedure and, thus, it can be implemented easily in Prolog environments.

To our knowledge, there is no similar work to ours in the deductive approach, although there exists some similar research in the context of the operational approach. We can only mention here the recent work from the DAIDA project, reported in [CKM91], where additional references to previous research can be found. DAIDA proposes a dependency-based framework for the mapping from a requirements specification into a system design. The framework is dependency-based in the sense that the mapping of parts of the requirements specification is guided by predefined allowable dependencies. At the same time, the framework is goal-oriented, in the sense that non-functional requirements are treated as possibly conflicting goals to be satisfied by the generated design.

In DAIDA, requirements specification prescribe not only the behaviour of the system, but also the environment within which it will function. In ODISSEA, we focus only on the system to be developed, and our specifications are executable. This allows us to automate part of the mapping from specification to design. On the other hand, we have not considered yet the role of non-functional requirements in the generation of designs.

The paper is organized as follows. Section 2 briefly introduces the main components of a DCM, including an example that is used throughout the paper. Section 3 describes the internal architecture of the system that we would like to be able to generate from a DCM. Section 4 presents the Internal Events Model (IEM), a key concept in our approach to design generation. Section 5 discusses the use of the IEM in transaction generation, and gives a formal method for deriving transaction specifications from a DCM. Section 6 provides an overview of the ODISSEA design environment, showing the context in which transaction generation can be done. Finally, Section 7 summarizes the results of our work and points out future research.

2. DEDUCTIVE CONCEPTUAL MODELS

A deductive conceptual model (DCM) of an IS consists of five sets:
- A set B of base predicates
- A set D of derived predicates
- A set IC of integrity constraints
- A set Q of predefined queries
- A set A of alerts

In the following, we briefly describe each of these sets. Figure 1 shows an example that will be used troughout the paper. The example is written in a simplified syntax of the ODISSEA language [QuS91].

Base predicates correspond to the external event types. They are the inputs to the IS. Each fact of a base predicate, called base fact, is an occurrence of an external event. We assume, by convention, that the last term of a base fact gives the time when the event occurred and was communicated to the IS. If $p(a_1,..., a_n,t_i)$ is a base fact we say that $p(a_1,..., a_n)$ is true or holds at t_i. All times are expressed in a unique time unit (such as second, day, etc.) small enough to avoid ambiguities.

In the example of figure 1 we have four base predicates: start, end, assign and mng. A base fact start(p,t) means that project p started at time t. A base fact end(p,t) reports that project p ends at time t. A base fact assign(pg,p,t) means that at time t, programmer pg is assigned to project p. A base fact mng(pg,t) means that at time t, programmer pg becomes a manager. We take as time unit a second.

Derived predicates model the relevant types of knowledge about the Universe of Discourse. Each fact of a derived predicate, called derived fact, represents an information about the state of the UoD, at a particular time point. We also assume that the last term gives the time when the information holds.

Each derived predicate is defined by means of one or more deduction rules. A deduction rule of predicate p has the form $p(X_1,..., X_n,T) \leftarrow L_1,..., L_m$, where $p(X_1,..., X_n,T)$ is an atom denoting the conclusion and $L_1,..., L_m$ are literals representing conditions. Each L_j is either an atom or a negated atom. Variables in the conclusion or in the conditions 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.

Condition predicates may be ordinary or evaluable ("built-in"). The former are base or derived predicates, while the latter are predicates, such as the comparison or arithmetic predicates, that can be evaluated without accessing a database.

We assume every rule to be allowed [GMN84], i.e. any variable that occurs in the rule has an occurrence in positive condition of an ordinary predicate. We also require every rule to be time-restricted. This means that for every positive literal q(...,T1) of a base or derived predicate q occurring in the body, the condition $L_1,..., L_m \rightarrow T1 \leq T$ must hold. This ensures that $p(X_1,..., X_n,T)$ is defined in terms of q-facts holding at time T or before.

In the example, there are seven derived predicates, with their corresponding (and hopefully self-explanatory) rules. Note that predicate active is defined by two rules. The first rule states that a project is active if it has a leader, while the second rule states that a project is also active when two programmers have been assigned to it.

Integrity constraints are closed first-order formulas that base and/or derived facts are required to satisfy. We deal with constraints that have the form of a denial $\leftarrow L_1,..., L_m$, with $m \geq 1$, where the L_j are literals and variables are assumed to be universally quantified over the whole formula. More general constraints can be transformed into this form as described in [LlT84]. For the sake of uniformity, we associate to each integrity constraint an inconsistency predicate icn, with at least a time term, and thus it has the same form as the deductive rules. We call them integrity rules.

In the example of figure 1 we show four inconsistency predicates, with their rules. To see how an inconsistency may arise, assume that base facts start(is1,10) and assign(john,is1,15) were received at times 10 and 15, respectively. Now, if at time 20, the IS receives end(is1,20) inconsistency ic2(20) will arise, because active(is1,19) does not hold. Therefore, the base fact end(is1,20) must be rejected.

Outputs from an IS may be requested by the users (queries) or triggered internally by the system when some condition holds (alerts). Each query is defined by a name, a number of parameters, to which the user will give values when he makes the query, and a body. The answer to a query is the

set of values that satifies the conditions given in the body. (In order to focus on our objective, we omit in this paper alert definition and handling).

In the example, there are three queries. When q1(t) is issued at time t, the system will provide the set of projects which are active at t. When query q2(p,t) is issued at time t, with parameter p, the system will provide the programmers assigned some time to p. Note that this query requires the system to know the complete history of predicate assigned. Finally, when query q3(m,t) is issued at time t, the system gives the set of programmers subordinated to m at t.

base start(Project,Time)
base end(Project,Time)
base assign(Programmer,Project,Time)
base mng(Programmer,Time)

derived project(P,T) ← start(P,T1), T1 ≤ T, not(completed(P,T))
derived completed(P,T) ← end(P,T1), T1 ≤ T
derived assigned(Pg,P,T) ← project(P,T),assign(Pg,P,T1), T1 ≤ T
derived active(P,T) ← leader(Pg,P,T)
← assigned(Pg1,P,T), assigned(Pg2,P,T), Pg1≠Pg2
derived inactive(P,T) ← project(P,T), not(active(P,T))
derived leader(Pg,P,T) ← assigned(Pg,P,T),mng(Pg,T1), T1 ≤ T
derived subordinate(Pg,M,T) ← assigned(Pg,P,T),leader(M,P,T), Pg ≠ M

inconsistency ic1(T) ← start(P,T), project(P,T-1)
inconsistency ic2(T) ← end(P,T), not(active(P,T-1))
inconsistency ic3(P,T) ← leader(Pg1,P,T),leader(Pg2,P,T),Pg1≠Pg2
inconsistency ic4(T) ← assign(Pg,P,T), not(project(P,T-1))

query q1(T) **is** active(P,T)
query q2(P,T) **is** assigned(Pg,P,T1), T1 ≤ T
query q3(M,T) **is** subordinate(Pg,M,T)

Figure 1. Example of Deductive Conceptual Model

3. INTERNAL ARCHITECTURE

In ODISSEA, we aim at designing and building information systems with simple and conventional architectures. The only notable feature is that we build the systems on top of a Deductive DBMS (DDBMS). Currently, we are using a system coupling Prolog to a Relational Database system (RDBMS) [CGT90], but other DDBMS could be used as well. On the other hand, it would also be possible to use a RDBMS.

In order to motivate the problems that must be solved to generate an IS from a DCM, we describe in this section the main components of a generated IS, and give the particular structure they could have in our example. We also introduce here some notation. In later sections, we will explain how to generate the components from a DCM.

Figure 2 shows the main components of an IS and their relationships. The User Interface Management System (UIMS) handles all input/output interaction. The emphasis of our project is not in this (important) component and, therefore, we only have simple menu-driven dialogues.

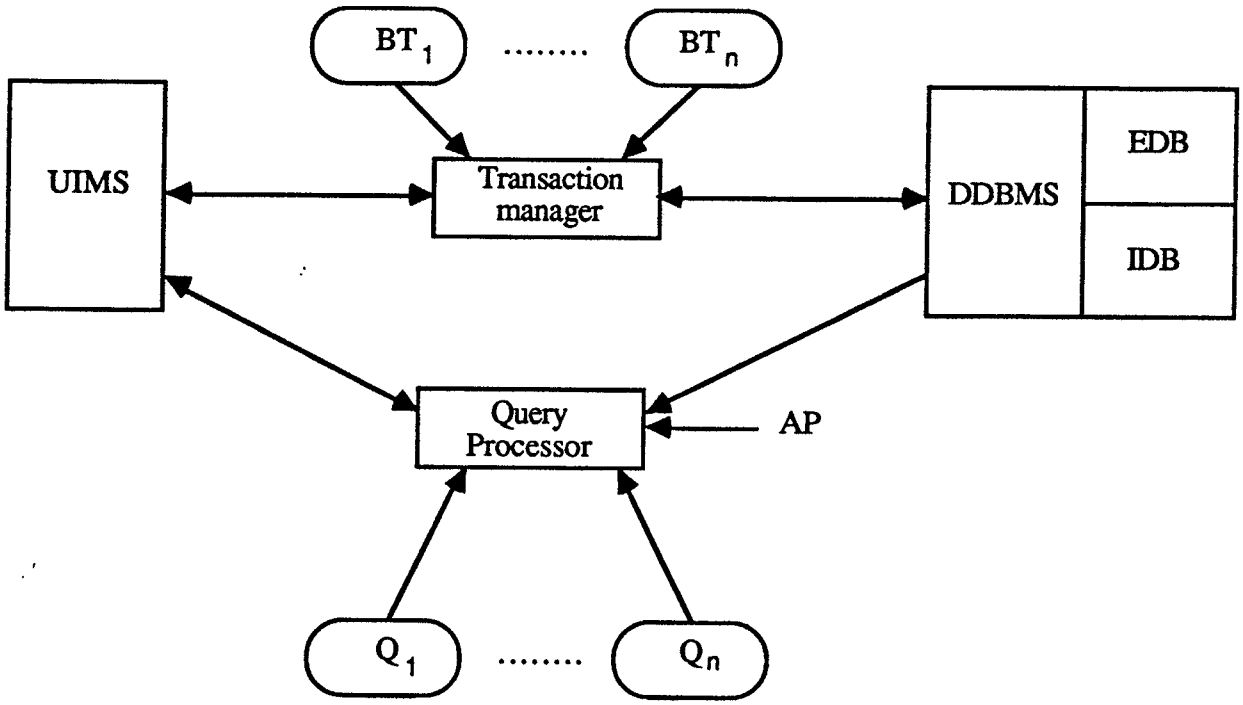

Figure 2. Internal architecture

The Extensional Data Base (EDB) comprises all facts explicitly stored in the data base. These facts do not need to be the base facts defined in the DCM. Indeed, it is quite usual to store in the EDB some derived facts, and not to store all base facts.

In our example, we could store in the EDB the current state of predicates projects and active, and the full history of predicates assigned and mng. Then a possible EDB schema could be:

PROJECTS(Project)
ACTIVE_PROJECTS(Project)
ASSIGNMENTS(Programmer,Project,T_start,T_end)
MANAGERS(Manager,Time)

The Intensional Data Base (IDB) comprises the rules that relate the DCM predicates to the facts stored in the EDB. Not all DCM predicates may be derivable from the EDB; this will depend on the designed EDB schema. We call Accessible Predicates (AP) those DCM predicates that may be derivable from the EDB. We also say that we have a Full History AP when the complete history of the predicate is derivable from the EDB, and that we have a Current State AP when only the current facts of the predicate are derivable. There are one or more rules in the IDB for each AP.

In our example, the APs are:

Full History: mng, assigned, leader, subordinate
Current State: project, active, inactive

and the main IDB rules are:

mng(M,T) ← MANAGERS(M,T).
project(P,T) ← PROJECTS(P).
assigned(Pg,P,T) ← ASSIGNMENTS(Pg,P,Ts,Te), Ts ≥ T, T ≤ Te.
active(P,T) ← ACTIVE_PROJECTS(P).
inactive(P,T) ← project(P,T), not(active(P,T)).
leader(M,P,T) ← assigned(M,P,T), mng(M,T1), T1 ≤ T.
subordinate(Pg,M,T) ← assigned(Pg,P,T),leader(M,P,T), M ≠ Pg.

Base Transactions (BT) update the EDB when a new base fact is received. There is a BT for each base predicate defined in the DCM. Typical actions performed by a BT are: receive the corresponding base fact from the UIMS, check integrity constraints satisfaction and update the EDB. We give in Section 5 a formal procedure to derive the BTs from a DCM.

In our example, there are four Base Transactions. We describe below the BT corresponding to predicates start and assign.

BT: start(p,t)
Check: project(p,t-1) does not hold (ic1)
Update: Insert <p> into PROJECTS

BT: assign(pg,p,t)
Check: mng(pg,T1),T1<t, leader(Pg2,p,t-1),pg≠Pg2 does not hold (ic3)
project(p,t-1) holds (ic4)
Update: Insert<pg,p,t,"∞"> into ASSIGMENTS
Insert <p> into ACTIVE_PROJECTS if
mng(pg,T1),T1<t, not(active(p,t-1))
or
assigned(Pg2,p,t-1),pg≠Pg2, not(active(p,t-1))

The Transaction Manager is a general module that handles all base transactions. We describe it in more detail in Section 5.

The Query Processor (QP) handles two types of queries: predefined and ad hoc queries. The former are those defined in the DCM, while the latter are queries whose definition must be given by the users at execution time. Users issue ad hoc queries in terms of predicates defined in the DCM, just as predefined queries, but only accessible predicates and facts can be requested.

If, in the example, the users issue at time 20 the ad hoc query:

? active(program1,10), leader(M,program1,20)

the answer would be "unknown", since only the current state of active-facts is accessible.

4. THE INTERNAL EVENTS MODEL

We have seen, in the previous Section, the internal architecture of the IS that we would like to be able to generate. Now, we start to describe our approach to the design and implementation of such IS from its DCM. The key concept of our approach is the Internal Events Model (IEM). This model was presented in [Oli89,San90], and has inspired novel solutions to the problems of integrity constraints enforcement and view updating of deductive databases [Oli91,TeO92]. In the following, we briefly describe the main concepts of an IEM, and show its application to the example. In later sections, we will discuss the use of the IEM in the design and implementation of IS.

4.1 Classification of predicates

Predicates defined in a DCM can be classified according to their temporal behaviour. For our purposes, the most important classification is the following. Let p be a predicate. Assume that fact p(**k**) holds at time T-1, where **k** is a vector of constants, and assume that no external events happen at time T, that is no base facts are received at T. What can we say about the truth of p(**k**) at time T?. There are three cases:

a) If we can say that p(**k**) will be false at time T, then we classify p as P-transient.
b) If we can say that p(**k**) will be true at time T, then we classify p as P-state.
c) If the truth of p(**k**) depends on the truth of some condition that must be evaluated at time T, we classify p as P-spontaneous.

Base predicates are always assumed to be P-transient. In our DCM example of figure 1, all derived predicates are P-state.

4.2 Internal events

The concept of internal event tries to capture in a natural way the notion of change in the extension of a predicate. We associate to each P-state or P-spontaneous predicate p an insertion internal events predicate ιp and a deletion internal events predicate δp defined as follows:

(1) $\forall X,T\ (\iota p(X,T) \leftrightarrow p(X,T) \wedge \neg p(X,T\text{-}1))$
(2) $\forall X,T\ (\delta p(X,T) \leftrightarrow p(X,T\text{-}1) \wedge \neg p(X,T))$

and then:

(3) $\forall X,T\ (p(X,T) \leftrightarrow (p(X,T\text{-}1) \wedge \neg\, \delta p(X,T)) \vee\ \iota p(X,T))$
(4) $\forall X,T\ (\neg p(X,T) \leftrightarrow (\neg p(X,T\text{-}1) \wedge \neg\, \iota p(X,T)) \vee\ \delta p(X,T))$

To P-transient predicates we only associate an insertion internal events predicate ιp defined as:

(5) $\forall X,T\ (\iota p(X,T) \leftrightarrow p(X,T))$

If p is a derived predicate, then ιp facts and δp facts represent induced insertions and deletions of derived facts, respectively. If p is an inconsistency predicate, then ιp facts that occur at time T will correspond to violations of its integrity constraint. Note that, for inconsistency predicates, δp facts

cannot happen in any transition, since we assume that the IB is consistent at time T-1 and, thus, p(X,T-1) is always false.

We also use the internal events concept for base predicates. In this case, ιp facts facts represent the external events (given by the environment) corresponding to insertions of base facts. We use sometimes the term "event" to denote either an internal or external event.

4.3 Transition rules

Let $p(X,T) \leftarrow L_1,..., L_m$ be a deductive or inconsistency rule. The rule defines the extension of p at time T in terms of the extensions at time T, or before, of the predicates appearing in the body of the rule. A transition rule for p is a rule that defines the extension of p at time T in terms of the extensions at time T-1, or before, of the predicates appearing in the body of the rule, and the internal events of these predicates that occur at time T. We obtain the transition rules substituting each L_j, j = 1...n, by its equivalent expression (see [Oli89,San90] for details).

Consider, for example, the rules for predicate active:

active(P,T) ← leader(Pg,P,T)
active(P,T) ← assigned(Pg1,P,T),assigned(Pg2,P,T),Pg1 ≠ Pg2

The literals in the body of the two rules can be replaced by their equivalent expressions, as given by (3). After distributing ∧ over ∨, we get :

active(P,T) ← leader(Pg,P,T-1),not(δleader(Pg,P,T))
active(P,T) ← ιleader(Pg,P,T)
active(P,T) ← assigned(Pg1,P,T-1), not(δassigned(Pg1,P,T)),
assigned(Pg2,P,T-1), not(δassigned(Pg2,P,T)), Pg1 ≠ Pg2
active(P,T) ← assigned(Pg1,P,T-1), not(δassigned(Pg1,P,T)),ιassigned(Pg2,P,T), Pg1 ≠ Pg2
active(P,T) ← ιassigned(Pg1,P,T), assigned(Pg2,P,T-1), not(δassigned(Pg2,P,T)), Pg1 ≠ Pg2
active(P,T) ← ιassigned(Pg1,P,T), ιassigned(Pg2,P,T), Pg1 ≠ Pg2

which are the transition rules for predicate active. Note that these rules allow us to infer active facts holding at time t in base to the leader and assigned facts holding at time t-1, and the events ιleader, δleader, ιassigned and δassigned that occur at time t.

4.4 Internal events rules

An internal event rule is a rule that defines the conditions upon which an internal event happens. Thus, internal event rules allow us to deduce which internal events happen in a given transition. For example, the rules:

ιproject(P,T) ← ιstart(P,T)
δproject(P,T) ← ιcompleted(P,T)

are an insertion and a deletion internal event rule, respectively. The first states that the occurrence of an ιstart fact (in this case, the insertion of an start base fact) induces a corresponding ιproject fact. The second rule states that the occurrence of a ιcompleted fact induces a corresponding δproject fact.

We get the internal events rules using a procedure based on the definitions (1), (2) and (5), replacing p(X,T) by its transition rules, and applying a set of standard simplifications, that may involve the integrity constraints. Figure 3 shows (part of) the result obtained by the application of the procedure to the DCM example of figure 1.

IDR.1 ιproject(P,T) ← ιstart(P,T)
IDR.2 δproject(P,T) ← ιcompleted(P,T)
IDR.3 ιcompleted(P,T) ← ιend(P,T)
IDR.4 ιassigned(Pg,P,T) ← ιassign(Pg,P,T), not(δproject(P,T))
IDR.5 ιactive(P,T) ← ιleader(Pg,P,T), not(active(P,T-1))
IDR.6 ιactive(P,T) ← ιassigned(Pg1,P,T), assigned(Pg2,P,T-1), not(δassigned(Pg2,P,T)), Pg1 ≠ Pg2, not(active(P,T-1))
IDR.7 ιactive(P,T) ← ιassigned(Pg1,P,T), ιassigned(Pg2,P,T), Pg1 ≠ Pg2, not(active(P,T-1))
IDR.8 ιleader(Pg,P,T) ← assigned(Pg,P,T-1), not(δassigned(Pg,P,T)), ιmng(Pg,T)
IDR.9 ιleader(Pg,P,T) ← ιassigned(Pg,P,T), mng(Pg,T1), T1 < T
IDR.10 ιleader(Pg,P,T) ← ιassigned(Pg,P,T), ιmng(Pg,T)

IDR.11 ιic1(T) ← ιstart(P,T), project(P,T-1)
IDR.12 ιic2(T) ← ιend(P,T), not(active(P,T-1)).
IDR.13 ιic3(P,T) ← ιleader(Pg1,P,T),leader(Pg2,P,T-1), not(δleader(Pg2,P,T)), Pg1 ≠ Pg2
IDR.14 ιic3(P,T) ← ιleader(Pg1,P,T), ιleader(Pg2,P,T), Pg1 ≠ Pg2
IDR.15 ιic4(T) ← ιassign(Pg,P,T), not(project(P,T-1)).

Figure 3. Part of the internal events model of the DCM example.

5. DERIVING BASE TRANSACTIONS

5.1 The approach

In this section we will focus on one aspect of the ODISSEA internal architecture: the Base Transactions. We show the use of the IEM to derive the set of actions that each BT should perform.

As we mentioned before, there is a BT for each base predicate defined in the DCM. A BT will receive the corresponding base fact from the UIMS, check integrity constraints satisfaction and update the EDB. We can also derive composite transactions, corresponding to the simultaneous occurrence of two or more base facts, but this extension will not be considered here.

Let TR={ιb(**k**,t)} the transaction corresponding to the insertion of a base fact b(**k**,t), where **k** is a vector of constants denoting the transaction parameters, and t the occurrence time. We derive its effects in the following way: For each integrity constraint icj specified in the DCM, we can check its violation by evaluating the internal event predicate ιicj. In the same way, for each stored predicate p(X,T), we can derive the induced insertion (resp. deletion) of a fact p(**x**,t) by evaluating the internal event predicate ιp(X,t) (resp. δp(X,t)). As IEM rules are defined in terms of accessible predicates extensions at time t-1 or before, to evaluate internal event predicates we also need the IDB rules, which relate accessible predicates to the facts stored in the EDB.

To obtain these evaluations we use the standard SLDNF proof procedure. More precisely, let I=i(X,t) be the internal event predicate to be evaluated. We say that TR induces I if goal {←I} succeeds from input set IEM ∪ IDB ∪ EDB ∪ TR. If every branch of the SLDNF-search space for IEM ∪ IDB ∪ EDB ∪ TR ∪ {←I} is a failure branch, then TR does not induce I.

Obviously, the evaluation can be completely done at execution time, when the concrete values of EDB and transaction parameters are known. However, we can do some preparatory work at compilation time, by partially evaluating IEM ∪ TR wrt I [LIS91]. The result of the partial evaluation is a set E of n rules (n≥0):

$$I \leftarrow C_i \qquad i = 1..n$$

where C_i is a conjunction of literals and such that evaluation of {←I} at execution time from E∪IDB∪EDB gives the same result al the evaluation of {←I} from IEM ∪ IDB ∪ EDB ∪ TR. If n=0 then TR does not induce I

Applying this procedure to our example case, at compilation time,we have to partially evaluate, for each possible transaction, internal event predicates ιic1(t), ιic2(t), ιic3(t), and ιic4(t), corresponding to the integrity checking conditions, and ιproject(P,t), δproject(P,t), ιactive(P,t), δactive(P,t), ιassigned(Pg,P,t), and δassigned(Pg,P,t), to derive the update conditions.

Take, as an example the transaction: TR={ιstart(p,t)}. These are the sets of rules obtained at compilation time:

ιic1(t) ← project(p,t-1)
ιproject(P,t) ←

5.2 Example

We will try to illustrate our approach with an example, previously to the formal definition of the method, given in next section.

Assume that we want to derive the transaction TR={ιassign(pg,p,t)}. We have to partially evaluate IEM ∪ TR wrt ιicj, corresponding to the integrity constraints. Consider, for example, the partial evaluation wrt ιic3(t). Possible sets of conditions C_i that would lead to the violation of ic3(t) are obtained by having some failed derivation of IEM ∪ TR ∪ {←ιic3(t)} succeed. Figure 4 shows this derivation tree, were the circled labels are references to the rules of the method, defined in section 5.3.

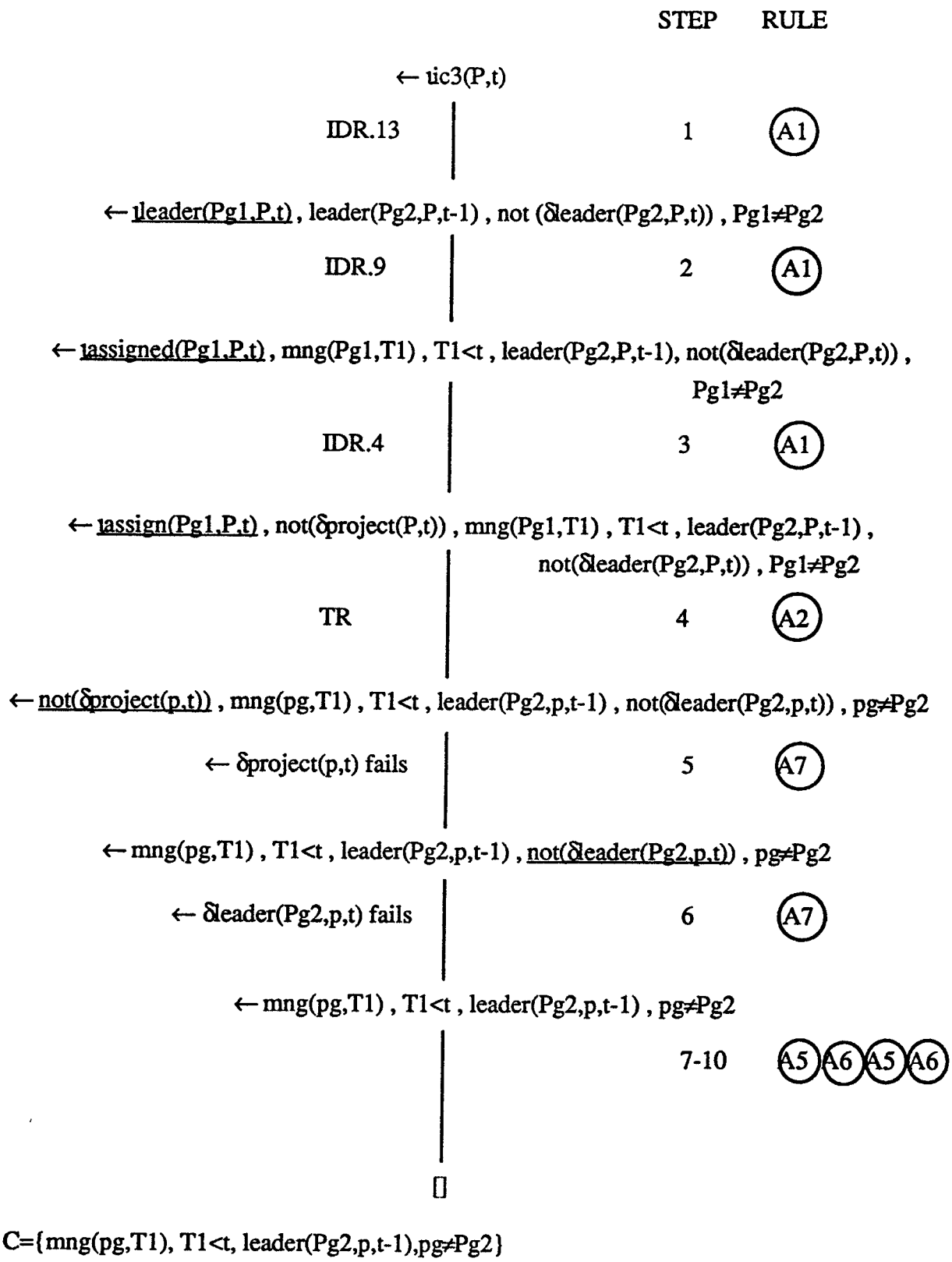

Figure 4

Steps 1,2 and 3 are SLDNF resolution steps, using rules of IEM as input clauses. At step 2 we

have several alternative rules to resolve with. IDR.8 or IDR.10 could have been used instead of IDR9, and each one of them may lead to a different set of conditions C. Nevertheless, in this concrete case, branches corresponding to IDR.8 and IDR.10 fail. At step 4 the selected literal is an external event literal ιassign(Pg1,P,t) that can be resolved with the base fact in TR. At steps 5 and 6 the selected literals are not(δproject(p,t)) and not(δleader(Pg2,p,t)) respectively. In order to get a successful derivation, SLDNF-search space must fail finitely for {←δproject(p,t)} ∪ IEM ∪ TR and for {←δleader(Pg2,p,t)} ∪ IEM ∪ TR. The failure for δproject(p,t) is shown in Figure 6, δleader(Pg2,p,t) fails in a similar way. From step 7 to 10 the selected literals correspond to stored predicates (mng(pg,T1)), to accessible predicates (leader(Pg2,p,t-1)), or to evaluable predicates (T1<t and pg≠Pg2). Observe that none of them can be evaluated at compilation time because they depend on the concrete values of the EDB facts at time t. As a consecuence, this literals are included in the condition set C and its evaluation is delayed until execution time.

The set of conditions obtained for ιic3(t) adds the following rule to the BT definition:
ιic3(p,t) ← mng(pg,T1),T1<T,leader(Pg2,p,t-1),pg≠Pg2
Rules for ιic1(p,t), ιic2(p,t), and ιic4(p,t) can be obtained similarly.

We also have to derive, for each stored predicate p(X,T), the set of conditions to be checked at execution time, in order induce an insertion or a deletion of a fact corresponding to this predicate, by partially evaluating IEM ∪ TR wrt ιp(X,t) and δp(X,t). Consider, for example, the partial evaluation for ιassigned(Pg,P,t). Possible sets of conditions C that would lead to the insertion of a base fact assigned(pg,p,t) are obtained by having some failed derivation of IEM∪TR∪{←ιassigned(Pg,P,t) } succeed. Figure 5 shows this derivation tree.

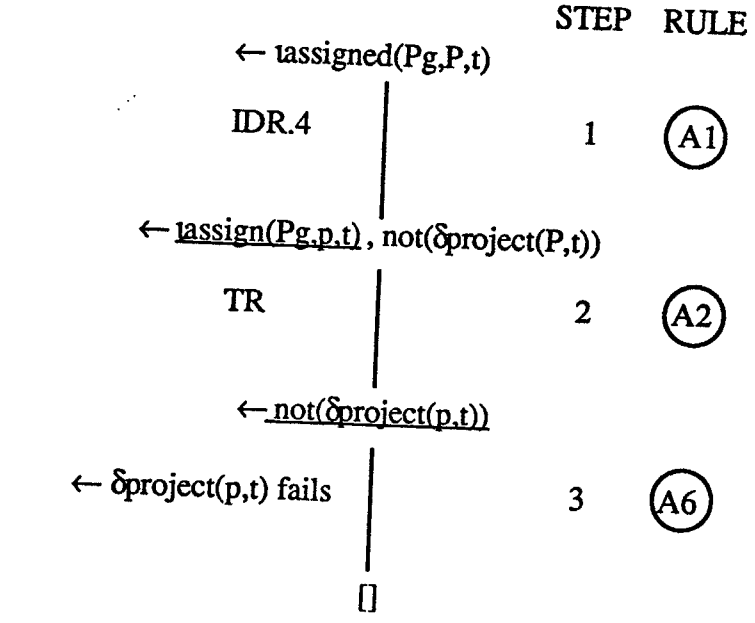

Figure 5

Steps 1 and 2 are SLDNF resolution steps. At step 1 rules of the IEM act as input clauses, while at step 2 the input clause is the base fact from TR. At step 3 the selected literal is a negation of an event literal not(δproject(p,t)). The failure for δproject(p,t) is shown in Figure 6. The derivation ends with the empty clause as a result and C={}. This means that the transaction ιassign(pg,p,t) induces inconditionally the insertion of a fact assigned(pg,p,t) to the EDB. As a consecuence, the following rule will be added to the BT definition:

ιassigned(pg,p,t) ←

Rules for ιproject(P,t), δproject(P,t), ιactive(P,t), δactive(P,t), ιassigned(Pg,P,t), and δassigned(Pg,P,t) can be obtained similarly.

Finally, Figure 6 shows the failure tree for δproject(p,t). Steps 1 and 2 are SLDNF resolution steps, where rules of the IEM act as input clauses. At step 3, the selected literal is the external event ιend(p,t) wich has to be resolved with facts in TR. Given that there is no fact ιend(p,t) in TR, the tree fails inconditionally.

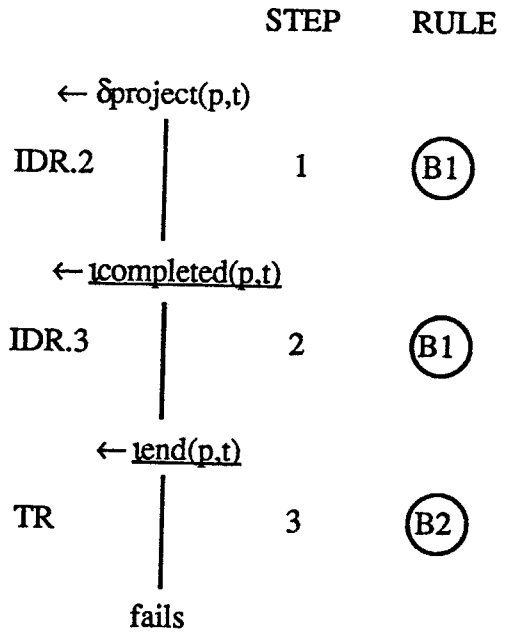

Figure 6

C'= {}

5.3 The method

This section describes, in a formal way, the method explained in sections 5.1 and 5.2. As we know, the transaction will be derived by partially evaluating IEM∪TR wrt internal events predicates corresponding to integrity constraints and stored predicates.

If I is the internal event predicate, C will be a set of conditions of I if there exists a **constructive derivation** from (I {}) to ([] C), having as input set IEM∪TR, where TR={ιb(**k**,t)}, being b a base predicate and **k** a vector of constants.

We will use the following notation:
- Non event literal is a literal corresponding to a base, derived or inconsistency predicate.
- Accessible literal is a non-event literal corresponding to an accessible predicate (AP).

Constructive derivation

A constructive derivation from $(I_1\ C_1)$ to $(I_n\ C_n)$ via a selection rule R, that selects literals not corresponding to evaluable predicates with priority, is a sequence:

$(I_1\ C_1)$, $(I_2\ C_2)$,..., $(I_n\ C_n)$

such that for each i>1, I_i has the form $\leftarrow L_1,...,L_k$, $R(I_i)=L_j$ and $(I_{i+1}\ C_{i+1})$ is obtained according to one of the following rules:

A1) If L_j is a positive internal event then $I_{i+1}=S$, and $C_{i+1}=C_i$, where S is the resolvent of some clause in IEM with I_i on the selected literal L_j.

A2) If L_j is a positive, external event literal, then $I_{i+1}=S$, and $C_{i+1}=C_i$, where S is the resolvent of fact in TR with I_i on the literal L_j.

A3) If L_j is a negative, external event literal "not(ιp(X,t))", and TR can not be unified with ιp(X,t) then $I_{i+1}= \leftarrow L_1 ,..., L_{j-1}, L_{j+1},..., L_k$, and $C_{i+1}=C_i$.

A4) If L_j is a negative, external event literal "not(ιp(X,t))", and TR can be unified with ιp(X,t) then $I_{i+1}= \leftarrow L_1 ,..., L_{j-1}, L_{j+1},..., L_k$, and $C_{i+1}=C_i \cup \{ X \neq k\}$.

A5) If L_j is an accessible literal then $I_{i+1}= \leftarrow L_1,... ,L_{j-1}, L_{j+1},..., L_k$, and $C_{i+1}=C_i \cup \{L_j\}$.

A6) If L_j is an evaluable literal then $I_{i+1}=\leftarrow L_1$,..., L_{j-1}, L_{j+1},..., L_k,and $C_{i+1}=C_i\cup\{L_j\}$.

A7) If L_j is a ground internal event literal "not(Q)" and there exists a **consistency derivation** from ($\{\leftarrow Q\}$ { }) to ({ } C') then $I_{i+1} = \leftarrow L_1,..., L_{j-1}, L_{j+1},..., L_k$, and $C_{i+1} = C_i\cup C'$.

A8) If L_j is a non-ground internal event literal "not(Q)" then $I_{i+1} = \leftarrow L_1,..., L_{j-1}, L_{j+1},..., L_k$, and $C_{i+1} = C_i\cup \{L_j\}$.

The step corresponding to rule A1) is an SLDNF resolution step. At steps corresponding to rules A2), A3) and A4), external event literals are evaluated (remember that we have the transaction as input set). This is done in order to have a failed derivation of IEM $\cup$ TR $\cup$ $\{\leftarrow I\}$ succeed. At steps A5), A6) and A8) accessible, evaluable and non-ground negative internal event literals are added to the condition set C. This is also done in order to have a failed derivation of IEM $\cup$ TR $\cup$ $\{\leftarrow I\}$ succeed. In case A7), the set of conditions necessary to ensure the failure of negative internal event literals are added to the condition set C

There are different ways in which a constructive derivation can succeed. Each one may lead to different insertion or deletion rule.

Consistency derivation

A consistency derivation from (F_1 C_1) to (F_n C_n) via a safe selection rule R, that selects literals not corresponding to evaluable predicates with priority, is a sequence:

(F_1 C_1), (F_2 C_2),..., (F_n C_n)

such that for each i>1, F_i has the form $\{\leftarrow L_1,..., L_k\} \cup F'_i$ and *for some* j=1...k, (F_{i+1} C_{i+1}) is obtained according to one of the following rules:

B1) If L_j is a positive internal event literal then $F_{i+1}= S' \cup F'_i$ where S' is the set of all resolvents of clauses in IEM with $\leftarrow L_1,..., L_k$ on the literal L_j, and $C_{i+1}=C_i$.

B2) If L_j is a positive external event literal, and TR can not be unified with L_j then $F_{i+1}=F'_i$, and $C_{i+1}=C_i$.

B3) If L_j is a positive external event literal then $F_{i+1} = S' \cup F'_i$ where S' is the resolvent of TR with $\leftarrow L_1,..., L_k$ on the literal L_j, and $[] \notin S'$, and $C_{i+1}=C_i$.

B4) If L_j is a negative external event literal "not(ιp(X,t))" and TR can be unified with ιp(X,t) then $F_{i+1} = F'_i$, and $C_{i+1}=C_i$, where S is the resolvent of TR with I_i on the literal L_j.

B5) If L_j is a negative external event literal "not(ιp(X,t))", and TR can not be unified with ιp(X,t) then $F_{i+1} = \leftarrow L_1,..., L_{j-1}, L_{j+1},..., L_k \cup F'_i$

B6) If L_j is an accessible non-event literal then $F_{i+1}=F'_i$, and $C_{i+1}=C_i \cup\{not(L_j)\}$.

B7) If L_j is an evaluable literal then $F_{i+1} =F'_i$, and $C_{i+1} =C_i\cup\{not(L_j)\}$.

B8) If L_j is a negative internal event literal "not(Q)" and there exists a constructive derivation from ({←Q} { }) to ([] C') then $F_{i+1}=F'_i$, and $C_{i+1}= C_i \cup C'$.

Steps corresponding to rules B1) and B3) are SLDNF resolution steps. In case B2) and B4) the current branch fails and thus, it can be eliminated. In case B5) the selected literal can not fail, and thus, it is eliminated from the subgoal set F, in order to make a successful SLDNF branch fail. In case B6) and B7) the negation of evaluable or non-event literals is added to the condition set C, also in order to make a successful SLDNF branch fail. This branch can be dropped because we ensure the failure for it. Finally, in case B8) the current branch can be dropped if there exists a constructive derivation for the negation of the selected literal. This ensures failure for it.

5.4 The Transaction Manager

Once we have described the transaction generation method, we describe now how Base Transactions are treated at execution time by the Transaction Manager.

When an external event occurs in the real world, it is received by the UIMS. If the event is a predefined transaction, rules specified in the corresponding BT have to be evaluated by the Transaction Manager. As we have seen before, BT rules follow the general pattern:

ιicj(x,Y,t) ← C(x,Y,t) for the integrity constraints, and
ιp(x,Y,t) ← C(x,Y,t)
δp(x,Y,t) ← C(x,Y,t) for stored predicates

At execution time, constants x and t take its values from transaction parameters, and the body C can be completely evaluated given that facts stored in the EDB are available at this time.

After receiving the transaction from the UIMS, and once the corresponding BT rules are known, the Transaction Manager acts in the following way:

1- For each rule ιicj(x,Y,t), checks if there exists any value y of Y such that C(x,y,t) evaluates to true, this means that the transaction violates the integrity constraint icj. In this case, an error message is sent to the UIMS and the transaction execution is stopped.

2- For each rule ιp(x,Y,t), find all possible values y of Y such that C(x,y,t) evaluates to true, and for all of them three cases are possible:

a) If p is stored "current" and its type is P-state or P-spontaneous:
INSERT tuple p(x,y) into the EDB.

b) If p is stored "full history" and its type is P-transient:
INSERT tuple p(x,y,t) into the EDB.

c) If p is stored "full history" and its type is P-state or P-spontaneous:
INSERT tuple p(x,y,t,∞) into the EDB.

3- For each rule δp(x,Y,t), find all possible values y of Y such that C(x,y,t) evaluates to true, and for all of them two cases are possible:

a) If p is stored "current" and its type is P-state or P-spontaneous:
DELETE tuple p(x,y) from the EDB.

b) If p is stored "full history" and its type is P-state or P-spontaneous:
SET tuple p(x,y,ts,∞) to p(x,y,ts,t)

(Note that, by definition, P-transient predicates are not deleted).

We have implemented the Transaction Manager using the Prolog language.

6. THE ODISSEA DESIGN ENVIRONMENT

We have seen in Section 4 the concept of Internal Events Model, and we have discussed, in the previous Section, the role it plays in transaction generation. We now briefly describe our design environment and show the context in which transaction generation is done. Figure 7 shows the five main components of the ODISSEA design environment.

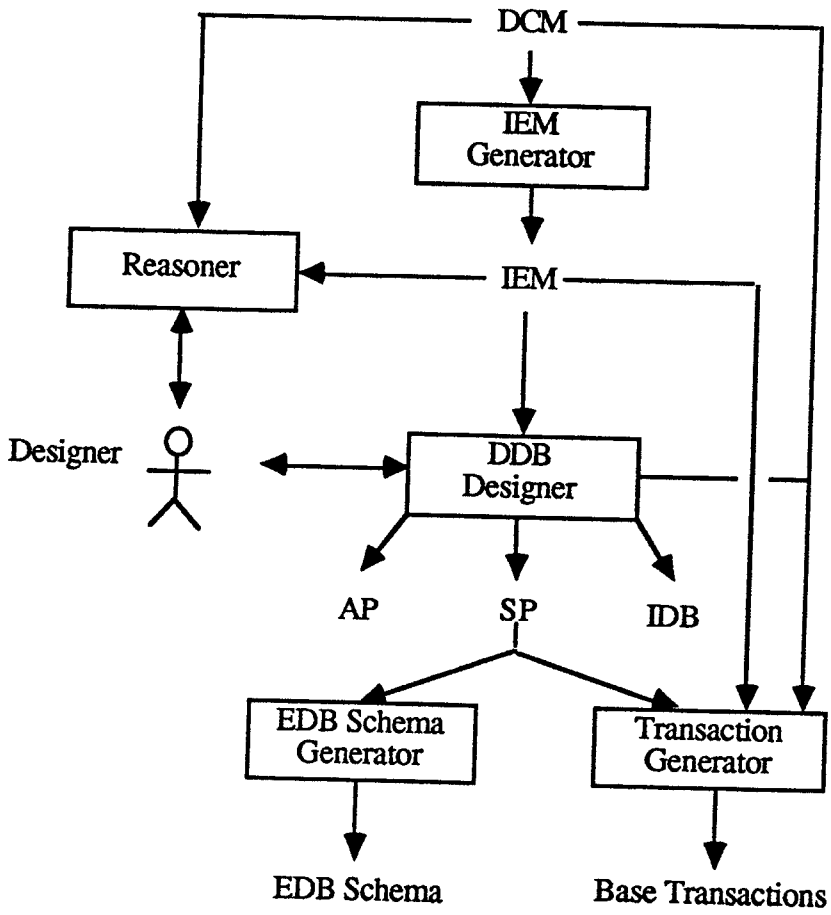

Figure 7. The ODISSEA design environment

The IEM Generator derives the IEM corresponding to the DCM, by using the procedure described in Section 4.

The Reasoner is the component of our environment for DCM validation. The Reasoner can answer two kinds of queries from the designer:

a) Given an initial and target states of the Information Base, together with a sequence of external events, check whether the sequence is able to perform the transition between both states.

b) Given an initial and target states, obtain one or more sequences of external events (plans) to perform the transition between both states.

The Reasoner uses SLDNF resolution as proof procedure, and plan generation techniques developed in AI. See [CoO91] for details on the method in which is based, as well as on the use of the Reasoner in DCM validation. Generating prototypes from a DCM is also possible and useful during the model validation.

The Deductive Database (DDB) designer is the component that generates the set of Stored Predicates (SP), the set of Accessible Predicates (AP) and the IDB rules.

Usually, there are many valid alternatives of SP sets and, thus, the designer must choose the one that he/she considers most appropiate. An SP set is characterized by:

- A set of base or derived predicates of the DCM to be explicitly stored in the EDB.
- For each of them, a time interval for which its facts will be stored.

We consider two cases of time intervals: current state or full history. If p(X,t) is a stored predicate, in the first case we store the set of facts $\{<X> \mid p(X,t)\}$ where t is the current time, and in the second case we store the set $\{<X,t1> \mid p(X,t1) \wedge t1 \leq t\}$.

A valid SP set is one that satisfies:

1) All internal events that happen at time t can be derived from the stored facts at time t-1 and the external events received at t.
2) The answer to all predefined queries can be derived from the stored facts.

Both conditions can be checked using SLDNF resolution with IEM and DCM as input sets. Figure 8 illustrates the checking that query q3(M,t) can be derived from the stored facts in our example. Literals in boxes correspond to literals that can be unified with stored facts or to evaluable predicates. Thus, in the second goal, literal assigned(P,Pg,t) can be resolved with the stored facts, since we store the full history of predicate assigned.

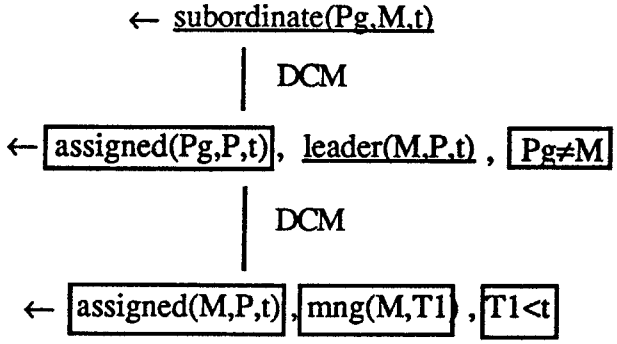

Figure 8 Checking derivability of q3.

Note that the alternative of storing the full history of all base predicates is always a valid one [Oli89].

Once the SP set has been designed, we determine the AP set, that is, the predicates whose facts can be derived from the stored facts. We use a procedure similar to the one described above for this purpose.

Finally, we generate the IDB rules with a simple transformation of the DCM.

The EDB Schema Generator writes the EDB schema for the set SP. For each current state predicate we define a relation scheme with the same attributes, except the time attribute, since the time will be implicit. For each P_state or P_spontaneous full history predicate p(X,T) we define a relation scheme p(X,T_start,T_end), where attributes T_start and T_end represent the interval in wich p(X) holds. For each P_transient full history predicate p(X,T) we just define a relation scheme p(X,T).

The Transaction Generator is the component that generate the Base Transactions, as described in the previous section.

7. CONCLUSIONS

We have presented the ODISSEA approach to the design of IS from a DCM. We have discussed the main decisions involved in this process: data base design and transaction design. Data base design can not be completely automated, but we provide some tools that help the designer in alternatives generation and analysis.

Transaction design can be completely automated, assuming a given hardware/software execution environment. We have presented in detail a formal method to derive the transactions from the DCM. The method is based on the use of the SLDNF proof procedure and, thus, it can be implemented easily in Prolog. We hope that our method will facilitate the implementation of DCM and, thus, it may help to overcome the main disadvantage of the deductive approach.

We plan to build an additional component in our design environment, which transforms our Base Transactions into a procedural language. In such a way, we can improve the performance of the generated system.

Acknowledgment

The authors wish to thank D. Costal, E. Mayol, J.A. Pastor, C. Quer, J. Sistac, E. Teniente and T. Urpí for their comments and suggestions on earlier drafts of this paper. This work has been supported by the CICYT PRONTIC program, project TIC 680.

REFERENCES

[Bub86] Bubenko,J.A. "Information system methodologies - A research view". In [OSV86], pp. 289-318.

[BuO86] Bubenko,J.A.;Olivé,A. "Dynamic or temporal modelling?. An illustrative comparison". SYSLAB Working Paper No. 117, University of Stockholm, 1986.

[CGT90] Ceri,S.;Gottlob,G.;Tanca,L. "Logic programming and databases", Springer-Verlag, 1990.

[CKM91] Chung,L.;Katalagarianos,P.;Marakakis,M.;Mertikas,M.;Mylopoulos,J.; Vassiliou,Y. "From information system requirements to design: A mapping framework", Information Systems, Vol.16, No.4, 1991, pp. 429-461.

[CoO91] Costal,D.;Olivé,A. "A method for reasoning about deductive conceptual models of information systems", Technical Report, Univ. Pol. Catalunya, Barcelona, 1991.

[Fre85] Frenkel,K.A. "Toward automating the software development cycle". Comm. of the ACM, Vol.28,No.6,June 1985, pp. 578-589.

[GMN84] Gallaire,H.;Minker,J.;Nicolas,J-M. "Logic and databases: A deductive approach". ACM Computing Surveys, Vol. 16, No. 2, June 1984, pp. 153-185.

[LIS91] Lloyd,J.;Shepherdson,J.C. "Partial evaluation in logic programming". J. Logic Programming, 1991, No.11, pp. 217-242.

[LIT84] Lloyd,J.W.;Topor,R.W. "Making Prolog more expressive". J. Logic Programming, 1984, No.3, pp. 225-240.

[GKB82] Gustafsson,M.;Karlsson,T.;Bubenko,J.A. "A declarative approach to conceptual information modelling". In [OSV82], pp. 93-142.

[MBJ90] Mylopoulos,J.;Borgida,A.;Jarke,M.;Koubarakis,M. "Telos: Representing knowledge about information systems", ACM Trans. on Information Systems, Vol.8,No.4, Oct.1990, pp 325-362.

[Oli82] Olivé, A. "DADES: A methodology for specification and design of information systems". In [OSV82], pp. 285-334.

[Oli86] Olivé,A. "A comparison of the operational and deductive approaches to conceptual information systems modelling". Proc. IFIP 86, North-Holland, Dublin, 1986, pp. 91-96.

[Oli89] Olivé,A. "On the design and implementation of information systems from deductive conceptual models". Proc. of the15th. VLDB, Amstedam, 1989, pp. 3-11.

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

[OSV82] Olle,T.W.;Sol,H.G.;Verrijn-Stuart,A.A. (Eds.) "Information systems design methodologies: A comparative review". North-Holland, 1982.

[OSV86] Olle,T.W.;Sol,H.G.;Verrijn-Stuart,A.A. (Eds.) "Information systems design methodologies: Improving the practice". North-Holland, 1986.

[QuS91] Quer,C.;Sistac,J. "ODISSEA: a language for deductive information systems". Proc. 2nd. Intl. Workshop on the Deductive approach to IS and DB, Aiguablava(Catalonia), 1991, pp. 22-49.

[San90] Sancho,M.R. "Deriving an internal events model from a deductive conceptual model". Proc. Intl. Workshop on the Deductive approach to IS and DB, S'Agaró (Catalonia), 1990, pp. 73-92.

[TeO92] Teniente,E.;Olivé,A."The events method for view updating in deductive databases", Proc. EDBT92, Vienna, Austria, 1992.

Departament de Llengatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

List of research reports (1993).

LSI–93–1–R "A methodology for semantically enriching interoperable databases", Malú Castellanos.

LSI–93–2–R "Extraction of data dependencies", Malú Castellanos and Fèlix Saltor.

LSI–93–3–R "The use of visibility coherence for radiosity computation", X. Pueyo.

LSI–93–4–R "An integral geometry based method for fast form-factor computation", Mateu Sbert.

LSI–93–5–R "Temporal coherence in progressive radiosity", D. Tost and X. Pueyo.

LSI–93–6–R "Multilevel use of coherence for complex radiosity environments", Josep Vilaplana and Xavier Pueyo.

LSI–93–7–R "A characterization of $PF^{NP\|} = PF^{NP[\log]}$", Antoni Lozano.

LSI–93–8–R "Computing functions with parallel queries to NP", Birgit Jenner and Jacobo Torán.

LSI–93–9–R "Simple LPO-constraint solving methods", Robert Nieuwenhuis.

LSI–93–10–R "Parallel approximation schemes for problems on planar graphs", Josep Díaz, María J. Serna, and Jacobo Torán.

LSI–93–11–R "Parallel update and search in skip lists", Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer.

LSI–93–12–R "On the power of equivalence queries", Ricard Gavaldà.

LSI–93–13–R "On the learnability of output-DFA: a proof and an implementation", Carlos Domingo and David Guijarro.

LSI–93–14–R "A heuristic search approach to reduction of connections for multiple-bus organization", Patricia Ávila.

LSI–93–15–R "Toward a distributed network of intelligent substation alarm processors", Patricia Ávila.

LSI–93–16–R "The Odissea approach to the design of information systems from deductive conceptual models", Maria Ribera Sancho and Antoni Olivé.

Internal reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics (U.P.C.)
Pau Gargallo 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es