DATABASE CASE TOOL ARCHITECTURE : PRINCIPLES FOR FLEXIBLE DESIGN STRATEGIES

J-L Hainaut, M. Cadelli, B. Decuyper, O. Marchand

Institut d'Informatique - University of Namur
rue Grandgagnage, 21 - B-5000 Namur (Belgium)
jlh@info.fundp.ac.be

Abstract. The paper describes the architectural principles of a database CASE tool that allows more flexible design strategies than those of traditional tools that propose oversimplistic *draw-and-generate* approaches. Providing this flexibility is based on four basic principles, namely a *unique generic specification model* that allows the definition of a large variety of specific design products, *transformational functions* as major database design tools, a *toolbox architecture*, allowing a maximal independence between functions, and *multiple model definition* through parametrization of the unique generic model. These architectural characteristics themselves derive from two fundamental paradigms, namely the *process-product-requirements* approach to model design behaviours, and the *transformational approach* to system design.

Keywords : design modeling, system design, database design, transformational approach, CASE tools.

1. INTRODUCTION

Most current database CASE tools provide four major families of functions for their users, namely conceptual specification entry, conceptual specification validation, reporting and executable code generation. They concentrate mainly on conceptual specification acquisition, leaving the problem of producing efficient physical schemas practically unsolved. They are based on simple and rigid strategies that give the illusion that database design is a straighforward and deterministic process once the conceptual schema has been developed. Indeed, many CASE tools propose a *draw-and-generate* strategy that leads to poor and sometimes unreadable DBMS schemas. More sophisticated strategies that would allow the integration of more realistic requirements (time efficiency, space efficiency, distribution, privacy, modularity, hardware constraints, etc) are impossible. That leads designers to two unacceptable practices : integrating these requirements into the conceptual schema, or modifying the generated DDL text with a text processor.

TRAMIS is an experimental CASE tool that proposes a different approach through which both novice and skilled designers can produce database schemas according to their own strategies, ranging from draw-and-generate to fine-tuning. In particular, it offers its users both a high degree of flexibility, and the possibility to restrict it when needed.

The paper describes the architectural principles of TRAMIS. It describes the four basic principles that give it this flexibility, namely a *unique generic specification model* that allows the definition of a large variety of specific design products, *transformational functions* as major database design tools, a *toolbox architecture*, allowing maximal independence between functions, and *multiple model definition* through parametrization of the unique generic model. These architectural characteristics derive from two fundamental paradigms, namely the *process-product-requirements* model of design behaviour, and the *transformational approach* to system design.

However, the paper will not concentrate on database design methods in particular, nor on the detailed description of a specific CASE tool.

The paper is organized as follows. Section 2 develops the concept of design modeling, i.e. the description of how designers behave or should behave. Section 3 analyzes some important aspects of the transformational approach applied to database design. The other sections develop the four architectural principles of TRAMIS, namely the unique specification model (section 4), the transformation functions (section 5), the functions and architecture of the CASE tool (section 6), and the model specialization as a tool to define the skeleton of specific strategies (section 7).

2. MODELING THE DESIGN ACTIVITY

The design of any technical system is a complex task that is broken down into smaller design activities in such a way that each of them can be dedicated to solving specific design problems or satisfying specific requirements. The nature of these activities, the requirements that are taken into account and the way these activities can be carried out (partly) define *a design method* (SSADM, JSD, YOURDON, MERISE, NIAM are examples of software engineering design methods). We are dealing with the general properties that several design methods share, i.e. with some kind of *generic design model.* In the realm of software systems, and more specifically databases, two kinds of generic design models can be put forward, namely the *multilevel approach* and the more flexible *process-product-requirements model.* These models try to state how designers must behave (forward engineering), or even to describe how designers behave in practice (reverse engineering [6]).

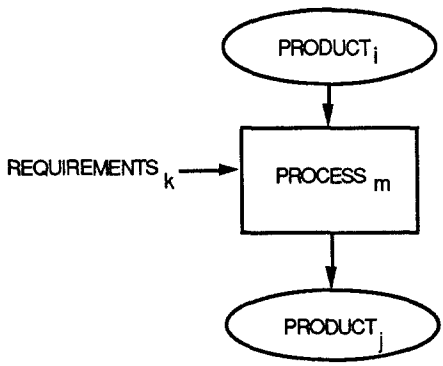
The **multilevel approach** relies on a fixed hierarchy of levels of specification description, characterized mainly by their degree of independence according to the implementation tools. Each level is defined by a set of target documents, and by the design activities that produce these documents. Most current practical database design approaches offer three levels, namely the conceptual, logical and physical levels [16] [15].

The **process-product-requirements model** proposes a more flexible approach according to which a design is carried out by design processes that transform products in order to make them satisfy requirements (Figure 1). A *product* is any standard specification set that is considered as significant in the design method in concern. A *process* is an activity that is aimed at producing a set of products from another set of products[1]. Any process is goal-oriented, that is it must give the output products properties that the input products don't satisfy. In most cases, the objective is meeting specific *requirements* such as normalisation, readableness, space efficiency, time efficiency, DBMS compliance, organizational constraints, hardware limitation, parallelism, user's skill, etc [2].

The way a process is carried out is defined by a design *strategy*, that can be based on formal rules or on heuristics. Applying the strategy generally implies carrying out other, lower-level processes. The latter are in turn design processes, and therefore produce output products from input products according to strategies as well, and so on (Figure 2). The output products of these low-level processes satisfy requirements that are either the requirements of the higher-level process, or specialization thereof (e.g. time efficiency requirements can be translated (as suggested by T_{km} in Fig. 2) into maximum paging rate requirements in some physical design processes).

---

[1] Note that a product can be seen as a specific state of another product as well : an un-normalized schema and a normalized schema can be seen either as two products, or as two states of the same product; in the scope of this paper, both views are considered as equivalent.



$$PRODUCT_j = PROCESS_m(PRODUCT_i, REQUIREMENTS_k)$$

Fig. 1. The process-product-requirements design model.

A strategy may use internal products that are not known at the higher level. Note that the multilevel approach can be seen as a specialization of this framework.

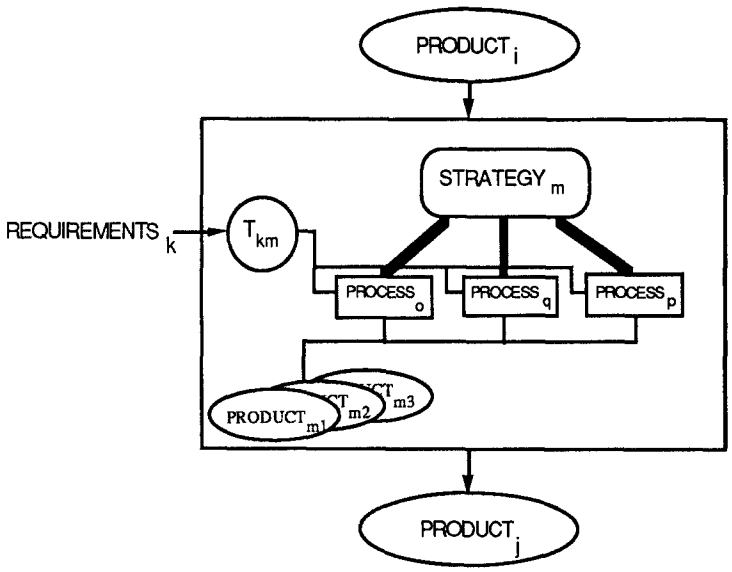


Fig. 2. A design process is defined by a strategy that describes how to produce $PRODUCT_j$ from $PRODUCT_i$. The strategy may use lower-level design processes, internal products and specialization of requirements.

3. THE TRANSFORMATIONAL APPROACH TO DATABASE DESIGN

3.1 Schema transformation

Modeling software design as the systematic transformation of formal specifications into efficient programs (and building CASE tools that support it) has long been considered as one of the ultimate goal of the research in software engineering [12]. Some important research programmes are dedicated to this approach (e.g. PROSPECTRA [10]). According to the former reference cited, *a transformation is a relation between two program schemes P and P' (a program scheme is the* [parametrized] *representation of a class of related programs*; a program of this class *is obtained by instantiating the scheme parameters). It is said to be correct if a certain semantic relation holds between P and P'.* These definitions still hold for database schemas, that are special kinds of abstract program schemes. The concept of transformation is particularly attractive in this realm, though it has not often been made explicit (for instance as a user tool) in current CASE tools. A (schema) transformation is most generally considered as an operator by which a data structure S1 is replaced by another structure S2 which has some sort of equivalence with S1. Schema transformation is a ubiquitous concept in database design. Proving the equivalence of schemas [9], refining a conceptual schema, integrating two partial schemas [1], producing a DBMS-compliant schema from a conceptual schema [13], restructuring a physical schema, DB reverse engineering [6], are basic design activities that can be carried out by carefully chosen schema transformations .

Though developing this concept and its formalization is beyond the scope of this paper (see [7] for a more formal treatment), we shall sketch the main definitions and properties that will be important from the methodological viewpoint.

A **transformation T** is an operator that replaces a source construct C in schema S by another construct C'; C' is the target of C through T, and is noted C' = T(C)
A transformation T is defined by , (1) a precondition P that any construct C must satisfy in order to be transformed by T, (2) a maximal postcondition[1] Q that T(C) satisfies. T can therefore be written T = <P,Q> as well. P and Q are pattern-matching predicates that identify the components and the properties of C and T(C), and more specifically :

- the components of C that are preserved in T(C),
- the components of C that are discarded from T(C),
- the components of T(C) that didn't exist in C.

As an alternative to this predicative specification, [12]) proposes procedural rules that define the removing, inserting, renaming operations that produce the target schema when applied to the source schema. In the TRAMIS tool, the predicative specifications have been translated into procedural rules. In some cases, (i.e. when the graphical language is powerful enough), it is possible to give a more readable representation of a transformation by expressing C and T(C) graphically (Figures 3 and 4). A transformation T1 = <P,Q> is *reversible*[2], or *semantics-preserving, iff* there exists a transformation T2 such that, for any schema S,

---

[1] The terms pre- and postcondition must not be confused with those used in programming theory (by Hoare and Dijkstra for instance). In the present context, predicates P and Q includes applicability preconditions but also terms that identify all the objects directly or undirectly implied in the transformation. This latter part is not relevant in programming theory.

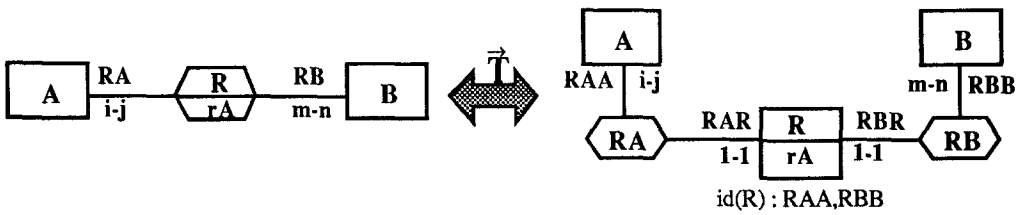
[2] In fact, the issue is a bit more complex, since a transformation must be defined not only by a mapping T between schemas but also by a mapping t between data populations (instances) of the schemas. Two kind of semantics preservation can be defined, namely *reversibility* and *symmetrical reversibility* [7]. T1 is reversible iff, *for any instance s of schema S such that P(S), s = t2(t1(s))*; T1 is

$$P(S) \Rightarrow T2(T1(S)) = S$$

T2 is the reverse of T1, and conversely. We have the following property : T2 = <Q,P>

3.2 Generic *vs* specific transformation

Let's consider one of the most popular reversible transformations that replaces a many-to-many relationship type by a simpler construct based on two one-to-many relationship types. Figure 3 gives both a graphical representation and the predicative representation of this transformation[1].




```
P = entity-type(A) & entity-type(B) & rel-type(R)
    & role(R,RA,A,[i,j]) & role(R,RB,B,[m,n]) & att(R,[aR])
Q = entity-type(A) & entity-type(B) & entity-type(R)
    & rel-type(RA) & rel-type(RB)
    & role(RA,RAA,A,[i,j]) & role(RA,RAR,R,[1,1])
    & role(RB,RBB,B,[m,n]) & role(RB,RBR,R,[1,1])
    & att(R,[aR]) & id(R,[RAA,RBB])
```

Fig. 3. Graphical and logic-based expression of a generic transformation. In the logic-based expression, predicate **entity-type**(X) means that X is the name of an entity type, predicate **role**(R,X,E,[i,j]) means that relationship type R has a role X, taken by entity type E, and with cardinality constraint [i,j], predicate **att**(R,[X,Y,..]) means that entity/relationship type R has attributes X,Y,.., and predicate **id**(X,[A,B,..]) means that entity/relationship type X has an identifier made up of {A,B,..}.

The transformation is defined for objects the names of which are generic (R, A, B, RAA, etc); therefore, it defines a *class of transformations* (a sort of *program scheme* according to [12]). Replacing these names by actual names gives a *specific transformation* (see Figure 4).

Schema transformation is a ubiquitous concept in database design, and appears explicitly or implicitly in many modeling and design activities (e.g. top-down design, reverse engineering, view integration, multibases). However, it has not been often studied as an design process of its own. Let's only mention [3], [9], [11], [14], [7] as some proposals in this direction.

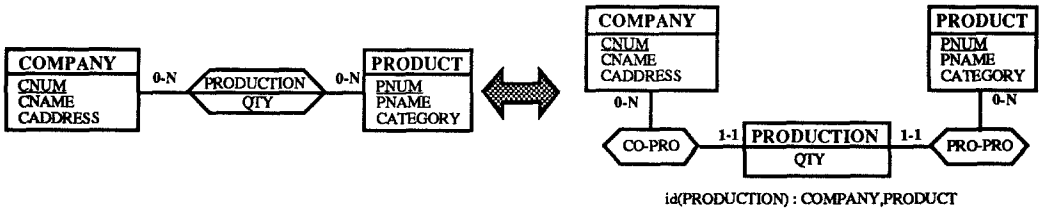
3.3 Transformations in design processses

Schema transformations are not really design processes by themselves, but rather basic tools that can be used to carry out such processes.

---

symmetrically reversible iff both T1 and T2 are reversible, i.e., in addition to the property mentioned above, *for any instance s' of S' such that Q(S'), s' = t1(t2(s'))*. For simplicity, we shall ignore this distinction in this paper.

[1] To be quite precise, some parts are missing in predicate Q. In particular, R cannot have a role in other relationship types than RA and RB. Without these parts T is not reversible. See [7] for more details.




```
P =  entity-type(COMPANY) & entity-type(PRODUCT) & rel-type(PRODUCTION)
     & role(PRODUCTION,COMPANY,COMPANY,[0,N])
     & role(PRODUCTION,PRODUCT,PRODUCT,[0,N])
     & att(PRODUCTION,[QTY])
Q =  entity-type(COMPANY) & entity-type(PRODUCT)
     & entity-type(PRODUCTION) & rel-type(CO-PRO) & rel-type(PRO-PRO)
     & role(CO-PRO,COMPANY,COMPANY,[0,N])
     & role(CO-PRO,PRODUCTION,PRODUCTION,[1,1])
     & role(PRO-PRO,PRODUCT,PRODUCT,[0,N])
     & role(PRO-PRO,PRODUCTION,PRODUCTION,[1,1])
     & att(PRODUCTION,[QTY]) & id(PRODUCTION,[COMPANY,PRODUCT])
```

Fig. 4. Graphical and logic-based expression of a specific transformation. It has been obtained by substituting actual data type names for the generic ones.

Studying the problem of design strategies in some details is beyond the scope of this paper, however the role of transformations in a design process can be grossly sketched as follows :

Let R be the set of requirements (expressed as a set of predicates or rules) of process P,
S the input schema of the process,
C a construct of schema S,
r a rule of R such that : $\neg$ r(C)
$\mathbb{T}$ the set of available transformations.

C is a construct of schema S that does not satisfy requirements R, and that must be transformed.

An obvious elementary strategy is as follows :

(1) select a transformation T of $\mathbb{T}$ such that $P_T(C)$ & $(Q_T \Rightarrow r)$
(2) replace C by T(C) in S

Potential problems may arise that require more sophisticated strategies. Let's examine some of them.

P1 : Construct C may violate more than one rule.
Strategy : Let R' be the set of rules that C doesn't satisfy. Choose a rule r in R' such that there exists a transformation T in $\mathbb{T}$ such that : T(C) violates as few rules of R as possible. This strategy may generate a set of solutions[1].

P2 : More than one transformation satisfies : P(C) & (Q $\Rightarrow$ r)
Strategy : the selection of T can be done either arbitrarily, or according to other rules. In the latter case, P generates a set of solutions. The final selection will be done according to other kinds of requirements, i.e. in another design process.

---

[1] In all generality, there will be a tree of solutions, since more than one design process may encounter this problem.

P3 : No transformations satisfy P(C)
Diagnostic : either the transformation set $\mathbb{T}$ is not powerful enough or the requirement cannot be satisfied.
Strategies : extend the set of transformations, keep construct C as it is or discard C.

P4 : No transformations satisfy : $Q \Rightarrow r$
Strategy : choose a transformation T such that $P_T(C)$, then select another transformation T' such that : $(Q_T \Rightarrow P_{T'})$ & $(Q_{T'} \Rightarrow r)$; if the latter cannot be satisfied, iterate the process. This strategy may generate a set of solutions.

P5 : T(C) violates a rule that C satisfied
This problem can be local, i.e. it concerns the termination of P, or it can be global, such as when P may destroy the effect (i.e. the satisfaction of a set of requirements) of a former process. Though it has been solved in TRAMIS with adhoc strategies, this problem still has no general solution and is currently under investigation.

These problems may induce the production of a large solution space. In such a situation, the concept of *output products* (PRODUCTj in Figure 1) must be replaced by that of *set of equivalent output products* that must be explored according to other criteria. A higher-level strategy must be defined to manage this space and reduce it to one solution.

4. THE UNIQUE SPECIFICATION MODEL OF TRAMIS

According to the generic design model developed in section 2, designing a database is decomposed into several processes, starting from, say, requirements collection, and ending with DDL schema generation[1]. These processes define a set of standard products, such as the validated conceptual schema or the executable DDL schema. This set of products can also be perceived as a set of states of the database schema. According to this view, designing a database consists in applying the design processes to a source schema (general a conceptual schema) in order to transform it into an executable database schema that satisfies a given set of requirements such as semantic correctness, readableness, efficiency and executability.

TRAMIS is based on a *unique generic model* to express database structures in any of its possible states. This unique model allows a neutral definition of the design processes, as well as a high flexibility in the design strategies used by the designers. For instance, a conceptual transformation can be used on a physical schema as well. The TRAMIS model is an extension of the Entity-Relationship model that can express both conceptual and technical structures. Its formalization can be found in [5]; however, we will follow a more intuitive approach in this section.

In addition, the design activities need richer specifications than mere structural description of the data. For instance, statistical information is needed for choosing adequate physical parameters, or for evaluating the volume and the response time of the database. Descriptive textual information and annotations are essential to define the semantics of the data objects, and to document the design process itself.

The *TRAMIS specification model* is made up of six generic objects classified into high-level objects that define macrostructures (schema, entity type, relationship type and space) and the

---

[1] In all generality, the database life-cycle is broader than suggested. For instance, phases such as schema reverse engineering, or schema and data maintenance according to changes in the requirements must be taken into account as well. It has been proved that the paradigms that underly TRAMIS and that are presented is this paper are well suited to support these phases [6].

low-level objects that define microstructures (attribute and group). Some objects (relationship type and group) can be given additional characteristics specifically aimed at describing technical or physical structures.

Schema : represents a description of a data base. Any number of entity types, relationship types and spaces can be associated with a schema. Schemas may be linked to other schemas according to specific relationships : *is the normalized version of, is a view of, is the ORACLE version of, is a space-efficient version of, is integrated into*, etc.

Entity type : represents any information unit that can be perceived or manipulated as a whole, at any level of the design process. It can be used to model both conceptual entities and technical objects (such as record types, tables, segment types, etc).

Relationship type : represents any significant aggregate of at least two entity types. Each position in the aggregate is called a role that is played by one or several entity types. The cardinality constraint of a role is a couple of integers specifying in how many (min-max) relationships an entity must and can play that role (N stands for ∞). It can be used to model both conceptual relationships and technical constructs (such as tuple linkage, CODASYL sets, parent/child relationships, file coupling, etc).

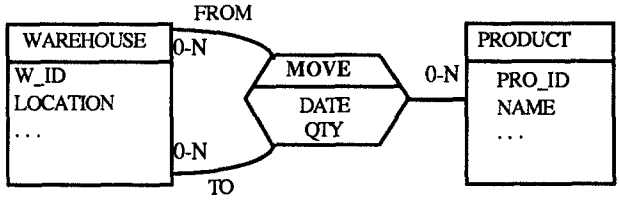


Fig. 5. Example of a relationship type of degree 3 with attributes

At a technical or programming level, a relationship can be perceived not only as a logical association between entities, but also as an access mechanism to navigate through (technical) entities; therefore, a binary relationship type can support zero, one or two *access paths*, each defined from the *origin* role toward the *target* role. This concept can be seen as an abstraction of CODASYL set types, IMS parent/child relationship, TOTAL paths, ADABAS file coupling, etc.

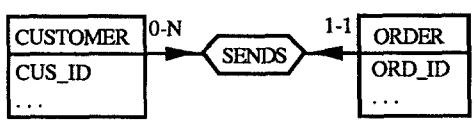


Fig. 6. A relationship type supporting two access paths

Space : is a collection of entities, possibly of different types. At the technical level, a space can model objects such as files, dbspaces, tablespaces, areas and datasets.

Attribute : is associated with a parent object, i.e. an entity type, a relationship type or a compound attribute. An attribute can be atomic or compound. It is qualified by a cardinality constraint expressed as a couple of integers stating how many (min-max) values can and must be associated with the parent object. Any number of attributes (including zero) can be associated with an entity type or a relationship type.

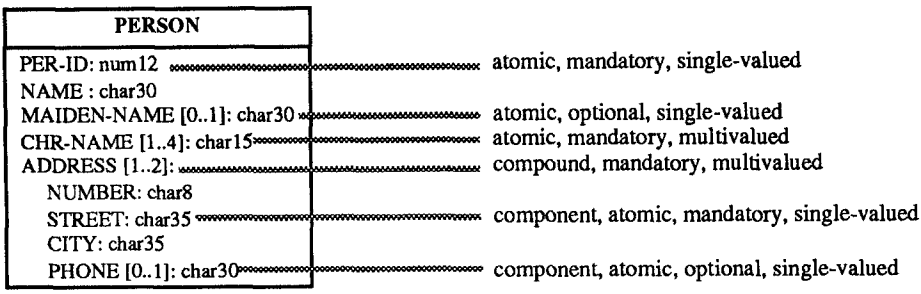


Fig. 7. Various types of attributes

Group : the group is a simple and powerful construct with which one can model entity and relationship identifiers, referential integrity, intra- and inter-entity attribute redundancy, logical access, statistical support, etc. A group is a collection of attributes and/or roles and is associated with an entity type or a relationship type. A group can have some specific functions regarding its parent entity or relationship type :

• it can be an **identifier** (in the example below, an EMPLOYEE is identified by its EMP-ID and its origin SUBSIDIARY, a MOVE relationship is identified by its date and its source and destination WAREHOUSES, a SCHEDULE is identified by TEACHER + TIME values and by TIME + PLACE values). . .

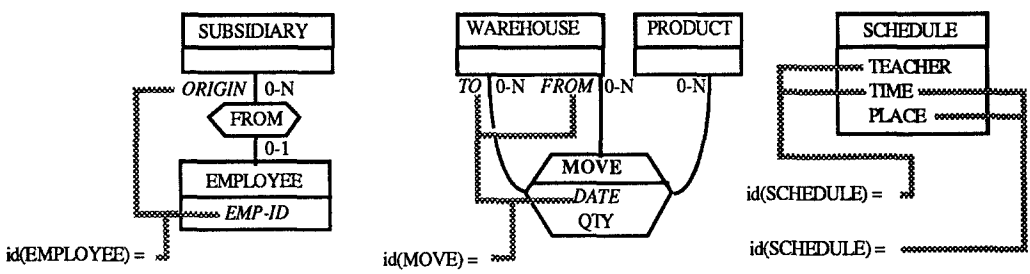


Fig. 8. Representation of identifiers as groups of attributes/roles

• a **reference** (inclusion, equality, copy) to other attributes (in the example below, SUBS-ID + EMPL-ID is a *foreign key* to EMPLOYEE) . . .

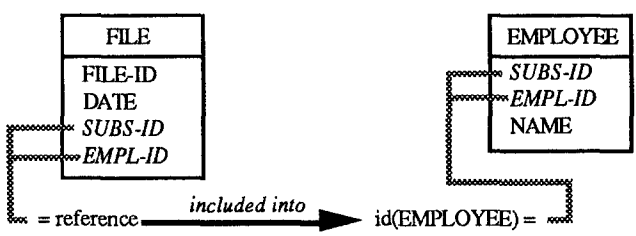


Fig. 9. Representation of an inclusion constraint (here a referential constraint) as the inclusion of a group into another one.

• or, at the technical level, an access mechanism called **access key** (the abstraction of index, calc keys, hash organization, etc). In the example below, FILE-ID is both an identifier and an

access key to FILE; DATE and the EMPLOYEE entity it is coming FROM constitute an access key to FILE (i.e. given an EMPLOYEE entity and a DATE value, one can gain a quick and selective access to the concerned FILE entities).

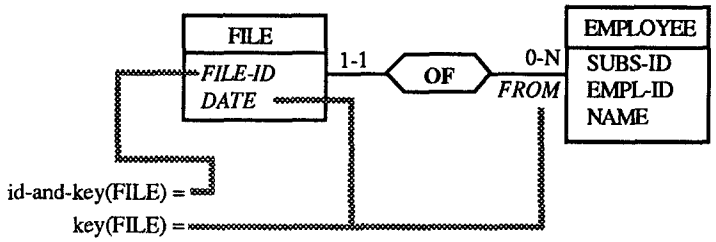


Fig. 10. Representation of an access key as a group of attributes/roles. This example is typical of CODASYL (indexed set types) and IMS structures.

Besides its intrinsic properties, a TRAMIS object can be characterized according to four facets : naming, structural, statistical, informal specification.

The ***naming facet*** names an object with up to four names[1].
- the *natural name* : the name that was first given when the object was defined;
- the *short name* : an abbreviated name for the object; can be used to build default names in some transformations;
- the *origin* : the natural name of the object from which the current object is derived;
- the *technical name* : a synonym that satisfies DBMS or programming language naming syntax and that will be used to generate executable descriptions (DDL schema); by default, the technical name is the natural name.

The ***structural facet*** defines the relationships between an object and its neighbor objects.
- a *schema* : comprises entity types and relationship types; is linked to other schemas;
- *entity type* : is in a schema, plays roles in relationship types, has attributes, has groups, is in spaces;
- a *relationship type* : is in a schema, has roles, has attributes, has groups;
- a *role* : is defined in a relationship type, is played by entity types, is in groups;
- an *attribute* : is attached to an entity or relationship type, is in a group;
- a *group* : comprises attributes and/or roles; is attached to an entity/relationship type; is linked with another group (included into, equals, copy of);
- a *space* : contains entity types.

The ***statistical facet*** defines static and dynamic quantification of the data.

Static statistics :
- entity type : average population size (N in schema 11);
- relationship type : average population size (derived: = μ of a role x N of its entity type));

---

[1] The notion of synonyms for a concept is considered another way. Indeed, when a problem is reasonably formalized, a consistent set of names is associated with a collection of objects in a given context, such as in a service. Therefore, it is better to represent the context in which a name is given to an object, rather than all the names under which this object is known. We have chosen to represent a context by a view, which in turn is represented by a schema. So, an object can have an arbitrary number of names, but each of them is defined in one of the contexts in which it is known, i.e. in a schema that includes this object.

- attribute : average length (μL), average number of values per entity/relationship (μN);
- role : average number of times an entity plays that role (μ);
- group : average number of distinct values (N), average number of entities with which a group value is associated (μ);
- space : average number of entities of each type it contains.

Dynamic statistics (average number of operations per time unit) :

- entity type : creation (add), deletion (delete), update (update) of entities; average number of sequential accesses (x average number of entities for each access);
- relationship type : for each access path, average number of accesses (x average number of entities for each access);
- group (with role *access key*): average number of accesses (x average number of entities for each access);

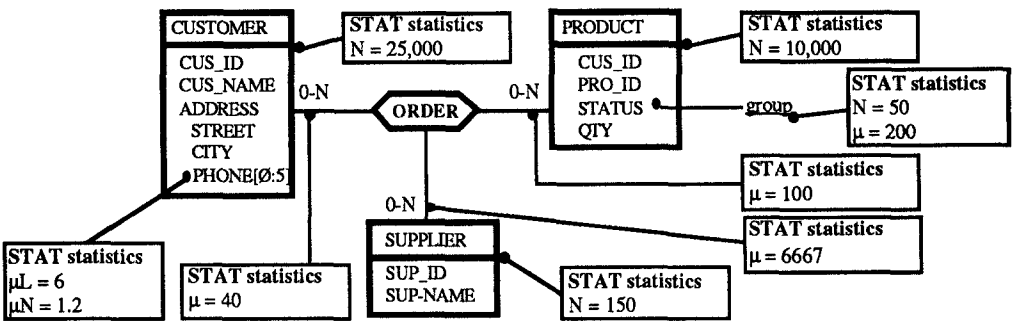


Fig. 11. Example of static statistics

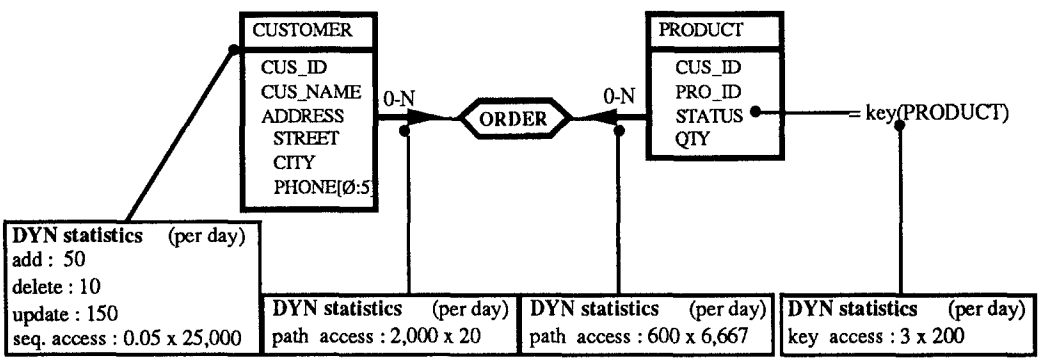


Fig. 12. Example of dynamic statistics

Through the ***informal specification facet*** textual information is added to objects.

- *Description* : gives an object its *interpretation* according to the real world concepts it denotes. This description gives the semantics of the object.
- *Technical notes* : annotation that gives justification concerning design choices or gives the future programmers and administrators useful informations and advices to maintain integrity of the DB. It is managed by both the designer and TRAMIS. Generally concerns lower-level schemas.

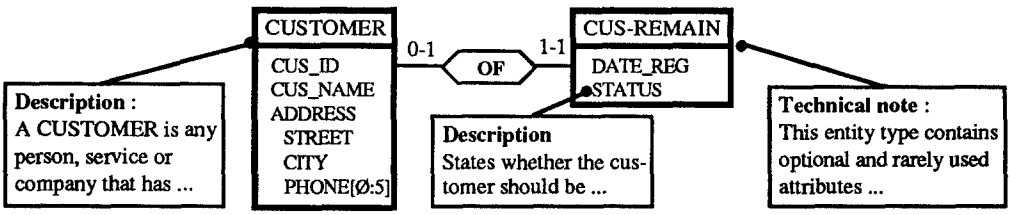


Fig. 13. Example of informal specifications of objects

The generic specification model of TRAMIS can be **specialized** into a great variety of **submodels**, for instance according to the design levels or design product classes of a standard or user-defined multilevel design method, or according to the target DBMS. The following schemas are typical examples of a consistent set of specification models describing standard design products. Each schema must include all the specifications of the previous one (i.e. the information must be either the same, or transformed through specification-preserving operations). Such products can be used by strategies such as that of Fig. 22.

- a *conceptual schema* comprises a schema, entity types, relationship types, attributes, identifier groups, static statistics.
- In an *interpreted conceptual schema* all objects have been given a semantic textual description;
- a *quantified conceptual schema* is a conceptual schema with static statistics;
- a *logical access schema* is a binary schema in which the logical access constructs (access key groups and access paths) needed by the applications have been defined;
- a *quantified access schema* is a logical access schema with dynamic statistics;
- an *optimized access schema* is a logical access schema in which performance-oriented transformations have been carried out (and justified in a technical note);
- a *DBMS-compliant schema* is a logical access schema that has been transformed in order to make it compliant with the specific model of a DBMS.

Specializing the generic model into a specific submodel can be done by the set of rules the schema must satisfy. These rules formalize a specific category of requirements as defined in 3.3.

Example : an ORACLE-compliant schema is a TRAMIS schema in which,

- there are no relationship types;
- there is from one to 254 attributes per entity type;
- the attributes are single-valued and atomic;
- a group must be an access key (i.e. an index);
- an entity type cannot be in more than one space.
- a name is made up of 1 to 30 characters, the 1st being a letter, and the other ones being letters, figures, '_', '$' or '#';
- a name cannot belong to the ORACLE reserved-word list ('create', 'index', etc);
- the total length of the attributes that make a group cannot exceed 240 char.

In these constraints an entity type is to be interpreted as a *table*, an attribute as a *column*, a space as an ORACLE *space*.

5. THE TRANSFORMATIONAL APPROACH IN TRAMIS

Specifications transformation is a basic paradigm of recent CASE tools such as KBSA [8]. In this perspective, TRAMIS generalizes the concept of schema transformation by providing a large set of transformations that can be used at any level of the design process. They are in no way

goal-oriented (except for model-driven ones), and can therefore be used in any design process and any strategy.

The notion of *reversibility* is an important characteristic of a transformation. If a transformation is reversible, then the source and the target schemas have the same descriptive power, and describe the same universe of discourse, although possibly with a different presentation (syntax). Reversible transformations are also called *semantics-preserving*. The transformations provided by TRAMIS preserve not only the semantics of the source schema (i.e. the conceptual structures), but the other aspects of the specifications as well : technical structures, names, statistical and informal information. For instance, the access structures (access keys and access paths) in S1 are transformed into different, but functionally equivalent access structures in S2; statistics in S1 are recomputed according to the new structures in S2; informal specifications of objects that disappear from S1 are transferred, possibly with some adaptations, to the new objects in S2. TRAMIS transformations are not only semantics-preserving but also *specification-preserving*[1].

5.1 The transformation toolset

TRAMIS proposes a three-level transformation toolset[2] that can be used freely, according to the skill of the user, the desirable efficiency of the executable schema and the time allowed for producing that schema :

- **elementary transformations** : one transformation is applied to one object; with these tools, the user keeps full control on the schema transformation since similar situations can be solved by different transformations; e.g. a multivalued attribute can be transformed in four ways;
  Formally : given construct C in schema S and transformation T, both selected by the user, C is replaced by T(C) if $P_T(C)$.
- **global transformations** : one transformation is applied to all the relevant objects of a schema.
  Formally : given transformation T and a class of constructs CC, both selected by the user, for each C in CC such that $P_T(C)$, replace C by T(C). Examples : replace all one-to-many relationship types by foreign keys + referential constraints; replace all multivalued attributes by entity types + many-to-one relationship types.
- **model-driven transformations** : all the constructs of a schema that do not comply with the rules a given model are transformed; these transformations require virtually no control from the user; the resulting schema is correct, complies with, say, the relational or CODASYL model, but has few refinements as far as efficiency is concerned[3].

---

1 To be quite precise, all the transformations of TRAMIS are specification-preserving, but some of them are not semantics preserving. For instance, transformation Tr-A3 is specification augmenting, since ordering and identity properties, lacking in the source schema, are given to the instantiated attributes. On the contrary, transformation Tr-A4 is clearly semantics-degrading.

2 A schema transformation should not be confused with a design process, although the latter is said to *transform* a product into another one. Indeed, a design process that preserves the specifications need not be reversible (an unfortunate fact as far as reverse engineering is concerned). For example, adding an entity type to a schema transforms it into a richer schema that preserves the previous specifications, while the reverse process (deleting the entity type) doesn't preserve the specifications that are in the second schema.

3 This is the only schema production function that is provided by many current database CASE tools.

This process is based on a complex strategy driven by two sets of requirements, namely compliance to the selected data model, and time efficiency[1]. According to the analysis presented in section 3, problem P1 is solved by rule priority (the rule with the highest priority among those that C violates triggers first), problem P2 is solved by defining a default transformation for each rule (a more sophisticated strategy is planned), problem P3 is solved by discarding the construct and by writing its description as an annotation in the informal specification, problem P4 uses a monotonicity property of some transformations that can be informally interpreted as follows : given a construct C that violates rule r, transformation T such that $\neg P_T(C)$ can be applied provided T(C) "*is a less difficult problem according to r*" or T(C) induces "*a less severe violation of r*" or T(C) is "*closer to a satisfying construct than C*"[2]. Problem P5 is solved by adhoc ordering of rules to be checked.

Here follows a list of some **elementary transformations** proposed by TRAMIS. For conciseness, only conceptual structure transformations will be considered and given a short explanation; the transformation of the other facets will be evoked later. A formal description and the proof of the reversibility of most transformations that introduce/remove an entity type (TR-E1, TR-E2, TR-R1, TR-A1, TR-A2) can be found in [7].

5.2 Transformation of entity types

- Tr-E1 : Decomposes an entity type E into two entity types E1 and E2, and distributes the attributes, the roles and the groups of E among E1 and E2.
- Tr-E2 : Integrates an entity type E2 into an entity type E1. Reverse of Tr-E1.

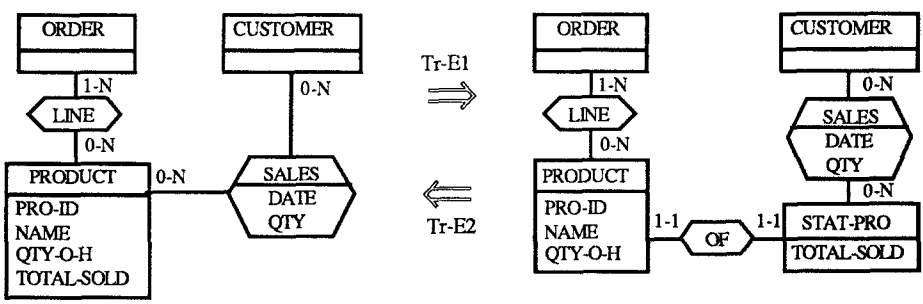


Fig. 14. Decomposition of PRODUCT into two entity types by extracting one attribute and a role.

5.3 Transformation of relationship types

- Tr-R1 : Replaces a relationship type R with an entity type and as many one-to-many relationship types as R had roles.
- Tr-R2 : Replaces a one-to-many relationship type R between E1 and E2 by attributes of E2 that are a copy of the identifying attributes of E1 + referential integrity constraint.

---

1 In the current version, the efficiency requirements are satisfied by very simple rules. More advanced optimisation rules will be used in the future. The rationale is that, at least at present time, high efficiency can mainly be ensured by human expertise.

2 This concept depends on the set of transformation that is available. For instance, according to the transformations set that has been chosen in TRAMIS, a one-to-many relationship type is not as far from relational structures as a many-to-many relationship type is. Another approach to this problem could have been to define new transformations as the composition of simpler ones [12].

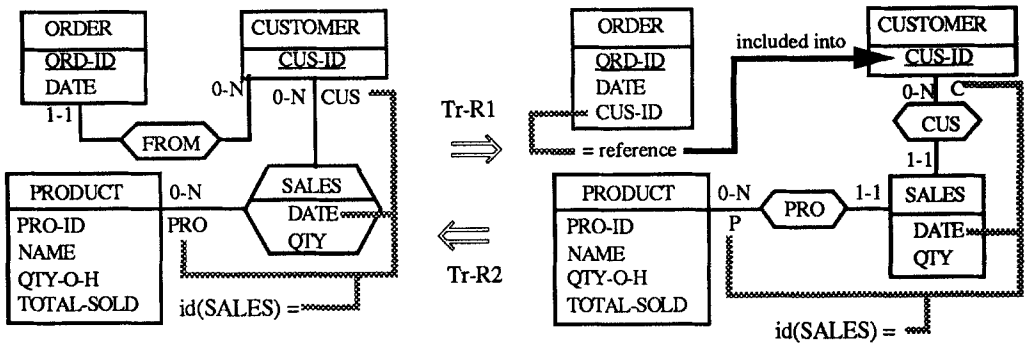


Fig. 15. Transformation of relationship type FROM into reference attributes (foreign key), and transformation of relationship type SALES into an entity type.

5.4 Transformation of attributes

- Tr-A1 : Transforms an attribute A of entity type E into an entity type EA and a many-to-many relationship type RA between E and EA. Each EA entity represents a distinct value of A.
- Tr-A2 : Transforms an attribute A of entity type E into an entity type EA and a one-to-many relationship type RA between E and EA. Each EA entity represents an occurrence of an A value attached to an E entity.
- Tr-A3 : Transforms a multivalued attribute into a list of single-valued attributes.
- Tr-A4 : Transforms a multivalued attribute into one single-valued attribute.
- Tr-A5 : Aggregates a group of attributes into a compound attribute.
- Tr-A6 : Decomposes a compound attribute. Reverse of Tr-A5.
- Tr-A7 : Transforms a compound attribute into an atomic attribute.

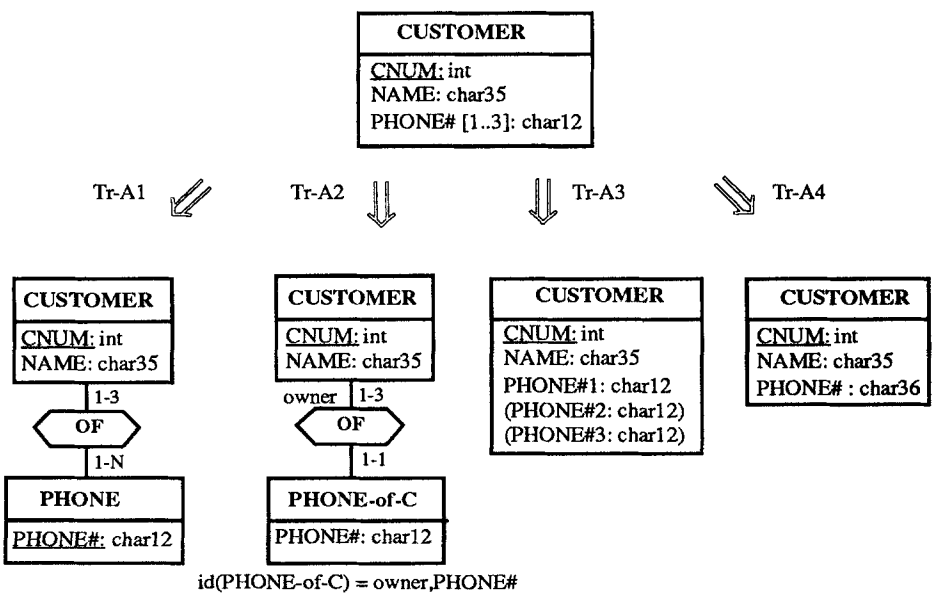


Fig. 16. Four possible transformations to get rid of a multivalued attribute.

5.5 Transformation of groups

- Tr-G1 : Separates two overlapping groups by duplicating common attributes (e.g. for COBOL data structures).
- Tr-G2 : Replaces a role component by the identifier of the entity type playing that role (e.g. for CODASYL schemas as in the schema below).
- Tr-G3 : Adds a singular relationship type role to a group (e.g. for CODASYL schemas).

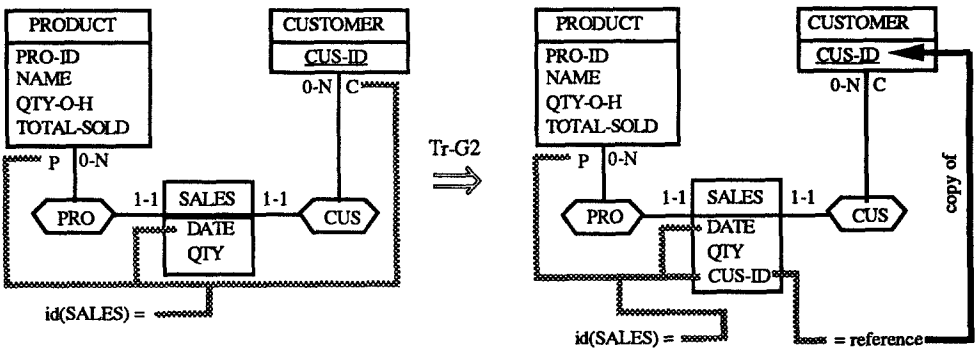


Fig. 17. Replacing role C in the identier of SALES by the identifier of CUSTOMER (the CODASYL data model doesn't accept more than one role in a unique constraint).

5.6 Transformation of names

- Tr-N1 : Substring substitution.
- Tr-N2 : Name prefixing.

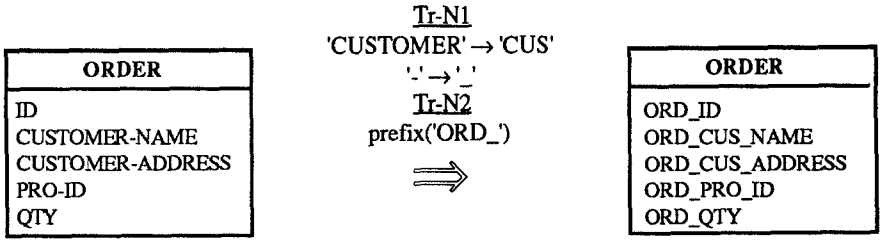


Fig. 18. Translation and prefixing operations.

5.7 Transformation of the non-conceptual facets

As already specified, the transformations concern all facets of the objects. For instance, transformation Tr-A1 processes the access structures ...

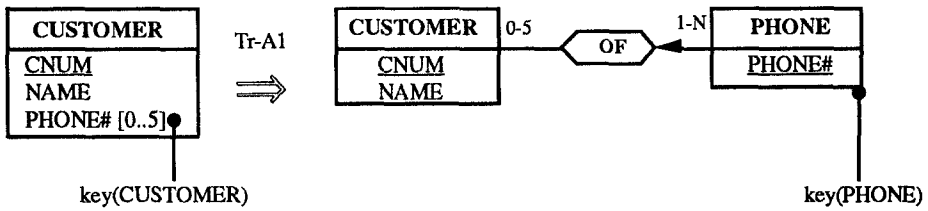


Fig. 19. Translating an access key into an access path (e.g. in a CODASYL schema that doesn't accept multivalued access keys).

as well as the statistical facets (only the static statistics are illustrated below), ...

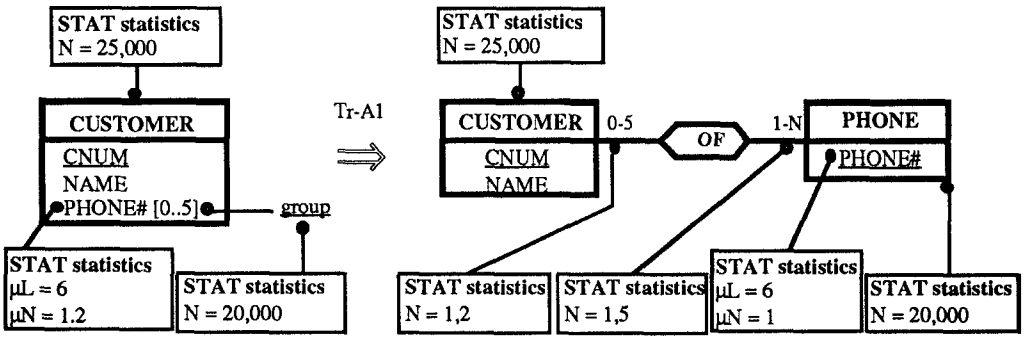


Fig. 20. Conversion of static statistics in a structural transformation.

6. ARCHITECTURE AND FUNCTIONS OF TRAMIS

TRAMIS provides a set of tools that can be used freely. The tools are independent from each other and communicate through the common specification database only. The toolset architecture of TRAMIS results primarily from methodological decisions about the kind of design processes the tool is aimed at.

- A design process, particularly in software engineering, generally follows an incremental, trial-&-error global strategy, in which prototyping and backtracking are common practices and in which more than one activity can be carried out in parallel;
- Both skilled and novice (or hurried) designers must find their way easily. In other terms, some users need fine-tuning tools on which they have full control, while other users need simple and automated tools for quick production of executable descriptions.
- Database schema specifications can be evaluated and processed even if they are incomplete and non consistent. Therefore, the tools must be *robust*.
- If the conceptual design step is recognized as a major activity, the technical design step must also receive sufficient attention. In most case, an automatic translation of the conceptual schema into a DBMS-DDL produces no more than an incomplete, prototype, database schema that must be further processed by hand. A database CASE tool must provide its users with a large set of tools for the evaluation and production of efficient executable solutions.
- The behaviour of the tool, and more specifically the specification model and the user interface, must be non-obtrusive for the different classes of users. The user must not be aware of concepts and tools he wants to ignore.

The toolkit organization of TRAMIS, together with the three-level architecture of the transformation toolset, provides an adequate basis for such a flexible database design approach.

TRAMIS is organized as an extensible collection of toolsets working on a common specification database (or repository). The specification database contains the current state of the project database schemas under development. The toolsets are accessed through a common dialog manager providing a WIMP-based user-interface.

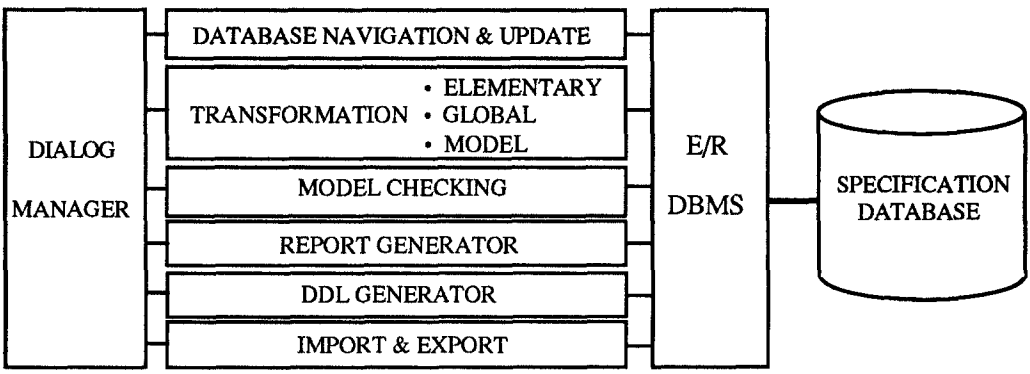


Fig. 21. Gross architecture of the TRAMIS CASE tool.

NAVIGATION and UPDATE : allows the browsing of the current state of the specification database, object selection by name or in a list of objects and the navigation from object to associated objects. Allows the consultation and updating of the different facets of the current object, together with the creation and deletion of objects.

SCHEMA TRANSFORMATION : allows the application of one elementary transformation on the current object, the application of one transformation on all the objects of the schema, and the application of all the necessary transformations on the objects of the schema in order to transform it into a new schema that complies with a selected model. All the facets of the transformed objects are recomputed in order to preserve their information contents (a typical elementary transformation requires from 10 to more than 100 elementary updates, a performance that cannot be done by hand).

MODEL CHECKING : at any moment, the user can validate its specifications against a selected submodel. TRAMIS compiles the rules defining the submodel and analyzes the current specifications according to these rules. TRAMIS generates diagnostics that will be examined by the user. A submodel is defined in an ASCII text consisting of a list of statements expressed in a specific rule language. The user can build any number of submodels. TRAMIS includes four predefined, general-purpose submodels (standard E/R, relational, CODASYL, COBOL files).

REPORT GENERATION : production of a detailed report , for all objects or for one object, a statistical report (with volumes and operations), a dictionary report (concise structured list of object names with *natural/technical* names translation or conversely).

DDL GENERATION : translation of a TRAMIS schema into executable descriptions according to a selected DBMS. The generation is twofold : production of a **DDL text** and generation of an **additional document** that reports all the integrity constraints that have not been translated in the DDL text together with validation procedures for these constraints. This report, aimed at the programmer and the DBA, includes the technical notes as well.

IMPORT / EXPORT : TRAMIS has been given an external specification language (Information System Specification Language, or ISL) in which any specification can be expressed. This language allows communication between tools, selective backup or even data input. Through the *export function*, a selected part (what objects and what facets) of the specification database contents can be generated in ISL. Through the import function, the contents of an ISL text can be integrated to the specification database. This function allows an incremental integration of different schemata.

7. METHODOLOGICAL SPECIALIZATION OF TRAMIS

The toolkit approach of TRAMIS gives the user a high degree of freedom in its design behaviour. This freedom is not always welcome, particularly where a strict methodological standard has to be followed. TRAMIS can help enforce such standards through the concept of *submodel.* Any design method can be formalized as a set of processes, each of which consists in transforming and enriching a specific state of the specifications into another one. A specific state of specifications, e.g. a database schema, can be characterized by the information it includes and by the constraints this information must satisfy. The schemas that are mentionned and described at the end of section 4 constitute examples of significant states in many design methods. Each of these kinds of schema can be defined by a *specialization* of the general specification model of TRAMIS, i.e. by a *submodel.* This submodel is defined by a set of restricting rules stating whether an arbitrary schema satisfies the submodel. Some general-purpose submodels are already built in TRAMIS : standard E/R, relational, CODASYL and standard file structures. More specific models are also proposed for actual DBMS (ORACLE, RDB, IDS-2, etc). However, TRAMIS provides the user with a **rule language** with which he can define his own submodels. In addition, the **model checking** function of TRAMIS can evaluate the current state of the specifications under development against a chosen set of rules, i.e. according to a given submodel, and reports all the violations[1].

It is therefore easy to implement a specific design method as follows :

- identify the main design processes;
- identify and define precisely the products, i.e. the specification states that are inputs and outputs for these processes;
- define these states by compliance with a submodel of TRAMIS and implement this submodel as a set of rules.
- each design process can be considered complete when its resulting schema satisfies the submodel concerned .

The following schema illustrates two typical design methods (processes and products) that can be implemented into TRAMIS through the definition of adhoc submodels. The left-hand approach defines a simple, three-phase process, particularly suited for prototyping. The right-hand approach proposes a detailed methodology for optimized schema production.

---

[1] In the current state of TRAMIS, the role of a submodel is basically passive. Indeed, the whole generic model is available to the users. The latter can only ask the tool to evaluate the current specifications against a selected submodel. An active role would be as follows : at any time, a submodel is current; all the operations on the specifications are monitored by the rules of the model. For instance, a submodel that defines the Bachman's model (Data Structure Diagrams) would prevent users from defining relationship types with a degree greater than two. Such a behaviour will not go without problems, since changing the current model (for example shifting from E-R to Bachman, or to SQL) would make the current state of the specifications partly invalid.

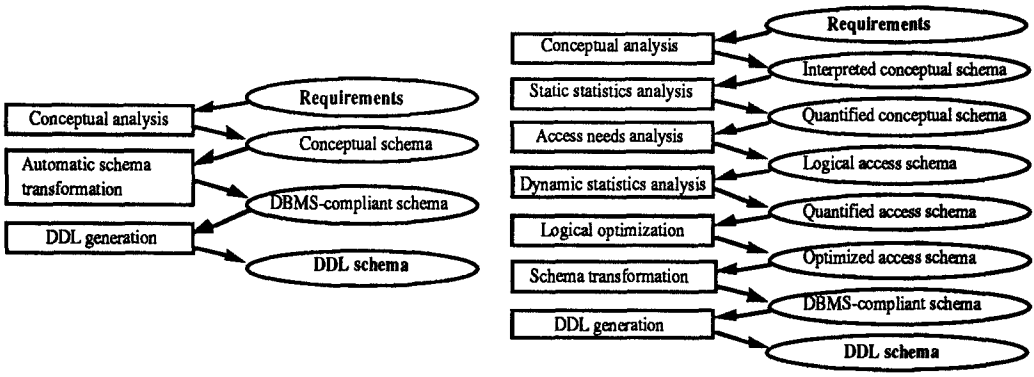


Fig. 22. Two typical strategies for database design.

8. CONCLUSIONS

TRAMIS is a database CASE tool that has been designed with two important hypotheses in mind. The first hypothesis is that designing a complex system can be perceived as carrying out design processes that produce design products from other design products, in such a way that the former satisfy requirements the latter don't satisfy. A tool must accept a great variety of design behaviour, and therefore, must support several standard or non-standard design strategies. The second hypothesis is that schema transformation is a major tool for database design, and that is can be the basis of many design processes. For instance, database schema design can be seen as a step-by-step transformation of some initial version of the conceptual schema.

These principles have lead to four important architectural decisions : the unique specification model whatever the design level, a toolset interface that includes transformational functions, and a facility for fefining design product classes through submodels. Indeed, the flexibility that must be provided to ensure maximum freedom in design processes and strategies makes it impossible to define a fixed set of models and standard design methods.

However, the two basic hypothesis have not been fully applied yet. Indeed, many design processes and strategies must be carried out by the designer himself, since the tool automates mainly low-level processes and strategies. The most complex process offered by TRAMIS is the automatic production of DBMS-compliant schemas. Beyond this function, defining and conducting sophisticated processes and strategies is up to the user, mainly helped by the submodel definition and the model checking function.

On the other hand, several additional schema transformations should be included in order to allow more sophisticated schema restructuring techniques. In particular, operators for manipulating generalization/specialization constructs, additional redundancy and denormalization transformations, and new reverse engineering transformations should be added.

TRAMIS has been developed by the University of Namur both as an educational tool and, later, as a component of an industrial design environment. It is written in C for MS-Windows workstations. A commercial version of TRAMIS is available under the name TRAMIS/Master[1]. It is complemented with TRAMIS/View, a graphical editor and integrator for MERISE-like database descriptions, and with TRAMIS/Flow, allowing the definition of data-flow diagrams.

---

1 Distributed by CONCIS, 37bis, rue du Prébuard, 95100 Argenteuil, France.

10. REFERENCES

[1] Batini, C., Lenzerini, M., Moscarini, M., *View integration, in Methodology and tools for data base design*, Ceri, S., (Ed.)North-Holland, 1983

[2] Dubois, E., Van Lamsweerde, A., *Making Specification Processes*, in Proc. of the 4th Intern. Workshop on Software Specification and Design, Monterrey (CA), April, 3-4, 1987, pp. 169-177

[3] Giraudin, J-P., Delobel, C., Dardailler, P., *Eléments de construction d'un système expert pour la modélisation progressive d'une base de données*, in Proc. of Journées Bases de Données Avancées, Mars, 1985

[4] Hainaut, J-L., *Theoretical and practical tools for data base design*, in Proc. of Very Large Databases, pp. 216-224, September, 1981

[5] Hainaut, J.-L., *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on *Information System Concepts: an in-depth analysis*, North-Holland, 1989.

[6] Hainaut, J-L, *Database Reverse Engineering, Models, Techniques and Strategies*, in Proc. of the 10th Conf. on Entity-Relationship Approach, San Mateo, 1991

[7] Hainaut, J-L., *Entity-generating Schema Transformation for Entity-Relationship Models*, in Proc. of the 10th Conf. on Entity-Relationship Approach, San Mateo, 1991

[8] Johnson,W., L., Cohen, D., Feather, M., Kogan, D., Myers, J., Yue, K., Balzer, R., *The Knowledge-Based Spesification Assistant*, Final Report, UCS/Information Sciences Institute, Marina del Tey, 19 Sept. 1988

[9] Kobayashi, I., *Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence*, in Information Systems, Vol. 11, No 1, pp. 41-59, January, 1986

[10] Krieg-Brückner, B., *Algebraic Specification and Functionals for Transformational Program and Meta Program Development*, in Proc. of the TAPSOFT Conf. LNCS 352, Springer-Verlag, 1989

[11] Kozaczynsky, Lilien, *An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation*, in Proc. of Entity-Relationship Approach, 1987

[12] Partsch, H., Steinbrüggen, R., *Program Transformation Systems*, Computing Surveys, Vol. 15, No. 3, 1983

[13] Reiner, D., Brown, G., Friedell, M., Lehman, J., McKee, R., Rheingans, P., Rosenthal, A., *A Database Designer's Worbench*, in Proc. of Entity-Relationship Approach, 1986

[14] Rosenthal, A., Reiner, D., *Theoretically sound transformations for practical database design*, in Proc. of Entity-Relationship Approach, 1988

[15] Teorey, T. J., *Database design*, Prentice-Hall, 1989

[16] Tardieu, H., Rochfeld, Coletti, *La méthode Merise*, Les Editions d'Organisation, 1983