Incremental Consistency of the Predicator Set Model

Iván Alejandro Ballón Carranza

University of Nijmegen
The Netherlands

... A quienes amo y
son todo en mi vida:
Chelita, Jorge, Rosita, Edwin,
Patty, César y María Angélica,
Angélica Reyna,
Lillian y nuestros hijos

Contents

List of Figures

Preface

This thesis is the outcome of the research that was part of my graduation in computer science at the Catholic University of Nijmegen in the Netherlands.
In assignment of the Department of Information Systems of the university, it was my task to write this report on the incremental consistency of the predicator set model.

Here, I would like to thank my attendant Dr. Ir. Theo van der Weide who has spent a rather large part of his valuable time on my research and who also helped me to achieve these results.
I also want to thank all those people here in the Netherlands ('Wat een kikkerlandje!') and especially those abroad (! A toda mi gente en Perú !) who helped and supported me to complete this research and my whole study.

Iván Alejandro Ballón Carranza
Nijmegen, February 1994

Abstract

The Predicator set model (PSM) is a powerful data modelling technique that makes possible the representation of complex information structures. This model is an extension of NIAM and is a general platform for object-role models. Conventional data modelling techniques allow only flat structures and are not capable of representing complex structures in a natural way like PSM does.

An information structure is represented on a scheme which, after checking the corresponding constraints, is supposed to be consistent. PSM facilitates the implementation of case-tools which also are intended to detect inconsistencies. By updating an information structure we need to maintain an consistent scheme; this is rather an NP-complete problem [7].

By applying some heuristics to the predicator set model it is possible to determine the consistency of some scheme.

The strategy to be followed is to check just a limited control of the scheme that results after extending or shrinking de original scheme.

Introduction

The predicator set model (PSM) is an extension of NIAM , an existing data modelling technique, and is widely used in the conceptualization of information structures.

The efficiency requirements in programming languages and in databases grow stronger. In programming languages and also in context free grammars it is possible to add or remove some parts of an existing structure without the need of checking the whole work again. Some programming languages recompile just the part that has been added or the remaining program if some part has been deleted and not the whole program.

In PSM, the constraints are important for the identification purposes. The validity of these constraints in the whole context, after a possible update, has to be checked and, by following some heuristics, it might be possible to determine whether or not the resulting scheme, stays consistent.

In the first part of this thesis, the comprehension *incrementation* will be explained. Also the relation of incrementation with databases is shown.
In the first chapter a description of the incremental subject in programming languages will be given. The incremental approach is applied to the need of updating databases.
The second chapter gives a description of the issues that appears by including new requirements in a database.
In the third chapter, a sequence order of the update process is described, in order to take into account the steps that will be followed in the update of a PSM scheme. This will be considered in different stages in the next chapters.
Chapter four describes shortly, the predicator set model. This chapter also shows how constraint checking takes place on a PSM scheme.

In the second part of this thesis, the incremental consistency of a PSM scheme is analysed, followed by proposed heuristics.
Chapter five focus more on complex interactions between the relationschips on a scheme during life and, the heuristics to be followed for updating a scheme incrementally within a certain environment.

Chapter six offers a mean of applying these heuristics by steps on different stages during the modelling process.
Finally, chapter seven focus on the incrementation issues on a PSM scheme during transformation, where also guidelines are given for updating a relational database scheme.

Part I

Incremental topics and updates

Chapter 1

Incrementation on databases

Incremental issues have become a challenge in the research of programming languages and also in the development of databases.
Incremental issues mean series of regular consecutive additions, or a positive or negative change in the value of one or more of a set of variables. In databases this can be seen as an extending or shrinking of one information structure in parts. In this thesis, schemes of information structures will be analysed. Increment, in a sense of checking the relations of objects modules within a program, deals with the analysis how they influence each other in order to gain efficiency and to save time.

For the incrementation matter a lot of research has been done, especially in programming languages and context free grammars.
Each programming language has a different approach to this issue or a different way to describe the dependence relationship between objects.

1.1 Incremental Background

In programming languages, the incremental problem becomes clear as the maintenance of a program is measured in relation to the time that it takes to update some part of it, and as time seems to be wasted in recompiling programs that have been modified because of a default or just because a new part has been added; The action of re-compiling the whole program becomes worthless.

Researchers make strenuous efforts to let the system discover by itself that these relationships **do** exists and, at this point, two thinking streams are depicted.
The first one is to let the user indicate all the dependencies to the system; this is well known to be the cheapest way, but the inconsistency possibility must be taken into account because the user can omit some dependency or just consider a dependancy to be irrelevant.
In programming languages like *'Turbo Pascal'* for example, the 'users relation' has

to be given in advance. The relationships between what has to be imported and what has to be exported have to be given for the sake of the user. Another example is the programming language *'C'* that uses of 'make files' to determine the correspondent relations between objects.
The second one is to let the system determine the dependency between objects. What would be the impact on the system when changing one object?. There are also two possibilities; either the system works with explicit specifications - in that way it is always known what happens exactly at each variation, or let the system work slower - in this case a result might be faster.
This second approach is quite expensive because of the complex calculations that have to be done, and also because of the research that strives to this feature. This has lead to the *Object Oriented* approach where all changes in objects must be processed locally.

Computer science used to emphasize on processing data, while later the emphasis switched to the development of databases. A database in the early days used to be just the ability to provide data to a program, the input, and to get the result, the output. Programming languages have been helpful in the development of databases. Nowadays a lot of research has been done in order to improve computer languages which support relational databases. Also, a lot of research has been done in order to improve our knowledge of context free grammars.

Context free grammars can also be seen as programs in which two parsing methods can be distinguished: *the bottom-up* and *the top-down* approaches. Both of them strive toward the analysis of the relationships between objects. Research on generating environments is motivated by the exploration of the incremental behaviour of such environments [8]. Programming environments are mutually unrelated collections of languages; independent tools available on an arbitrary computer like editors, debuggers and compilers [15].

1.2 Incremental performance

The programming language *'Modula-2'* gives an approximation to the solution of the incremental problem. The concept of introducing modules is the solution of the program size-related problem. Modules allow program components to be kept in libraries. These modules are already compiled, so it can be avoided to duplicate work when writing any programg that uses the same code.
These programs can import these modules from a library and re-use them without the need of recompiling [16].
The solutions given by the programming languages offer a survey and categorization of a number of proposals, but this does not mean that one of them is the best solution. In databases the problem seems to have surrounding matters that can lead to further abstraction.

1.3 Increment and PSM

Imagine one wishes to build, use and maintain an information structure, translated on a relational database. Of course, by mantaining the structure one mean the updating of this structure when more requirements are imposed on the *universe of discourse* (UoD) at conceptul level. This matter leads to the question of adding or removing information, or contradicting the existing constraints in order to make an update succeed and perhaps contradicting the *conceptualization principle*.
The Predicator Set Model (PSM) is an extension of the Predicator Model which on its turn is a formalization of the data modelling technique NIAM [11].
The PSM is a data modelling technique which do not violate the conceptualization principle that states a conceptual scheme deals exclusively with the UoD and aims at the representation of data at high level.

A very important issue for further incremental analysis is the relation type. A relation type in the Predicator Set Model is considered to be a set of predicators, where a predicator is the combination between an object type and a role; the populations are the states of the UoD. Unfortunately not all populations correspond to real states of the UoD and therefore some have to be ignored, this happens when imposing the constraints on an information structure that limits the validity of some states.
These constraints are called static. Formal definitions and further description about PSM can be found in [12].

The incremental approach is then meant to reduce the number of transactions that results from updating object types or populations on a PSM scheme. The completely new scheme does not have to be evaluated, only the added parts or the remaining scheme if some part has been removed.
To check whether or not the update succeeded by no contradicting the existing constraints of the original scheme only the static constraints will be analysed. These constraints are categorized in hierarchy and must obey their own independency rules. The hierarchy of the static constraints is as follows:

- Uniqueness constraint
- Total role constraint
- Subset constraint
- Exclusion constraint
- Occurrence constraint

The update of a structured PSM scheme must take place in steps. This seems to be intuitively desired since there are no specific heuristics that prevent possible disturbing or variation of the information structure. Also for this purpose, the use of the history of the information structure is quite important. For each update on a PSM scheme the result must be checked to avoid keeping worthless information. In order to achieve this result, the interface has not been changed. As example see the figure 1.1 where the insertion of a tuple in a database has to be first validated. The tuple must satisfy the constraints, otherwise the tuple will be rejected.

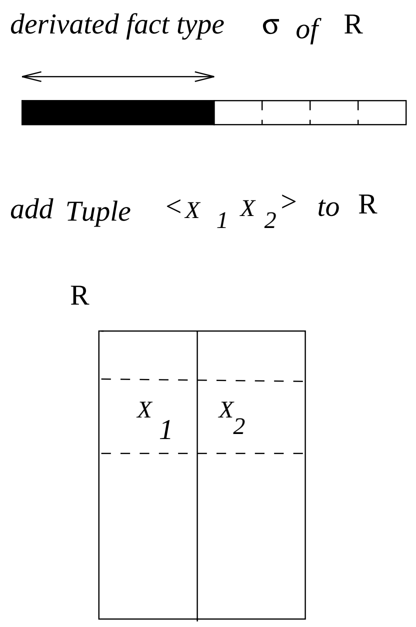


Figure 1.1: Example of validation of a tuple update

Chapter 2

Database update

In this chapter, one of the principal functions of a database management will be considered: the update.
An update takes place at different levels, but in this thesis more attention will be focused on PSM patterns.
The problem of updating databases in steps has been considered to be difficult for a long time. Given a view (where the operands are base relations, whose tuples are actually stored), the question arises - considering both theorical and practical issues - what the result of an update would be on this view.

2.1 Where to adapt after an update?

A database is a dynamic structure; performing variations on it must be possible. However, functionality of a database implies the whole structure should be checked again after an update in order to guarantee consistency at conceptual as well at relational level.
Conventional relational database design methods divide the database problem into logical and physical design phases. In the logical design phase, normalization is used to generate a scheme that satisfies dependency constraints, and therefore, certain update anomalies are avoided.

Normalization tends to break existing relations into smaller pieces, and does not take into account the user query and the update patterns. Therefore the resulting scheme may be very inefficient for processing frequently executed queries. In the physical design phase, the physical storage characteristics of the data are carefully chosen in order to optimize collectively the performance of all queries [5].
In this thesis some heuristics are proposed to reduce the area in the database which has to be adapted. Thus more effectiveness can be obtained by a local adaption. The determination of the adaption area, which will be called here *environment* will be done with the help of heuristics proposed in section 5.2.

2.2 About knowledge and data

Database management systems usually offer a means of expressing simple pieces of information that are best thought of as knowledge rather than data. There is a subtle difference between data and knowledge. *Data* is a mutable information that may be changed as the result of an incoming update.
The notion *knowledge*, which has been introduced in the beginning of the eighties, is a set of statements that describe the thruths of the actual world plus a set of constraints that describe statements that must be true in all possible worlds and statements that ought to be true (in all possible worlds).

In this sense, the closed world assumption will be taken into account. As a matter of fact, in the traditional database world the database schemes, integrity constraints, view definitions and also the closed world assumption itself are considered to be knowledge, which means that they cannot be changed by ordinary updates. The tuples residing in relations are the data portion of a traditional relational database.

2.3 Update approaches on databases

Incomplete information occurs when, there is insufficient knowledge about the state of the world, due to the fact that there might be more than one database (or internal representation) candidate to represent the current state of the world.
In the database world, one can imagine the user keeping a set of relational databases. He knows that one of these databases corresponds to the actual state of the world, but needs more information in order to know which database is the correct one.
There are undoubtedly different and numerous approaches to perform updates but at last, heuristic guidelines will be the leading ones. This becomes clear as it gives a powerful estimating tool for the different possible states of the world. Just one critical question will always remain:

How is one to incorporate these heuristics into an update algorithm. Anyhow, one of the most common ways to update a database has been to create special query languages like the fourth generation query language SQL.
However, traditional relational update languages are not sufficiently powerful when dealing with incomplete information. The traditional languages also lack semantics that are sufficiently formal for a rigorous examination of the properties of these languages.

The world changes - that is the turning by which databases may change too, and, therefore have to be updated. In this sense, conceptual data modelling techniques aim at the representation of data at high level of abstraction. Conventional conceptual data modelling techniques like NIAM, ER, IFO, or NF^2 have to violate the conceptualization principle when dealing with objects with a complex structure. In order to represent these objects, conceptually irrelevant choices have to be made. This implies overspecification which can be translated into adaptation of the UoD. The PSM disposes of sufficiently powerful construction mechanisms that avoid facing these kind of problems.

Chapter 3

Update process

In this section, the process of how an update in a database takes place is described. Performing updates in a database means not only adding subjects but also deleting useless information or modifying structures at conceptual and/or relational level.

3.1 The process

Every time a user requests an update, these requests are made against the basic relations; the user only sees the portions of the basic relations which are stored in the files where no constraints are violated. Figure 3.1 shows the update process.
The request processor accepts the request, parses the request, and changes it into an internal representation. If the system contains security modems, then, the security constraints will be checked in the whole process. Depending on this phase, an update operation can be accepted or rejected.

The update generator checks the request according to the constraints by following heuristics or policies defined in advance. Finally, this request goes into the file processor which will perform the necessary changes in the database, but will send a message to the user in order to verify the update request. In this way not only the environment of the system will be checked, but also the integrity constraints, inference rules and the real world information.

3.2 The update process on a PSM scheme

Due to this research, this process will be applied carefuly into a PSM scheme where three different stages have to be taken into account.
Incremental update during life where the most restructuring of populations will take place. When a tuple population is added, deleted or modified, the scheme on which the tuple has to fit may contain the necessary constraints which not contradict the tuple and, the tuple update may have to go in concordance with the scheme constraints in

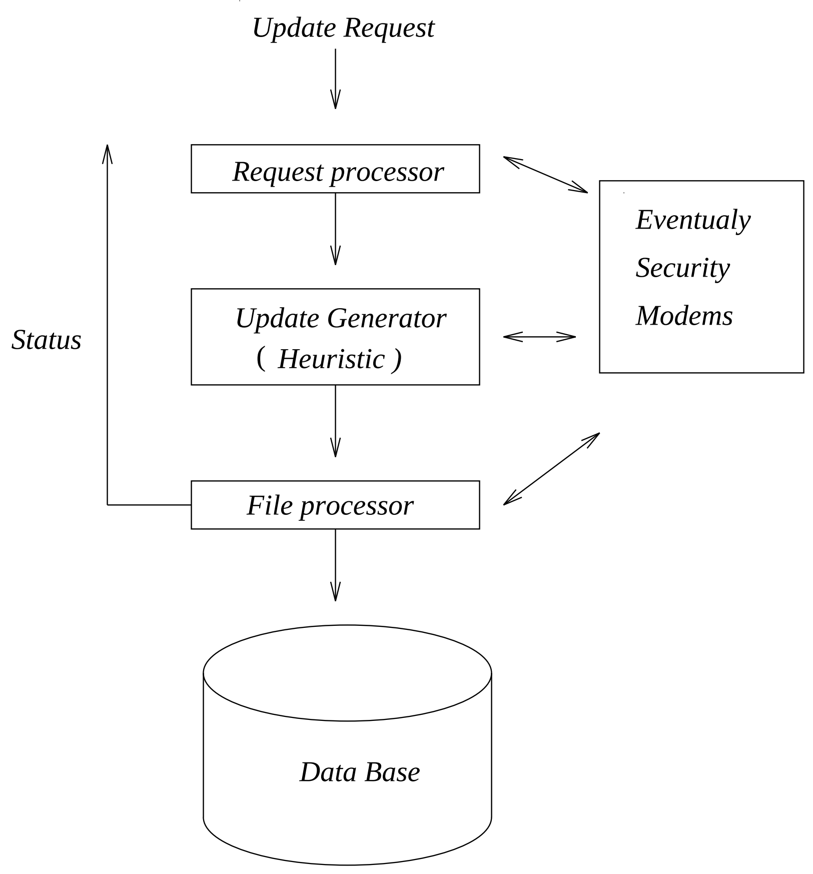


Figure 3.1: The update process

order to make the update succeed.
However, if there is constraint contradiction, the scheme or part of it will become the target for reviewing. This leads to the second stage.

The second stage is the analysis of consistency during the modelling process of a PSM scheme, where updates take place emphasizing the effects of the constraints check and scheme modification.
Another point to bear in mind is that by updating a scheme, the scope of control might be as short as possible avoiding oversizing and/or overpowering of the scheme.

The update process in the last stage regards another attention because of the level on which it takes place. At this point, the strategical heuristics have been followed and straight comparisons of the resulting relational database schemes have to be done.
After each update, these relational schemes are modified and kept in order to be reused in the next update.

Part II

Incremental consistency in PSM

Chapter 4

Incremental Techniques on data bases

In this chapter the basics of the Predicator Set Model (PSM) are given, they will not be extendendly explained. For further information about the PSM theory, see [12]. These heuristic techniques will be approached in three different ways:

1. Transformatiom in a PSM scheme during life.
2. Transformatiom in a PSM scheme during modelling process.
3. Transformatiom in a PSM scheme during transformation.

4.1 The Predicator Set Model

In PSM, a scheme $\Sigma = \langle \mathcal{I}, \mathcal{C} \rangle$ consists of an information structure $\mathcal{I}$ and a set of constraints $\mathcal{C}$. Any population of the scheme must fit within the information structure and satisfy the requirements specified in $\mathcal{C}$.

Definition 4.1.1 Information Structure

In the information structure, the following constructors are noted:

1. A finite set $\mathcal{P}$ of *predicators*.
2. A set $\mathcal{O}$ of *object types*, $\mathcal{L} \in \mathcal{O}$.
3. A partition $\mathcal{F}$ of the set $\mathcal{P}$. The elements of $\mathcal{F}$ are called fact types . Fact types are also object types, therefore $\mathcal{F} \in \mathcal{O}$.
4. A set $\mathcal{G}$ of *power types*. Power types are also considered object types, therefore $\mathcal{G} \subseteq \mathcal{O}$.

5. A function $\mathsf{Base} : \mathcal{P} \to \mathcal{O}$. The base of a predicator is the object part of that predicator.

6. A function $\mathsf{Elt} : \mathcal{G} \to \mathcal{O}$. This function yields the element type of a power type.

7. A partial order $\mathtt{Spec} \subseteq \mathcal{A} \times \mathcal{O}$ on object types, capturing *specialization*. $\mathcal{A}$ is the set of *atomic object types* and is defined as $\mathcal{O} \setminus (\mathcal{F} \cup \mathcal{G})$. There are two types of atomic object types:

   entities ($\mathcal{E} = \mathcal{A} \setminus \mathcal{L}$) and *labels* ($\mathcal{L}$).
   $\mathtt{sub}$ is a partial order of atomic object types, with the convention that $a \mathtt{sub}\ b$ is interpreted as a is a subtype of b. Note that the name *atomic* only refers to being undividable in the sense of not consisting of predicators (as fact types are).[14]

8. A partial order $\mathsf{Gen} \subseteq \mathcal{A} \times \mathcal{O}$ on object types, expressing *generalization*. The auxiliary function

   $$\mathsf{Fact} : \mathcal{P} \to \mathcal{F}$$

   is defined by:

   $$\mathsf{Fact}(p) = f \iff p \in f$$

   A fact type is called objectified, if it occurs as the base of a predicator. A predicator is called an objectification, if its base is a fact type.
   The set $\mathcal{H}$ contains all objectifications:

   $$\mathcal{H} \ = \ \{p \in \mathcal{P} \mid \mathsf{Base}(p) \in \mathcal{F}\ \}$$

   Each element of $\mathcal{A}$ has an associated (unique) top element, its *pater familias* and it is found by applying the function: $\sqcap : \mathcal{A} \to \mathcal{A}$.
   This function satisfies:

   (a) $a\ \mathtt{sub}\ b \Rightarrow \sqcap(a) = \sqcap(b)$

   (b) $a \neq \sqcap(a) \Rightarrow a\ \mathtt{sub}\ \sqcap(a)$

Concluding an information structure $\mathcal{I}$ is a tuple $\langle \mathcal{P}, \mathcal{O}, \mathtt{Sub}, \mathcal{F}, \mathsf{Base}, \sqcap \rangle$.

Definition 4.1.2 PSM Scheme properties

Population

A population $\mathtt{Pop}$ of an information structure $\mathcal{I}$ assigns in $\mathcal{O}$ a set of values of the universal domain to each object, conform the structure as prescribed in $\mathcal{P}$and$\mathcal{F}$, respecting the subtype hierarchy $\mathtt{Sub}$. This is denoted as:

$$\forall_{x,y \in \mathcal{A}} \left[x\ \mathtt{Sub}\ y \Rightarrow \mathtt{Pop}(x) \subseteq \mathtt{Pop}(y) \right]$$

This is referred to as the subtype rule.
The population of a composed object type is a set of tuples. A tuple t of a fact type f is a mapping of all its predicators to values of the appropiate type:

$$\forall_{f \in \mathcal{F}} \forall_{t \in \mathtt{Pop}\,(f)} \forall_{p \in f} [t(p) \in \mathtt{Pop}(\mathsf{Base}(p))]$$

This is referred as to the conformity rule.

Specialization

Specialization which in NIAM is referred to as subtyping, is a mechanism for representing one or more (possibly overlapping) subtypes of an object type. Specialisation is only to be applied on specific instances of an object type of those instances for which certain facts are to be recorded.
Specialization will be treated as common subtyping for constraint checking. Figure 4.1 shows an example of specialization.
The next section shows that for incremental updates, the subtypes do not affect constraints during life unless new constraints are required because of the new populations, or because possible failure of the identifiability.

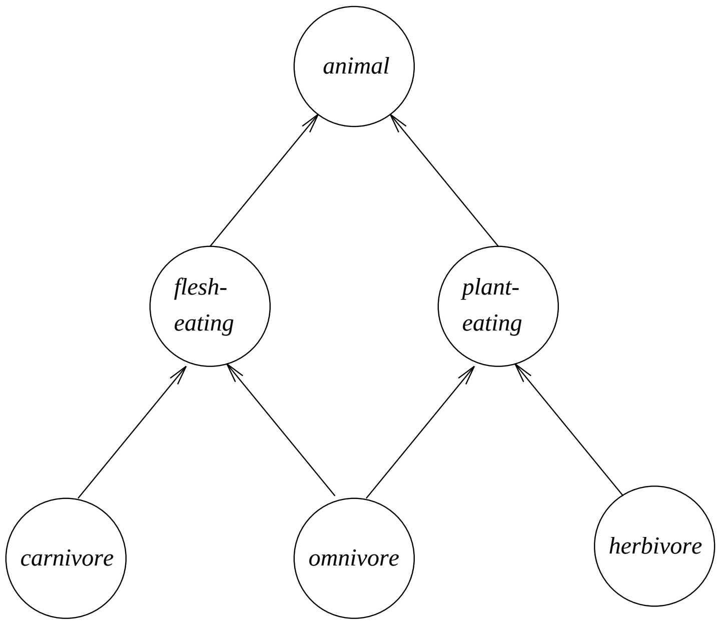


Figure 4.1: Example of specialization

Generalization

Generalization is a mechanism that allows the creation of new object types by uniting existing object types. For generalization it is typically required that the generalized object type is covered by its constituent object types (or specifiers). For constraint check purposes, the same heuristics as for specialisation will be followed. Figure 4.2 shows an example of generalization.

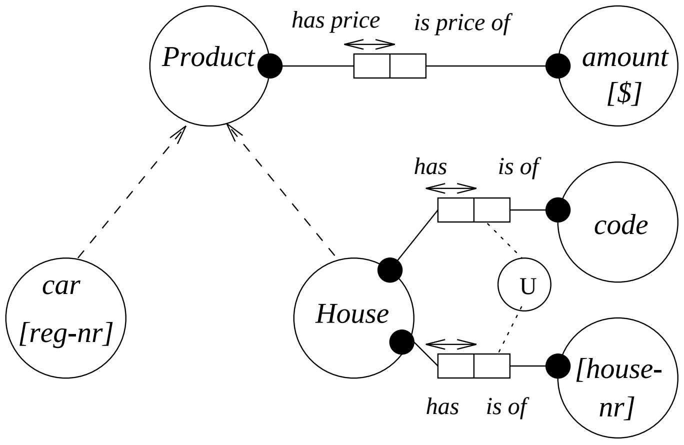


Figure 4.2: Example of generalization

Power types

The concept of power types in PSM forms the data modelling counterpart of power sets in conventional set theory. An instance of a power type is a set of instances of its element type. Such an instance is identified by its elements, just as a set is identified by its elements in set theory.

In this context, only static constraints will be analysed for consistency purposes. These constraints, from which forbidden populations are excluded, will be the kernel of this research.
Power types issues are not computer supported, that means that in order to express power types on a relational database, a translation to a more simple and well supported ordinary relation type must take place in advance as shown in figure 4.3.

Sequence types

Sequence typing offers the opportunity to represent sequences, built from an underlying element type. This notion is not elementary in PSM, as it is expressible in terms of

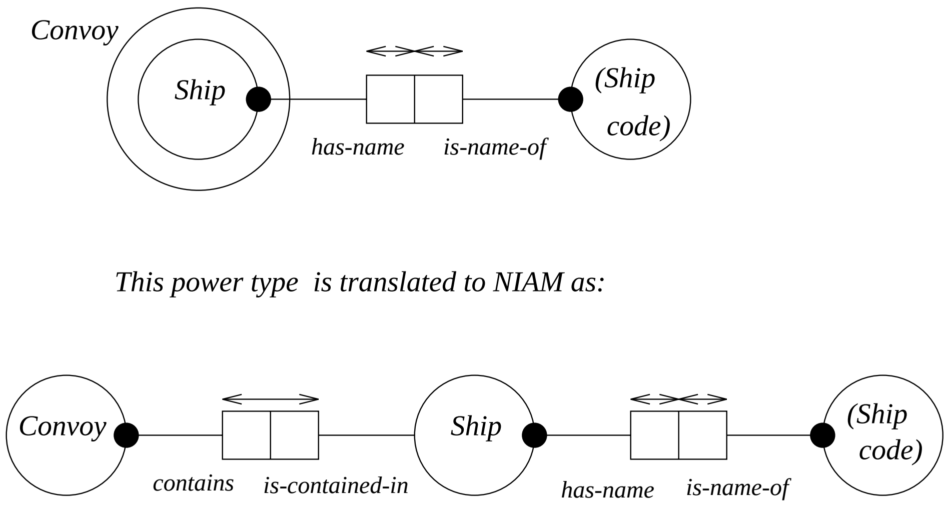


Figure 4.3: Representation of a power type in a simple relation type

generalisation.[12] The example of figure 4.4 shows how a sequence type can be translated to a well representational relation type; This figure has been taken from [10].

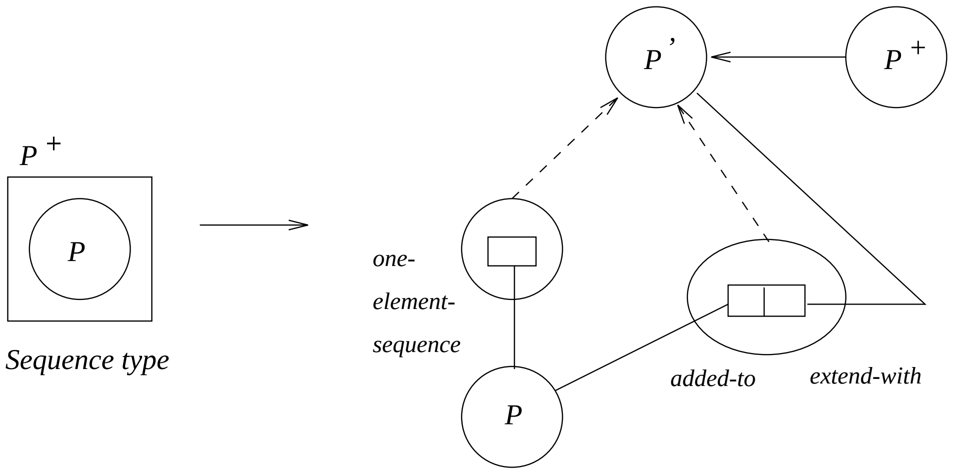


Figure 4.4: Representation of a sequence type in a simple relation type

4.2 Constraint checking

In this section an existing information structure diagram with its corresponding population and constraints is given. The representing scheme of this information structure will be updated in two ways:
The first one will be related to the population where data has to be added, removed or replaced, due to changes in the UoD of the user requirements.
During validation and verification, it must be checked whether or not the new information is valid and consistent. Updates in this sense are considered to be addition, modification or deletion. When updating only instances, such systems are considered to be traditional or snapshot systems because they do not take into account any concept of time. As a consequence, during an update of a state, the former state cannot be retrieved [2].

The second one will be related to the scheme itself. In evolving information systems, scheme specifications, activity models and behaviour specifications can be updated. The information systems allow updates like recording, correction and forgetting of all information recorded in the system [2]. This kind of scheme updates will be not considered on this research. This means time will be disregarded as an evolving factor for an information structure.

The scheme can not only be extended with new object types and with new relations to that scheme but also with new populations.
Indentation of object types and fact types leads to removal of a part of the population and loss of information. Most models consider only retrieval transactions but no update transactions such as deletion, insertion and modification.

4.3 Population update in a PSM scheme

In this section the updating of an existing PSM scheme is analysed. The framework for update is divided in three levels: event level, recording level and correction level [2]. For purpose of constraints checking and further update analysis, the time factor will not be considered. Since new events becomes requirements for a database, a PSM scheme will be extended with new object types, and new populations will be added.
Consider the next example where an incremental update is required. Imagine one has a relational database for Europe, consisting of countries with their cities. The capitals are marked. A country has a least one city. We assume that city-names are unique, so they appear in just one country.
Having created the scheme for Europe as shown in figure 4.5, all of a sudden a war breaks out in Yugoslavia and this country does not exist anymore and new countries have arisen.

One has to check whether or not the scheme will have to be adapted, starting with a constraint check. The constraints must be checked separately and put into a declarative form.
Consistency checking must be restricted to a section of the database as small as possible, keeping the response time during updates acceptable.

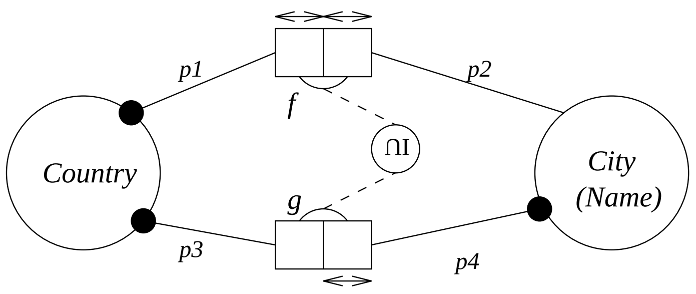


Figure 4.5: Example of scheme of countries and their cities

The original population values of the relational database are shown in figure 4.6.
The values of the instances in both fact types f and g have to be updated because of the splitting of the ethnical regions inside of Yugoslavia. For these tables this means that Yugoslavia have to be replaced by one of the country-names Croatia, Small Yugoslavia, Slovenia, Macedonia and Bosnia.

At first, the tuple $\langle Yugoslavia, Belgrado\rangle$ is removed from fact type f. If constraint checking would take place at this point, an inconsistency would be found; the subset constraint is violated. To solve this problem, the tuples $\langle Yugoslavia, Belgrado\rangle$, $\langle Yugoslavia, Ljubliana\rangle$, $\langle Yugoslavia, Zagreb\rangle$, $\langle Yugoslavia, Sarajevo\rangle, \langle Yugoslavia, Skopje\rangle$ in fact type g also have to be replaced by the new update tuples.
Executing the constraint check now would not reveal any inconsistencies. The update of this relational scheme is not complete at all. Due to the change of information needs, a new population should be added. As a result, new tuples will be introduced on fact type f. This is shown in figure 4.7.

The insertion of a new tuple in fact type f results in checking of predicator p_1 because it is a total role. Predicator p_2 is not checked because it is not total.
The uniqueness constraints are checked; p_1 and p_2 are both unique; this means names repetition on both sides are not allowed.

Before executing the subset constraint checking, the tuples in fact type g have to be added. Adding a new tuple in fact type g requires the checking of the total roles p_3 and p_4.

f

Country	*City*
Holland	*Amsterdam*
France	*Paris*
Yugoslavia	*Belgrado*

g

Country	*City*
Holland	*Amsterdam*
Holland	*Nijmegen*
France	*Paris*
France	*Calais*
Yugoslavia	*Belgrado*
Yugoslavia	*Ljubliana*
Yugoslavia	*Zagreb*
Yugoslavia	*Sarajevo*
Yugoslavia	*Skopje*

Figure 4.6: The original population values

f

Country	*City*
Holland	*Amsterdam*
France	*Paris*
S. Yugoslavia	*Belgrado*
Slovenia	*Ljubliana*
Croatia	*Zagreb*
Bosnia	*Sarajevo*
Macedonia	*Skopje*

g

Country	*City*
Holland	*Amsterdam*
Holland	*Nijmegen*
France	*Paris*
France	*Calais*
S. Yugoslavia	*Belgrado*
Slovenia	*Ljubliana*
Croatia	*Zagreb*
Bosnia	*Sarajevo*
Macedonia	*Skopje*

Figure 4.7: The updated population values

Because of the uniqueness constraints on p_4, city-names repetition is not allowed.

In this example three different constraints have been checked: total, uniqueness and subset constraints. The checking also took place in that order:

1. *Total-role* to determine a *kind of environment* to see how far an update will take place on a scheme.
2. *Uniqueness* to remove invalid populations and
3. *Remaining constraints*

These ideas will be worked out in the next chapters.

Chapter 5

Incrementation in a PSM scheme during life

In this chapter the populations of a scheme will be analysed in order to see the modification steps that results by an update request. Heuristics will be proposed to reduce the environment in the database where the adaption will take place. Furthermore, an algorithm will be given for determining an area where constraints have to be checked. Different constraint types will be analysed how to be checked after an update.

5.1 PSM scheme evolution

To obtain heuristics for determining the check area, an example has been used. Consider the structure of figure 5.1 where the update on it consists of grouping countries according to their membership to the European Community (EC). This means, in order to update the structure, the former object type on which the new relation will be built has first to be searched.
For this purpose, object type *Country* is chosen. The new relations are denoted as subtypes. Each country on this database should be member or not of the EC. By executing the constraints checking at this point, no inconsistencies are found. The sub-typing update did not affect the consistency and did not contradict the other constraints at all. Just new populations are created referring memberschip.

Consider now new requirements for this information structure be from sportive ground, on which each city has a football club and each country has 'representative' footbal players. All players belong to a football club.
In figure 5.2, a new tuple will be added to the football players population with the knowledge of the dutch player Koeman plays for the club *Barcelona* from Spain.
The tuple $\langle Koeman, Holland \rangle$ will be added on fact type h. If constraint checking would take place at this point, the first inconsistency would appear: the constraint

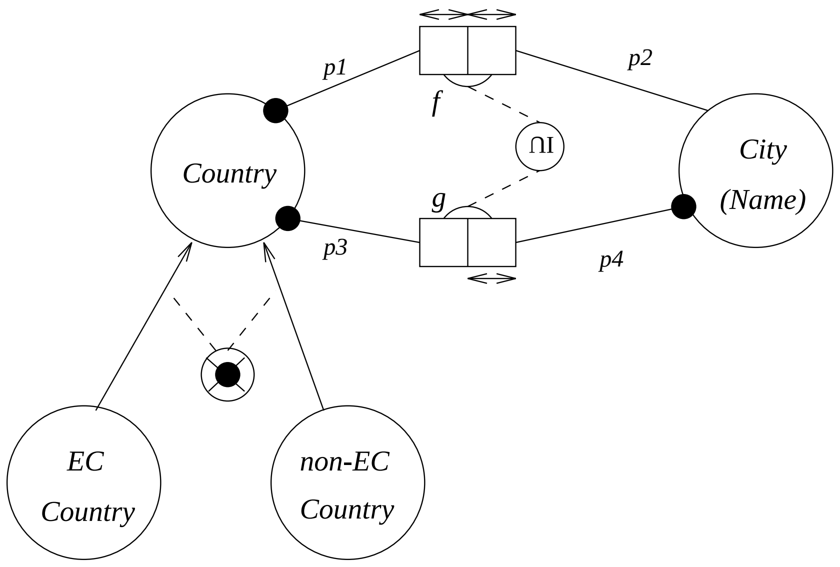


Figure 5.1: Grouping to EC membership

contradict each other.
As conclusion it can be said that in order to execute this update: the club Barcelona might have been also recorded. This also means the city Barcelona might have been recorded. Consequently, the country Spain might have been also recorded in advance. Finally it must be determine whether or not Spain is a EC-member.

A *Chain* of updates should have taken place in advance. By adding the tuple $\langle Koeman, Holland \rangle$ in fact type h, predicators p_5 and p_6 were checked. Country Holland was already recorded.

Because of the total role on predicators p_7 and p_8, also a tuple must be added on fact type i. Predicators p_9 and p_{10} are also total and a tuple population in fact type j had to be added. In this case there is no uniqueness constraint, so the remainig constraints can be checked. By meantime, all the populations become kept and if some inconsistency would appear then it could be easily to update (remove, add or correct) the population of the corresponding fact types.

5.2 Determining a check area

The comprehension *Chain* has been introduced. By updating the population on a PSM scheme, a sequence of updates is originated and the next analysis of this update behaviour can be noticed: The *environment* on which the update will take place have to be determined. The update takes place through the whole environment.

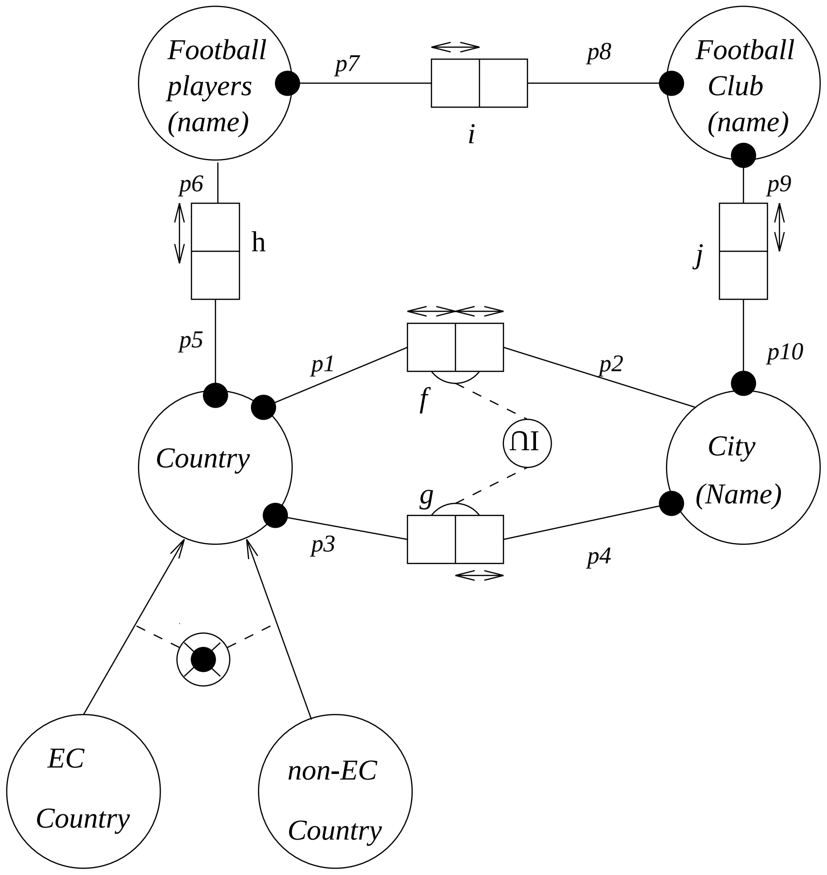


Figure 5.2: The extended scheme

Definition 5.2.1 The chain of updates

Let τ be a non-empty set of $\mathcal{P}$ with $\texttt{total}(\tau)$: see figure 5.3.
$p_i \in \mathcal{P}$ and $i \in N$ form a *Chain* if:

$$\begin{aligned}&\texttt{total}(p_1) \wedge p_2 \in \mathsf{Fact}(p_1) \wedge \mathsf{Base}(p_2) \sim \mathsf{Base}(p_3)\\ \wedge\ &\texttt{total}\ (p_3) \wedge p_4 \in \mathsf{Fact}(p_3) \wedge \mathsf{Base}(p_4) \sim \mathsf{Base}(p_5)\\ \wedge\ &\texttt{total}\ (p_5) \ldots\end{aligned}$$

The *Chain S* corresponds to the next formula:

$$\forall_{1 \leq i \leq \texttt{length}(S)-2\ ,\ i \in \mathrm{N}\ \mathsf{and}\ p_1 \in S} [\texttt{total}\ (p_i) \wedge p_{i+1} \in \mathsf{Fact}\ (p_i) \wedge \mathsf{Base}\ (p_{i+1}) \sim \mathsf{Base}\ (p_{i+2})]$$

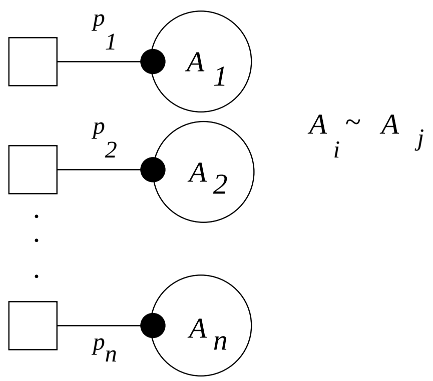


Figure 5.3: Total roles

The same formula can be applied to determine the update environment if other constraints are concerned. In that case, the parameter *total* can be substituted by the corresponding constraint.

In the situation of figure 5.4

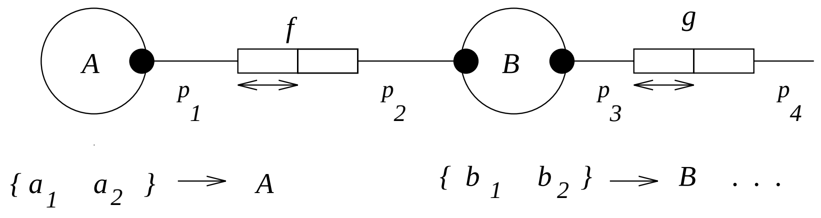


Figure 5.4: Insertion of new populations

If a new population is put on Base (p_1), then a complete update can be originated throughout the *chain*. In other words, the environment on which the update will take place is determined at first.

This heuristic is quite relevant also for purpose of extending a PSM scheme with a fact type or with an object type as in the next part will be explained.

5.3 The environment algorithm

To determine the checking environment, the next heuristic algorithm is proposed:
The *Chain* is extended :

1. Throughout all total roles where in the next fact type an update takes place.

2. The predicator (let $p \in \mathcal{P}$, If $\forall_\tau$, total$(\tau), p \notin \tau$), where there is a relationschip of the new element in p.

3. Let $p_i \in \mathcal{P},\ p_i \in \mathcal{C}$

   If there is a constraint in the chain: constraints $\{p_1 \ldots p_n\}$ where a p_i can be found which is related with a predicator or a fact type beyond the chain, then the chain will be extended to that fact type.

The extension of the environment is upto the next object type if:

- The element in the predicator of the next first fact type already was recorded in the population of the adyacent object type.

- The extension goes upto the next object type if the element still has to be added.

It holds then as algorithm:
<u>if</u> 1 <u>or</u> 2 <u>or</u> 3
<u>then</u> extension of the environment
<u>else</u> reprocess the algorithm from step 1 for the next update.

As a consequence:

- All the constraints must be checked independently; In case there are no more updates, the algorithm will stop.

- If an update takes place by influence of a constraint, then all the constraints on that fact type must be checked. If as consequence of this action, new populations must be added to an object type, the algorithm will start again from the first step.

- If the algorithm stops then the checking environment has been determined.

- After each insertion, the new fact types and object types have to be checked again in order to extend the environment according to the algorithm.

5.4 Check environment and constraints

In this section, the constraints will be first treated separatelly. Incrementation at this stage is concern to the populations. Therefore, the concordance betwee constraints and update instantiations might harmonize.

Total role constraint

A population that satifies this constraint must be checked by:

$$\bigcup_{q \in \tau} \texttt{Pop}\ (\mathsf{Base}(q)) = \bigcup_{q \in \tau} \texttt{Pop}\ (\pi_q(\mathsf{Fact}(q)\ )\ )$$

Just the updated population must be checked, the former population was suppose to be correct. By no inconsistency with other remaining constraints the tuple will be recorded.

For the total role constraint, the old information is in fact not relevant for being kept, just the new element of the update is important for control checking of remaining constraints.

Uniqueness constraint

For the uniqueness constraint on PSM schemes, two types can be recognized: Uniqueness constraints where δ ($\texttt{Unique}(\delta) \wedge \delta \subseteq \mathcal{P}$) does not exceed the boundaries of a single fact type f (see [12]). On that way, the uniqueness constraint is bound to the relation $(\xi(\delta) = f)$. The elementary fact type is given in figure 5.5.

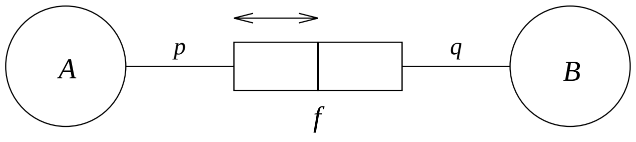


Figure 5.5: A single fact type uniqueness constraint

In this case, the uniqueness constraint is checked independently like in the european databes example, where name repetitions are not allowd. By adding an element a_1 to object type A then the next heuristics can be followed:
Search all $b_1 \in$ $\texttt{Pop}\ (\pi_f(p))$ with $a_1 = b_1$. In case that no b_1 is found then a_1 can be added, otherwise no update will take place because a_1 was already recorded.
The second type is the uniqueness constraint for derivated types.
Let σ be a non-empty set of predicators $\sigma \in \mathcal{P}$ where it holds:

$$\xi\ (\sigma) = \sigma_{\mathcal{C}(\sigma)} \bowtie_{f \in \mathsf{Facts}\ (\sigma)} f$$

Then, by adding an arbitrary tuple to the result of $\xi\ (\sigma)$, just the elements in the update tuple will be checked that corresponds to the combination of edges of $\xi(\sigma)$. This is shown in figure 5.6:

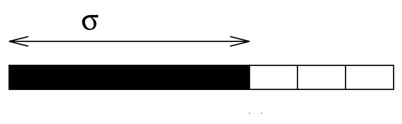


Figure 5.6: Derivated fact type

Let $\xi\ (\sigma) = R$, where R is the derivated fact type. Let $x_1 \in \sigma$ add $\langle x_2\ ,\ x_3 \rangle$ to R, for a derivated fact type, the next situation must be taken into account:

<u>if</u> $x_1 = x_2$
<u>then</u> tuple was recorded
<u>else</u> menace of constraint violation.

In figure 5.7, an example of the uniqueness constraint with derivated fact types is given:

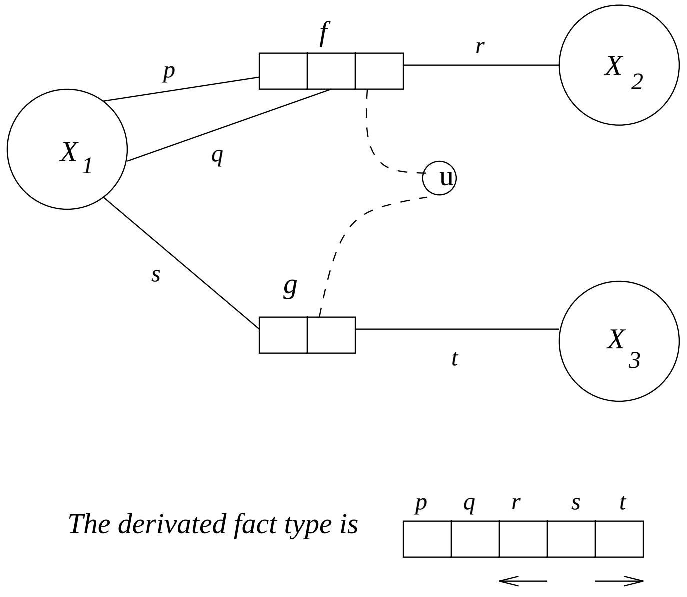


Figure 5.7: Uniqueness combination

`Unique` (σ) where $\sigma = \{\ r,\ t\}$. The semantic of figure 5.7 is :

$$\xi\ (\sigma)\ =\ \sigma_{\ p=q\ \wedge\ q=s}\ (f \bowtie g)$$

By adding a new tuple $\langle a_1, a_2, a_3, a_4 \rangle$ to this derivated fact type, the combination on roles r and t must be checked, that means repetition of thees populations on this set are not allowed. For purpose of scheme evolution, the incremental approach concerns now about the question of what information to keep as intermediate result in order to re-use it for later checking.
When a scheme is updated, it consequently becomes bigger. New populations with new restriction arises. It is then worthless to keep all the invalid former populations unless no memory costs are important. For a 'low budget' memory it is better to make a choice of what is going to be kept in memory. For the uniqueness constraint, the population which concerns the uniqueness combination must be kept, the rening values which do not take part of this combination does not have to be recorded, the derivated fact type guarantees the consistency.

Set constraints

Let σ and τ set of predicators, for the set constraints

$$\texttt{Subset}_{\phi}\,(\sigma\ ,\ \tau)\ ,\ \texttt{exclusion}_{\phi}\,(\sigma\ ,\ \tau)\ ,\ \texttt{equal}_{\phi}\,(\sigma\ ,\ \tau)$$

it holds checking of constraint consistency by searching a mapping between the predicators in σ which concerns the constraint as shown in the graph of figure 5.8:

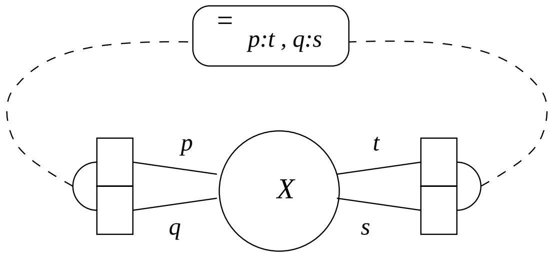


Figure 5.8: Example of equal constraint

Occurrence frequency constraint

This is merely a restriction constraint. Let σ be a non-empty set of predicators involve on this constraint. This is denoted as:

$$\texttt{frequency}\,(\sigma\ ,n\ ,m)$$

By adding a new tuple, it will be checked whether or not the new tuple appears at least n and at most m times in this set.
In this case, keeping of old information by updates is not required, even if this constraint is applied in combination with other constraints.
Figure 5.9 gives an example of occurrence constraint.

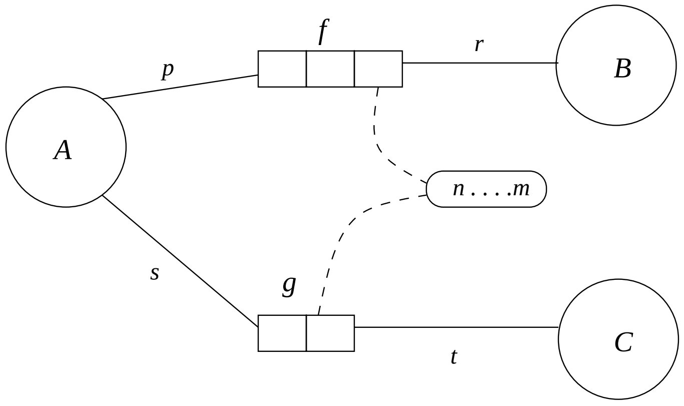
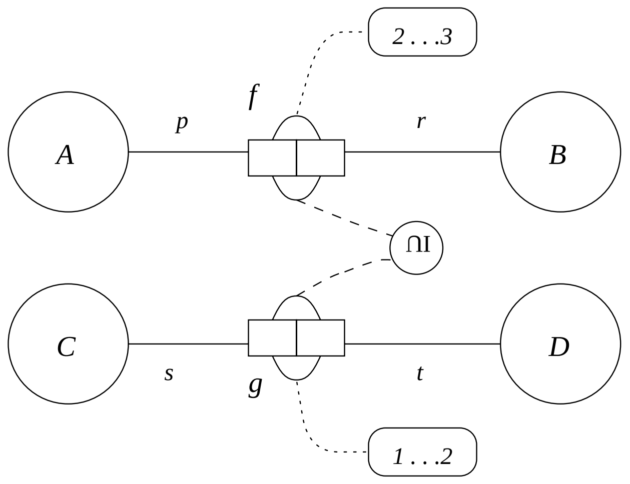


Figure 5.9: Example of occurrence constraint

It can be possible that for this constraint, contradicting occurrences are placed. The schemes under these situations must keep a valid population that satisfies both occurrences as shown in figure 5.10.



Population for f :	*Population for g :*
$< a_1, b_1 >$	$< a_1, b_1 >$
$< a_1, b_1 >$	$< a_2, b_2 >$
$< a_2, b_2 >$	$< a_3, b_3 >$
$< a_2, b_2 >$	
$< a_2, b_2 >$	

Figure 5.10: Example of contradicting occurrences

Chapter 6

Incrementation in a PSM scheme during modelling process

In this section, the update of an information structure at scheme level during the modelling process will be analysed. At this point an update would not seem to be necessary since the information structure at this level is not finished at all. Nevertheless as long as new specifications arise or forgotten items must be included or deleted, the scheme takes new proportions.

When an additional fact type or object type is inserted between two object types, it is customary not to leave just extra space between the corresponding items but it is a preamble to analyse whether or not the insertion will succeed.

In order to determine the consistency of the new resulting scheme, the incremental approach will be applied, in this context the *Object Relation Network* (*ORN*) of each resulting scheme by each update step will be kept for further comparison with the next first update on the same scheme. The *ORN* makes it possible to visualise complex operations on information structures, which are necessary to determine the semantics of some graphic constraints.
The nodes of this network are: $\{x \sim | \neg\, \mathtt{gen}(x)\}$, where $x \sim$ is the set $\{y \mid x \sim y\}$ of all object types that are type related with x. An edge, labelled p is drawn from node N to node M iff p is a predicator with $\mathsf{Base}(p) \in N$ and $\mathsf{Fact}(p) \in M$. This implies that the leaves of this network contain only elements from $\mathcal{O} - \mathcal{F}$. All non- leaves correspond to fact types [12].

6.1 Update heuristics in a PSM scheme

For this purpose the next heuristics will be applied:

1. The environment on which the update will take place has to be determined. One must try to keep the size of this environment as small as possible in order to make

the sub-scheme area of control limited.

2. This environment, after some update must remain *Identifiable*.
3. The resulting scheme must be *populatable*.

Theorem 6.1

Suppose scheme Σ is identifiable and also populatable, then:

(a)

$$\texttt{Identifiable}[\texttt{Environment}(\texttt{update})] \Longrightarrow \texttt{Identifiable}[\texttt{new scheme}]$$

(b)

$$\texttt{Populatable}[\texttt{Environment}(\texttt{update})] \Longrightarrow \texttt{Populatable}[\texttt{new scheme}]$$

Where new scheme is the result after updating scheme Σ.
Before the proof of this theorem can be formalised, the delimiters around these heuristics might be defined. Therefore it is convenient to analyse the behaviour of the check environment in advance.
The space used by an *environment* is the scope of the update; This update can be of two types:

- When a fact type is updated and
- When an Atomic object type is updated

6.2 Object type update

The object type update consists of a Fact type update and an atomic object type update. The environment must be determined in those two cases and the identifiability and the populatibility must be checked for the new scheme.

6.2.1 Fact type update

Since new requirements are being specified, adding a binary fact type is the most trivial way of relating two object types. By adding a binary fact type, the environment will be extended by the two connected object types. By adding an arbitrary n-ary fact type between (different) objects, the environment will be extended with all those connected object types.
Furthermore, after adding the fact types, the constraints can be added, which may influence the environment size.

It can be concluded that by adding a new fact type, the environment of constraint checking is determined by:

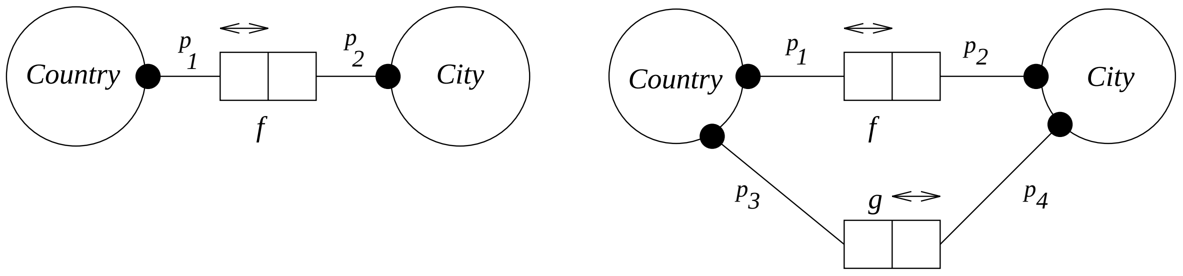


Figure 6.1: Example of fact type update

1. All object types on which the new constraint is supposed to be related with.
2. All fact types that are connected with the object types mentioned above via a *chain*.

The three heuristics previously mentioned can now be worked out.

Environment

In this environment, all the constraint connections between the different fact types must be checked. This will be executed according to the environment algorithm. It might be possible that a constraint must be added to the scheme; in that case, the checking will then be continued with another constraints on the same object types.

In case of insertion of an 6.2, the *ORN* changes automatically, taking care that the new scheme remains correct. The semantic of figure 6.2 is:

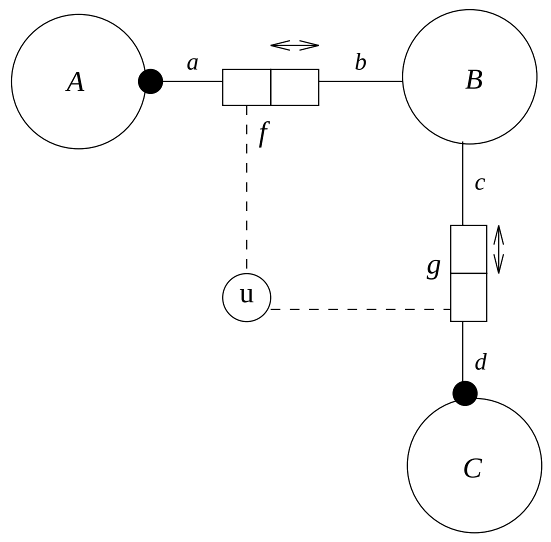
$$\xi(\alpha) = \mathcal{C}_{\ b=c}\ \ (\ g \bowtie\ h\ )$$

Assume object types $A_1, \ldots, A_n$ are connected with fact type *f*.
The predicators of $A_1, \ldots, A_n$, are the only ones which are concerned with all constraints.

Only for the `uniqueness` constraint, all fact types of the whole scheme are important. In this case, *all* existent uniqueness-combinations of *those* fact types must be checked.

$$\xi(\alpha) = \mathcal{C}_{\ b=c\ \wedge\ c=e}\ \ (\ g \bowtie\ h\ \bowtie\ f\ )$$

In the figure 6.3, it is shown that it is possible to deduce a selection of new predicators in the uniqueness constraint. In other words, it can easily be seen which predicator cannot exist in the uniqueness constraint.



The ORN of this scheme is:

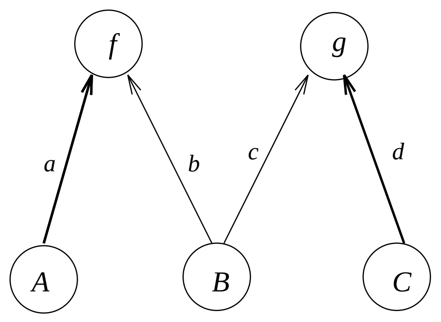


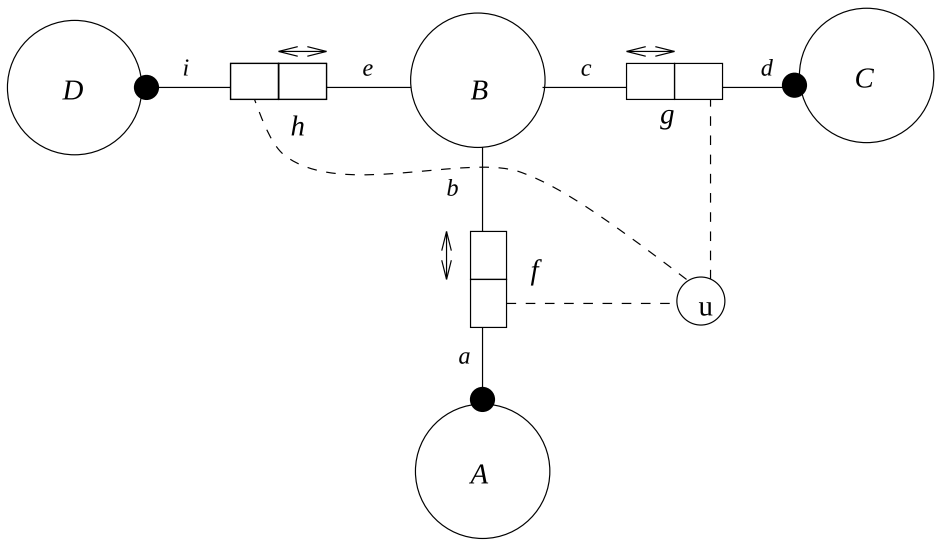
Figure 6.2: Example of a scheme with ORN

The original *ORN* can be kept to make new *ORN*'s with the added predicator(s) and with the new fact type(s). The original *ORN* then will be kept as long as the new *ORN* has no more joinable descendants (or otherwise the new *ORN* would be incorrect).

Only if a correct *ORN* would result, in other words, an *ORN* with joinable descendants, the former *ORN* will be replaced by the new one. Figure 6.4 shows an adding with its corresponding *ORN* with no joinable descendants. Thus, the environment can be limited by observing the *ORN* during the evaluation of the uniqueness constraint.

Identifiability

In this section the new scheme and the *ORN* must be checked for identifiability by structural identification:



The ORN of this scheme is:

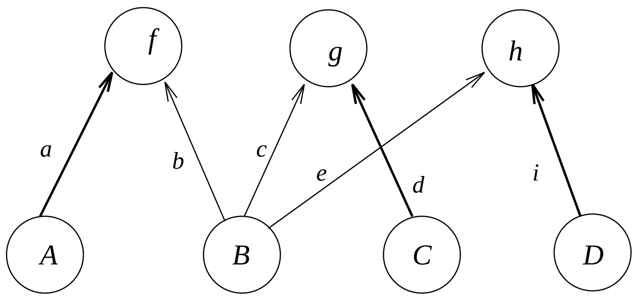


Figure 6.3: Augmented scheme with new ORN

1. Each label type should occur in some total role constraint:

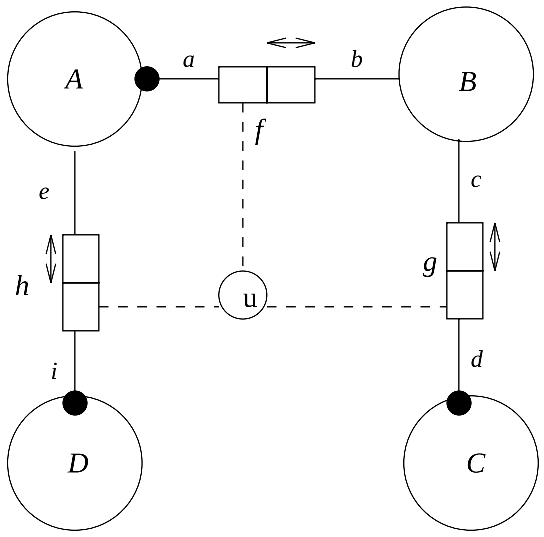
$$\forall_{x \in \mathcal{L}} \exists_{p \in \mathcal{P}} \exists_{\texttt{total}(\tau) \in \mathcal{C}} \left[\mathsf{Base}(p) = x \wedge p \in \tau\right]$$

2. All entities can be identified:

$$\forall_{x \in \mathcal{E}} \left[\texttt{Identifiable}(x)\right]$$

   where the predicate `Identifiable` is defined by:

   (a) If x is a label type, then $\texttt{Identifiable}(x)$.



The ORN of this scheme is:

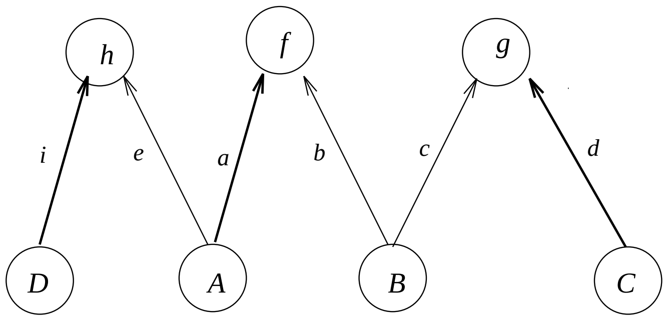


Figure 6.4: Incorrect update with ORN

(b) If x is a composed object type (or generally a set of predicators) then

$$\forall_{p \in x} \left[\texttt{Identifiable} \left(\mathsf{Base}(p) \right) \right]$$

(c) If x is an entity type, then there are two cases:

i. If x is a subtype, then x is identifiable if its associated pater familias is. In addition, there should be a unique subtype defining rule.

ii. If x is not a subtype, then x can be identified, if there exists a set τ of predicators that can be used for this purpose, a identifier for x:

- $\tau \neq 0$
- $\texttt{unique}(\tau)$

- $\mathtt{Identifiable}(\tau)$
- $\forall_{f \in \mathsf{Fact}(\tau)} \exists_{p \,\in\, \mathtt{Compl}\,(\tau,x)} [\mathtt{unique}(p) \wedge \mathtt{total}(p)]$

The set $\mathtt{Compl}(\tau, x)$ of co-roles with respect to x is defined as

$$\mathtt{Compl}(\tau, x) = \{p \in \bigcup \mathsf{Fact}(\tau) \backslash \tau \mid \mathsf{Base}(p) = x\}$$

This checking is quite feasible since the added fact type belongs to an existing object type which is supposed to be Identifiable. If the Object type is identifiable then also the fact types that belong to it. The formula should then be applied only within the environment check.

Populatability

The updated scheme must be checked for populatability. Only a fact type has been added, the checking is done by:

$$\mathtt{LocFactPop}(\Sigma) \equiv \forall_{f \in \mathcal{F}} \exists_{\mathtt{Pop}} [\ \mathtt{IsPop}\ (\Sigma, \mathtt{Pop}) \wedge \mathtt{Pop}(f) \neq \phi]$$

Populatability at the fact level is a stronger property than at atomic level as it will be shown in the next section. If the sheme is $\mathtt{LocFactPop}(\Sigma)$ then is also $\mathtt{LocAtomPop}(\Sigma)$. Adding a new fact type, does not influence the populations in other fact types. This means $\mathtt{GlobFactPop}(\Sigma)$ remains to be valid.
Adding an atomair object type, does not influence the $\mathtt{GlobFactPop}(\Sigma)$ as well, because it is not even connected with the graph.
But, adding a constraint, may influence the scheme on a way that the $\mathtt{GlobFactPop}(\Sigma)$ will loose validity. This is shown in figure 6.5.

Let $f\ : \{\ \langle a_1, b_1\rangle\}\ \ g : \{\ \langle a_1, b_1\rangle\}$ then the tuple $\langle a_1, b_1\rangle$ may only exist in fact type f or in fact type g, but not in both because of the exclusion constraint.

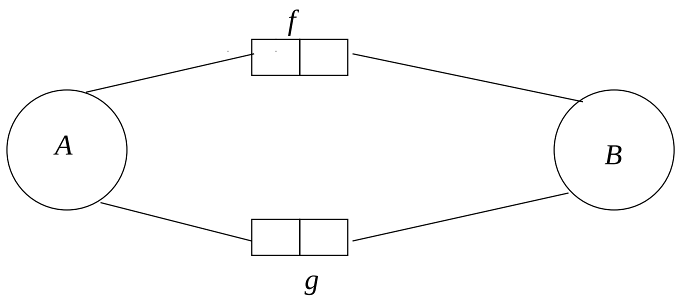
6.2.2 Atomic Object type update

For the update of an information structurewith new object types at scheme level, the same procedure as in the previous section will be applied. There is an small variation in the environment determination step:

Environment

Determine the kind of the new atomic object types, the possible classes are:

1. **Label type**: In this case, the analysis of possibilities to joining or splitting with other label types are done. An existing label type could be splitted in more label types on wich the new label type could be joined with. At this point the step seven of the NIAM process can be applied. Further can be seen whether the new label type belongs or can be put under another former label type.



With the exclusion constraint, the scheme is not GlobFactPop :

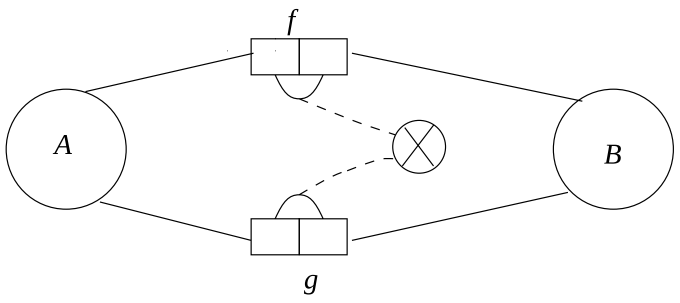


Figure 6.5: Non Global Fact Population Scheme

2. **Entity type**: In this part, the rules of NIAM holds also for PSM where three cases must be analysed:
   (a) Object types will be treated as entity types.
   (b) If there is no Generalization or Specialization then determine similarly entity types for possible joining.
   (c) Determine possible Generalizations or Specializations of the new entity types with the existing entity types. If there is a Specialization or Generalization then add the new entity type in this new Gen/Spec structure with their corresponding constraints.

3. Determine the relationschips with all object types with the new atomic object type. Translate these relationschips in fact types which can be unary, binary, or n-ary.

4. The new fact types will be checked per fact type according to the heuristics of the previous section. Thus, the new constraints will be added and the neccesary constraints will be checked within the environment.

For adding one entity type, it is necessary to add two fact types and the corresponding uniqueness constraints. After this checking, the whole update must be checked for

identifiability.
All these entity type updates could lead to confusions because of the additional fact types and uniqueness constraint combinations which also belong to an update of an entity type. To solve this problem, all entity types will be considered like in figure 6.6.

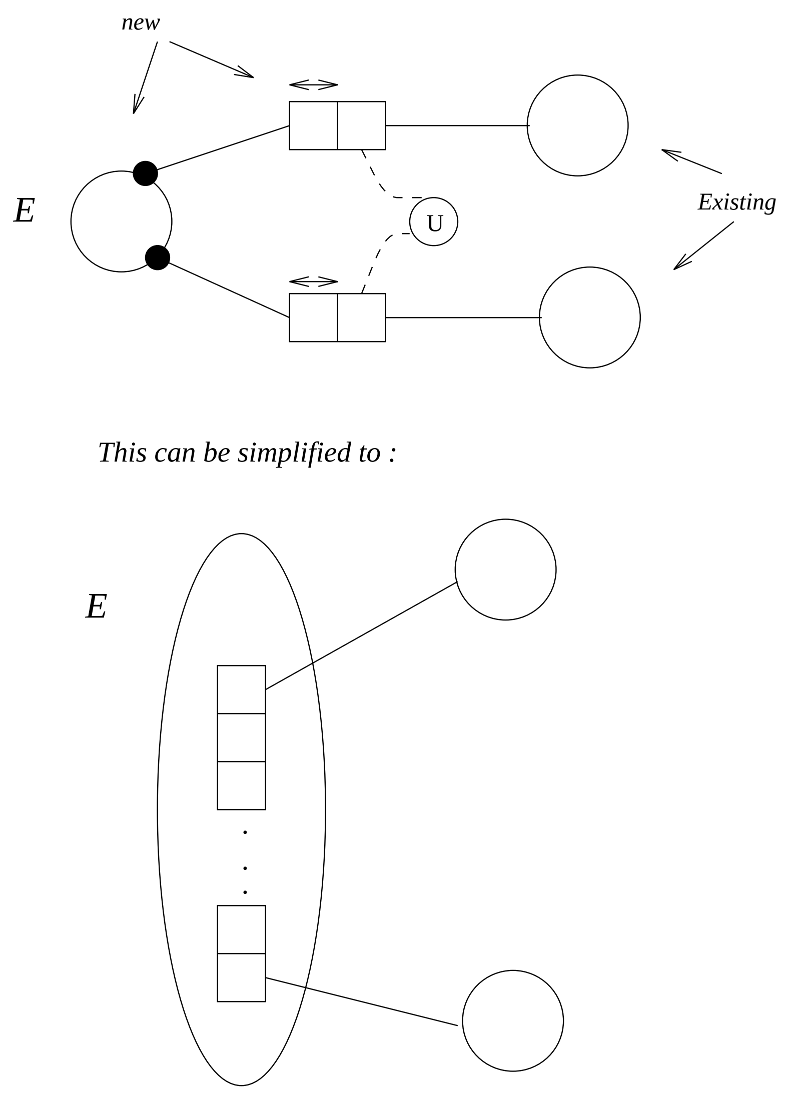


Figure 6.6: Example of entity type

Identifiability

Only the object types will be checked to see whether or not they are identifiable. This is done by:

Suppose x is the new object type, then it can be concluded $\mathtt{Identifiable}(x)$ if all components of x are identifiable:

$$\forall_{p \in X}[\mathtt{Identifiable}(\mathsf{Base}\ (p))]$$

To ensure termination of this checking, the same procedure as for fact type update must be executed.

Populatability

For populatability only the fact types will be checked to see whether or not they are `GlobFactPop`. The same procedure checking as for fact type update must be executed.

The framework for the prove of theorem 6.1 has been given. Update of the differnt types, identifiability and populatability have been analysed independently. It can be then concluded:

Proof of theorem 6.1:

The analysis of this section provides an outline of a proof for:
Let Σ' be the new scheme, retrieved from the scheme Σ after updating.

(a) For Identifiability :

$$\mathtt{Identifiable}[\Sigma] \wedge \mathtt{Identifiable}[\mathtt{Environment}(\mathtt{update})] \Longrightarrow \mathtt{Identifiable}[\Sigma']$$

(b) For Populatability :

$$\mathtt{Populatable}[\Sigma] \wedge \mathtt{Populatable}[\mathtt{Environment}(\mathtt{update})] \Longrightarrow \mathtt{Populatable}[\Sigma']$$

□

Chapter 7

Incrementation in a PSM scheme during transformation

Incremental consistency of a PSM scheme can be achieved by applying a sequence of heuristics that concerns updates of fact types or atomic object types as shown in the previous chapter. In this section an approach of a transformation of a PSM scheme at internal level will be given.
Transformation is a wide concept, meaning it deals with different aspects in databases as well as in other issues.
In the incremental consistency of PSM schemes, it can be said that there are three kinds of transformations:

1. Transformation at conceptual level.
2. Transformation at relational database level.
3. Transformation to another less expressive modelling technique (language).

This paper intents to describe the effects of the incremental patterns, this is explained in the second item. The third item deserves more accurate attention but is not quite in the scope of this research. Now first a review of transformation at conceptual level.

7.1 Transformation at conceptual level

In this section, the transformation caused by updates on PSM scheme will be dealt with. Until now, the analysis of updates was focused on checking the consequences of adding objects or populations when more requirements were imposed to a system; these updates took place in steps, but the possible operations on the scheme were not taken into account.
As depicted in [3], data transformation is a dichotomy; transformation of data schemes and transformation of data operations.

At the updating a PSM scheme because of new requirements, the possibility of changing the original structure of the relationschips exists. So the possibility that a binary relationschip might become a ternary one or another n-ary one, has to be considered.
To enforce and support extending of the system at conceptual level, PSM counts with constructors like: Power types, Sequence types, Generalisation and Specialisation which will be treated as common objects in order to be able to apply the proposed algorithm to make the update succeed.

Step four of the information analysis of NIAM gives the corresponding techniques which also can be used for the PSM schemes.
In addition of the splitting and re - grouping techniques, there are also the formal relational type transformations:

- Relationschip - to - object reduction
  Object - to - relationschip composition.
- Nesting / flattening.
- Object - to - role reduction
  Role - to - object composition.

Consider the next examples, where a summary of these techniques is shown. The start situation in figure 7.1 is the relation between *class*en *date*

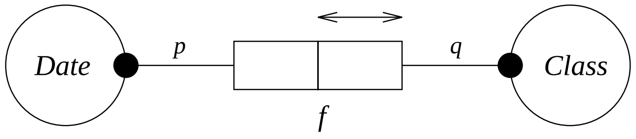


Figure 7.1: The begin situation

Consider now the scheme to suffer from variations because of the introduction of new requirements; then, the entity *class* can be extended to the next situation (see figure 7.2).
Due to the NIAM steps, this extended scheme presents some redundances that have to be removed (as shown in figure 7.2).
The constraint checking takes place in steps as is explained in the previous chapter.
Combination of possible new constraints will also be checked within the checking environment.

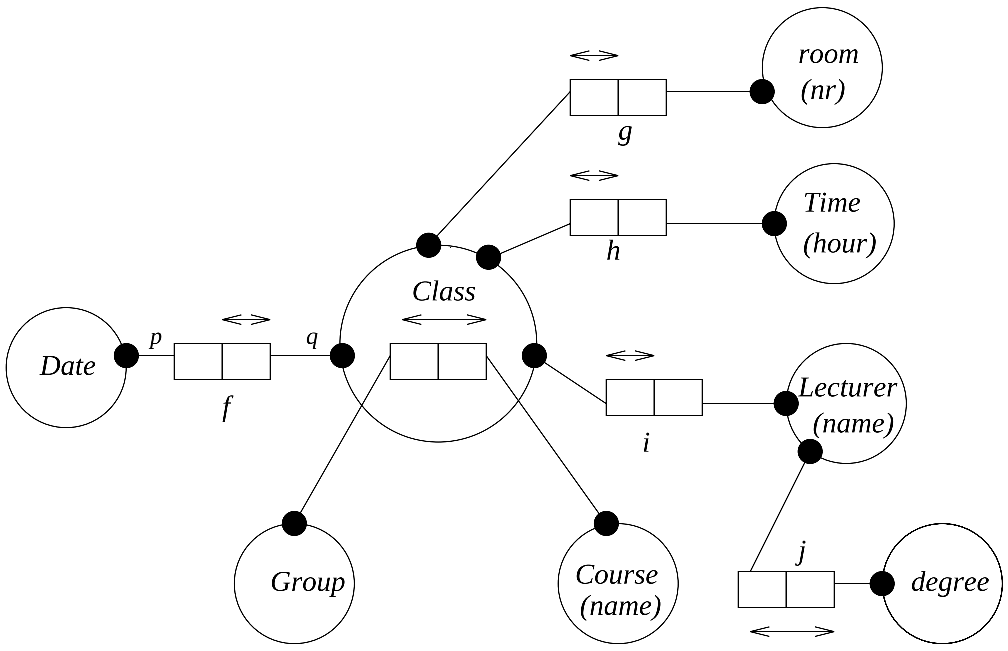


Figure 7.2: Example of an extended scheme

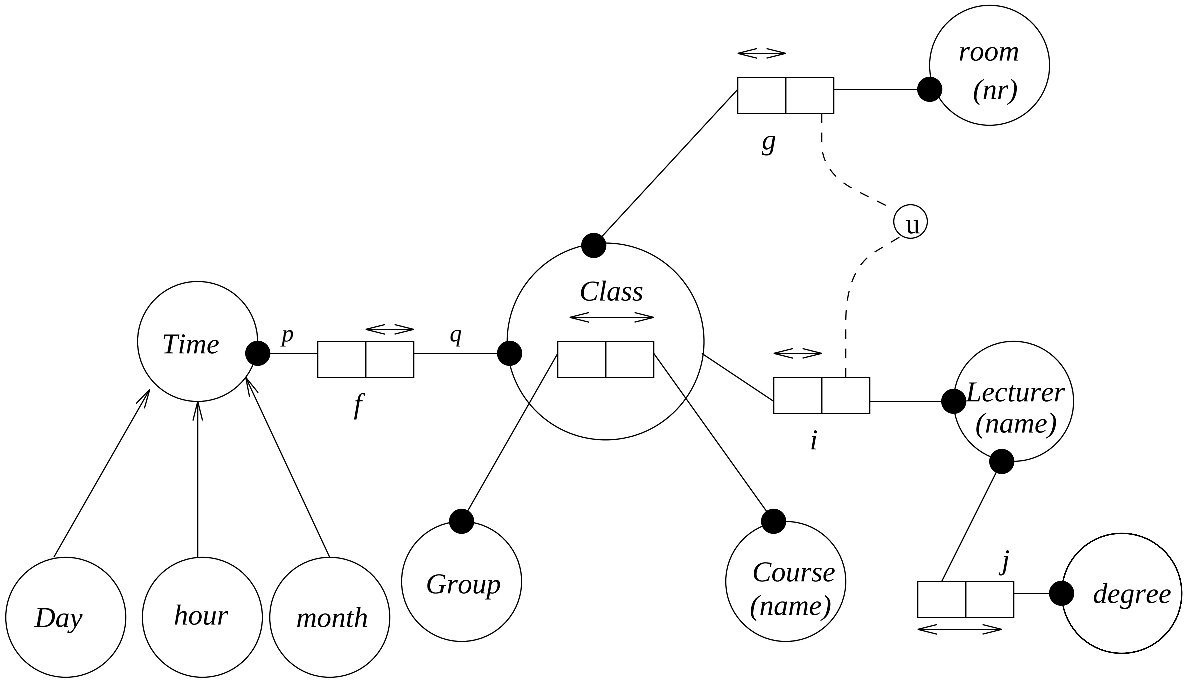


Figure 7.3: Example of an updated scheme

7.2 Transformation at relational database level

In this section a consistent PSM scheme with its corresponding relational database scheme will be updated. Once the conceptual model has been specified in a suitable language, an efficient realisation can (automatically) be derived, using a specification language that is more machine oriented [14].
To update a scheme means also to update its relational database scheme, this is achieved by performing operations on the scheme. Figure 7.4 shows the steps to be followed.

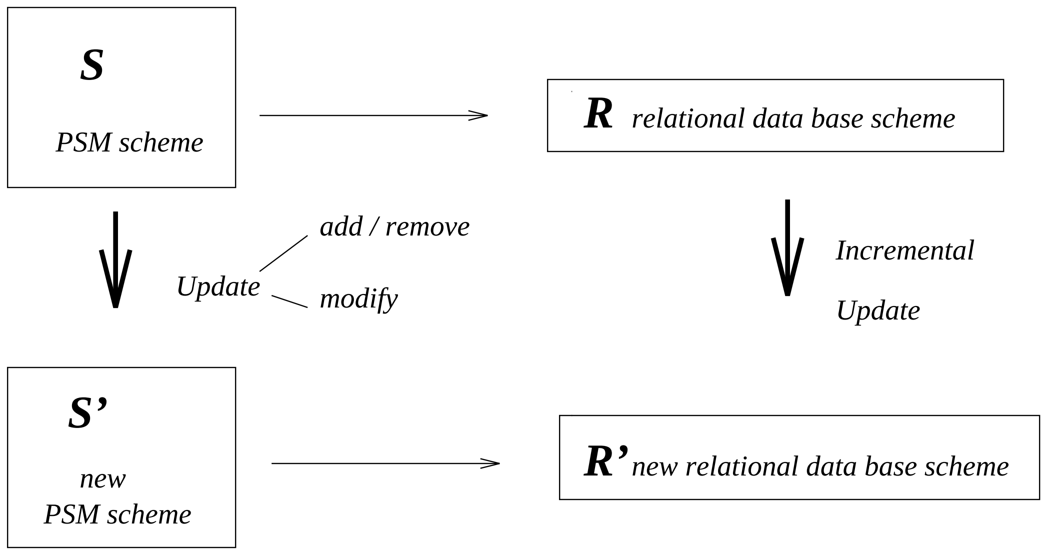
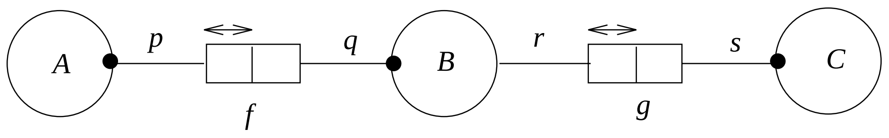


Figure 7.4: Transformation at relational level

The original PSM scheme **S** is related to the relational database **R**. The relational database **R'** is related to a new PSM scheme **S'**.
It must not be taken for granted that this specific relational database is the only candidate for representing the conceptual model **S** or any other model. The incremental update provides the new relational database **R'** with all record instances of the original relational database scheme, which it has 'saved' before the update took place.

How the transformation process can be guided in order to generate structures having certain predefined characteristics (like redundancy, optionals of size of the generated structure) can be found in [14]. In that chapter will also be a description of how a data structure can be recognised as a tree consisting of node of predicators. In this sense, the terminology used in this chapter concerning the relational representation will be referred to as a forest. Figure 7.5 shows an example of a tree representation.

This tree representation corresponds to relation type $[\overline{B}, A\mathtt{rep}, C\mathtt{op}]$, which in the



The forest representation of this scheme is :

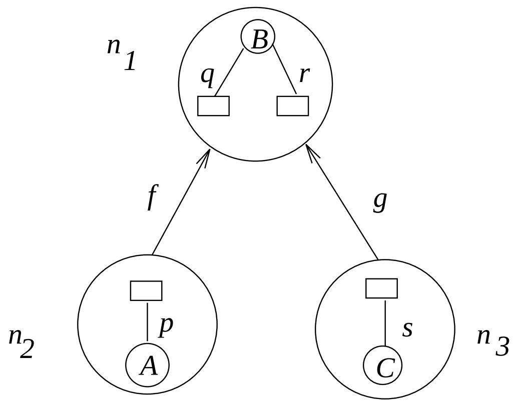


Figure 7.5: Example of a tree representation of a relational database

graph is: $[\overline{\{q,r\}}, \{p\}\texttt{rep}, \{s\}\texttt{op}]$.
The next heuristics will be applied to achieve this tranformation:

Input:

- A consistent PSM scheme **S**.
- A forest **R** of **S**.
- A sequence *U* of update commands.
  U will be applied to **S**.

Output:

- A new consistent PSM scheme **S'**.
- A mapping **R** on **R'** for **S'**.
- A re - arrange database which contains the new updates.

These heuristics can be achieved by applying the next steps:

Step 1:

Analyse the updates *U*, verifying whether or not the new PSM scheme **S'** they produce is correct:

- Produce the new PSM scheme **S'**, checking the constraints within the check environment, this environment can be determined according to the environment algorithm as described in Chapter 4.
- Reorganize the forest associated with **R** to reflect the updates.
  It must be kept in mind that the updates in some information structure $\mathcal{I}$ must be classified in advance. If the update concerns basically populations, then follow the 'during life' approach; If as consequence of population updates the structure $\mathcal{I}$ must change, then the 'during modelling process' approach is advised. All these updates concern constraint checking for which the environment algorithm can be used.
- Initiate the mapping between **R** and **R'** with operations to:

  1. Modify the current relation database scheme **R** leading to **R'**. At this point, the updated structure has taken new proportions for which is desirable to restrict the possibilities for generation of new forests.
     The new generated forest for $\mathcal{I}$ can be get by applying each object type of the graph to the algorithm:
     proc `GenerateForest`($\mathcal{I}$) : `Forest` ; as proposed in [14].
  2. Verify whether or not the current states, augmented with the new constraints, are consistent with the proposed changes.

- Reprocess step 1 to continue the reorganization of the new schemes.

Step 2:

- Compare the resulting PSM scheme to the old one in order to continue the generation of operations to:

  1. Check constraints with operations concerning uniqueness, *ORN*, identifiability, populatability, total and other constraints.
  2. Map the old representation **R** onto the new representation **R'** for **S'**. A forest alternative has been chosen and due to the incremental guidance, the mapping is meant not to compare the resulting forest sizes but rather to check whether the leaves of the former forest can be re-used in the new one.
  3. Map each record instance σ of **R** onto a consistent σ' of **R'**.

Consider the next example (figure 7.6), where a PSM scheme with its corresponding relational database is given.
The consequence of an update in the relational database is shown in figure 7.7.

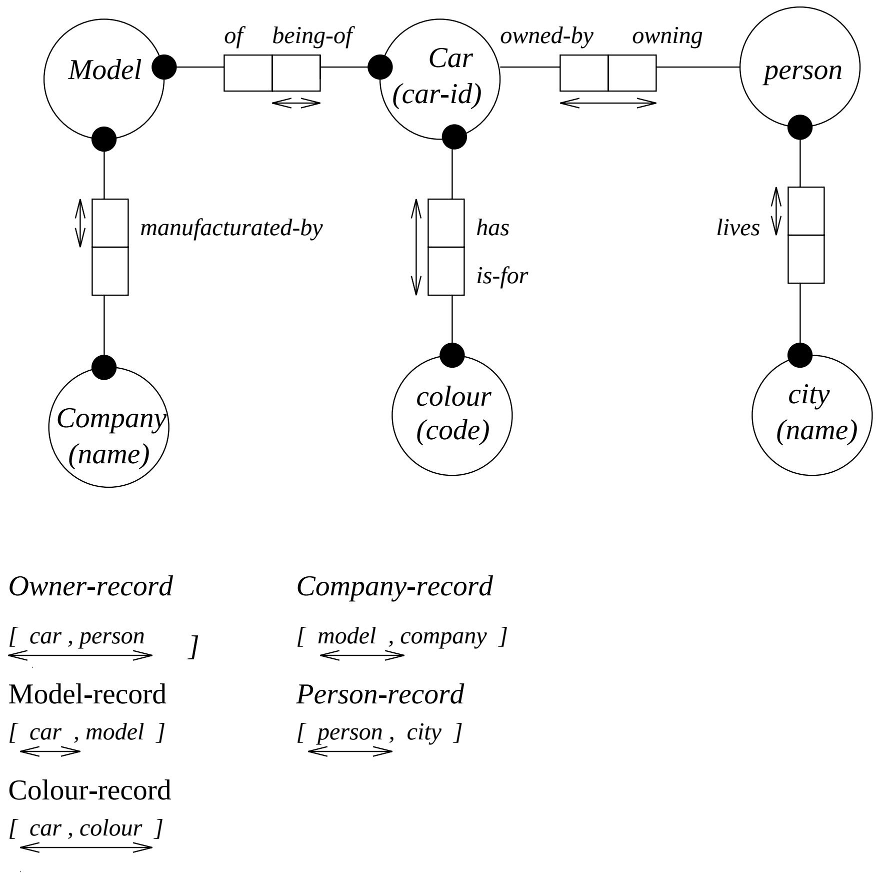
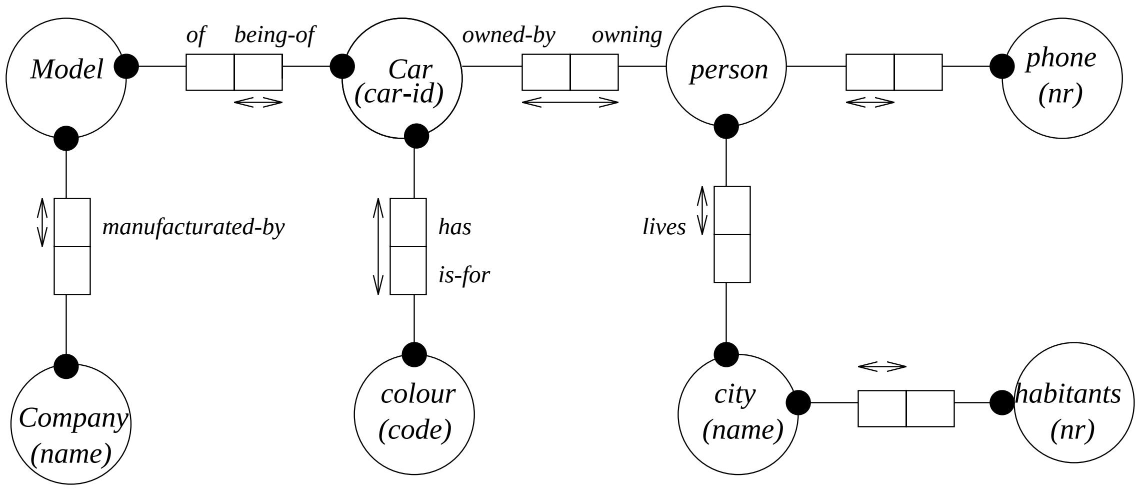


Figure 7.6: Example of scheme with relational database

The Atomic object types *Person* and *Community* have been extended with new object types which also will have to be included in the new relational database. See figure 7.7.
As shown in the previous examples, just one of the records has to be changed and one new record has to be added.
This can also be seen by using the tree structure of the relational database scheme, where the forest displays the resulting transformation which has been achieved in reduced steps. Figure 7.8 shows the tree representation concerning 7.6. Figure 7.9 shows the tree representation of the updated scheme.
Usually, the records generated by the objects outside of the environment remain in the same way in the new scheme.



Owner-record

[car , person]

Model-record

[car , model]

Colour-record

[car , colour]

Company-record

[model , company]

Person-record

[person , city , phone OP]

City-record

[city , nr-in-habitants]

Figure 7.7: Updated scheme and relational database

Due to the incremental approach, these untouched records can be 'saved' in order to be reused in the new relational PSM scheme.
The relation type which corresponds to the tree representation of figure 7.8 is:

$[\overline{Car}, [Model, Company]\,\texttt{rep}, Colour, [Person, City]\ \texttt{op}]$.
The relation type which corresponds to the tree representation of figure 7.9 is:

$[\overline{Car}, [Model, Company]\ \texttt{rep}, Colour, [\,[Person, phone\,\texttt{op}], [City, habitants\,\texttt{op}]\ \texttt{op}]\,]$.

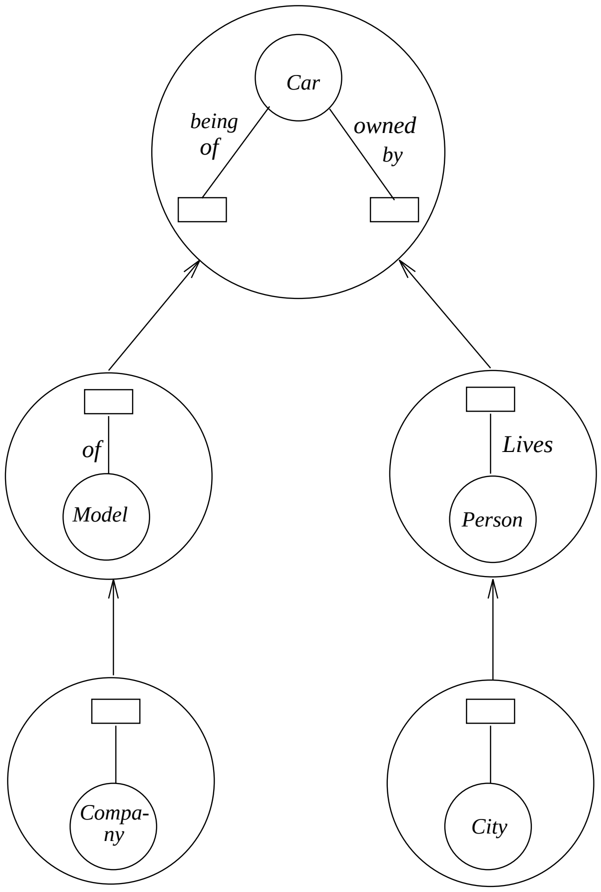


Figure 7.8: Updated scheme and relational database

7.3 Tranformation to less expressive data modelling techniques

The power of PSM becomes eclipsed by the lack of expressiveness in computer languages that support the direct translation from PSM schemes to relational database management schemes.
To obtain these relational databases, PSM schemes should first be translated to a less expressive data modelling technique like NIAM , because computer support does exist for this model.
This is rather a very expensive issue, on which reserarch has not finished and which therefore might be subject for future research.

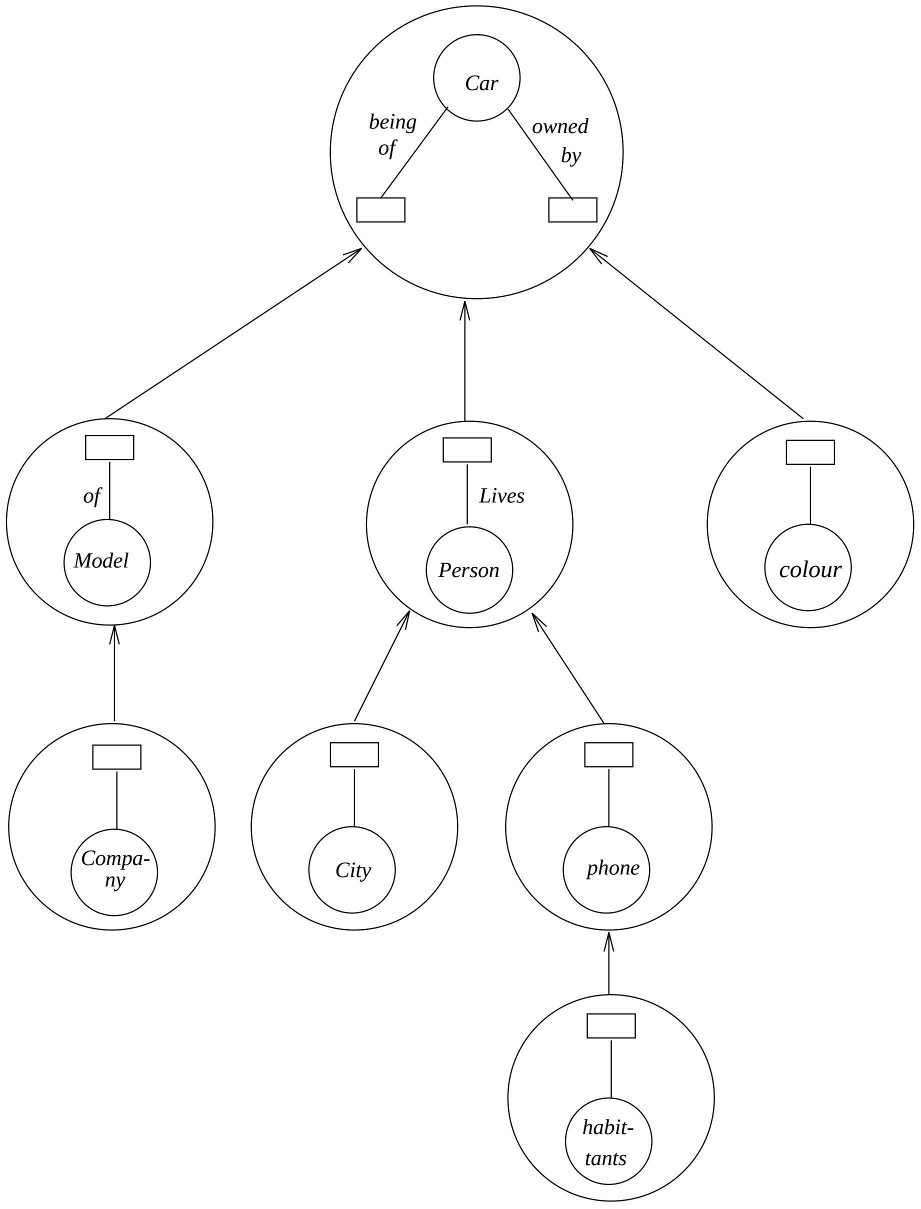


Figure 7.9: Updated scheme and relational database

Conclusions

In this thesis the current situation of incremental updates in databases has been depicted.
Different approaches like in the programming languages are the guidelines to update databases in parts, avoiding double work by checking the whole program. In this case, the whole PSM scheme.

Verification of PSM schemes, which are based on NIAM schemes, is a NP-problem; the possibility of several levels of inconsistency in these schemes exists, therefore they cannot be implemented efficiently in CASE-tools.
The constraints checking of a PSM scheme for an incremental update has been done on three different levels: during life, during modelling processing and during transformation.
Heuristics for updating a PSM scheme have been proposed. Before an update takes place, an environment check must be determined in order to limit the check area.

The comprehension *Chain* has been introduced. The part of the scheme which is out of the chain, remains untouched and does not need to be checked again.
A selection of the partial results of the incremental update process has been saved in order to increase the efficiency of the same update process. This selection is made by the chain determination.
The proposed heuristics can be applied to improve the relational database scheme as well.

Bibliography

[1] Paolo Atzeni and Riccardo Torlone. Updating intensional predicates in datalog. *Data and Knowledge Engineering 8*, 1992.

[2] H.A. Proper E. Falkenberg, J.Oei. Evolving information systems: Beyond temporal information systems. *SION. University of Nijmegen, The Netherlands*, 1992.

[3] Prof. E. Falkenberg. *Introduction to Information Analysis*. Katholieke Universiteit Nijmegen, Nederland, 1991.

[4] jr J. Bubenko and A. Olivé. *Information Systems: Theorical and formal aspects*. Elsevier science publishing company, INC, 1985.

[5] Magdi Kamel and Susan Davidson. Semi-materialization: a technique for optimizing frequently executed queries. *Data and Knowledge Engineering 6*, 1991.

[6] A. Laender M. Casanova, L. Tucherman. Algorithms for designing and maintaining optimazed relational representations of entity-relationschip schemas. *Rio scientific center IBM Brazil, Computer Science department, Federal University of Minas Gerais*, 1990. Brazil.

[7] Th. van der Weide P.van Bommel, A.ter Hofstede. Semantics and verification of object roles models. Technical report, 1993. The Netherlands.

[8] J. Rekers. Parser generation for interactive environments. Technical report, 1992. University of Amsterdam.

[9] I. Sommerville. *Software Engineering, third edition*. Addison-Wesley Publishing Company, 1990.

[10] A. ter Hofstede. Grondslagen van informatie systemen, part 1. *Katholieke universiteit Nijmegen, Nederland,* 1993.

[11] A.H. ter Hofstede and T. van der Weide. Expressiveness in conceptual data modelling. *Datak 159*, 1993. Elsevier, The Netherlands.

[12] A.H. ter Hofstede Th. van der Weide. *Formalisation of techniques: chopping down the methodology jungle*. Katholieke Universiteit Nijmegen, Nederland, 1993.

[13] M.B. Thuraisingham. Towards the design of a secure data / knowledge base management system. *Data and Knowledge Engineering 5*, 1990.

[14] P. van Bommel and Th. van der Weide. Reducing the search space for conceptual schema transformation. *Data and Knowledge Engineering 8*, 1992.

[15] M.C. van den Brand. *A generator for incremental programming environments*. cip-data kon.bib. The Hague, The Netherlands, 1992.

[16] N. Wirth. *Programming in Modula-2*. Springer - Verlag, 1992.

Index